

Christian Ullenboom

# Java

## ist auch eine Insel

Einführung, Ausbildung, Praxis

Erste  
Insel

- ▶ Programmieren mit der Java Platform, Standard Edition 11
- ▶ Java von A bis Z: Einführung, Praxis, Referenz
- ▶ Von Ausdrücken und Anweisungen zu Klassen und Objekten

Aktuell zu Java 11

14., aktualisierte und überarbeitete Auflage



Rheinwerk  
Computing

## Liebe Leserin, lieber Leser,

Sie halten nun die 14. Auflage unserer Insel in Händen. Seit ihrem ersten Erscheinen im Jahr 2001 im Rheinwerk Verlag, damals noch Galileo Press, ist sie das Standardwerk für alle Java-Entwickler und jene, die es werden wollen. Nirgendwo sonst bekommen Sie derart umfassende Informationen zu den Sprachgrundlagen, so viel nützliches Hintergrundwissen sowie zahlreiche Praxisbeispiele und Übungen wie hier. Die Insel ist hervorragend zum Selbststudium geeignet und ist zugleich die Java-Referenz für die tägliche Arbeit. Und sie ist natürlich immer up to date. Christian Ullensboom aktualisiert jede Auflage sehr gründlich und arbeitet die aktuellen Entwicklungen in der Java-Programmierung in sein Buch ein. Besonders wichtig sind ihm die Rückmeldungen seiner Leserinnen und Leser, deren Anregungen ebenfalls in jede neue Auflage einfließen.

Das dieser Auflage zugrundeliegende Java 11 ist die erste Long-Term-Support-Version seit Einführung der halbjährigen Java-Releases durch Oracle. Diese Version wird auch noch Jahre später unterstützt werden, sodass Sie eine solide Basis für die Java-Programmierung haben. Aus diesem Grund wird nun auch zum ersten Mal konsequent das OpenJDK verwendet. Damit ist sichergestellt, dass Sie auch in Zukunft unabhängig vom Lizenzmodell eines einzigen Anbieters Ihre Java-Anwendungen entwickeln und einsetzen können.

Jetzt wünsche ich Ihnen viel Freude beim Lesen unseres Kultbuches. Sie haben jederzeit die Möglichkeit, Anregungen und Kritik loszuwerden. Zögern Sie nicht, sich an Christian Ullensboom ([ullenboom@gmail.com](mailto:ullenboom@gmail.com)) oder an mich zu wenden. Wir sind gespannt auf Ihre Rückmeldung!

**Ihre Anne Scheibe**

Lektorat Rheinwerk Computing

[anne.scheibe@rheinwerk-verlag.de](mailto:anne.scheibe@rheinwerk-verlag.de)

[www.rheinwerk-verlag.de](http://www.rheinwerk-verlag.de)

Rheinwerk Verlag · Rheinwerkallee 4 · 53227 Bonn

# Hinweise zur Benutzung

Dieses E-Book ist **urheberrechtlich geschützt**. Mit dem Erwerb des E-Books haben Sie sich verpflichtet, die Urheberrechte anzuerkennen und einzuhalten. Sie sind berechtigt, dieses E-Book für persönliche Zwecke zu nutzen. Sie dürfen es auch ausdrucken und kopieren, aber auch dies nur für den persönlichen Gebrauch. Die Weitergabe einer elektronischen oder gedruckten Kopie an Dritte ist dagegen nicht erlaubt, weder ganz noch in Teilen. Und auch nicht eine Veröffentlichung im Internet oder in einem Firmennetzwerk.

Die ausführlichen und rechtlich verbindlichen Nutzungsbedingungen lesen Sie im Abschnitt *Rechtliche Hinweise*.

Dieses E-Book-Exemplar ist mit einem **digitalen Wasserzeichen** versehen, einem Vermerk, der kenntlich macht, welche Person dieses Exemplar nutzen darf:

Exemplar Nr. npkq-sbi6-jx4w-825r  
zum persönlichen Gebrauch für  
Oscar Ramirez,  
robayo.mauri@gmail.com

# Impressum

Dieses E-Book ist ein Verlagsprodukt, an dem viele mitgewirkt haben, insbesondere:

**Lektorat** Anne Scheibe

**Korrektorat** Petra Biedermann, Reken

**Herstellung E-Book** Denis Schaal

**Covergestaltung** Mai Loan Nguyen Duy

**Satz E-Book** SatzPro, Krefeld

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Bitte teilen Sie uns doch Ihre Meinung mit und lesen Sie weiter auf den *Serviceseiten*.

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 978-3-8362-6722-9 (E-Book)**

**ISBN 978-3-8362-6724-3 (Bundle)**

14., aktualisierte und überarbeitete Auflage 2019

© Rheinwerk Verlag GmbH, Bonn 2019

[www.rheinwerk-verlag.de](http://www.rheinwerk-verlag.de)

# Inhalt

Vorwort .....	31
---------------	----

## 1 Java ist auch eine Sprache 49

---

1.1 Historischer Hintergrund .....	49
1.2 Warum Java gut ist – die zentralen Eigenschaften .....	51
1.2.1 Bytecode .....	52
1.2.2 Ausführung des Bytecodes durch eine virtuelle Maschine .....	52
1.2.3 Plattformunabhängigkeit .....	53
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 .....	58
1.2.10 Bring den Müll raus, Garbage-Collector! .....	59
1.2.11 Ausnahmebehandlung .....	60
1.2.12 Angebot an Bibliotheken und Werkzeugen .....	61
1.2.13 Einfache Syntax der Programmiersprache Java .....	61
1.2.14 Java ist Open Source .....	63
1.2.15 Wofür sich Java weniger eignet .....	64
1.3 Java im Vergleich zu anderen Sprachen * .....	65
1.3.1 Java und C(++) .....	65
1.3.2 Java und JavaScript .....	66
1.3.3 Ein Wort zu Microsoft, Java und zu J++, J# .....	66
1.3.4 Java und C#/.NET .....	67
1.4 Weiterentwicklung und Verluste .....	68
1.4.1 Die Entwicklung von Java und seine Zukunftsaussichten .....	68
1.4.2 Features, Enhancements (Erweiterungen) und ein JSR .....	69
1.4.3 Applets .....	70
1.4.4 JavaFX .....	71
1.5 Java-Plattformen: Java SE, Jakarta EE, Java ME, Java Card .....	72
1.5.1 Die Java SE-Plattform .....	72
1.5.2 Java ME: Java für die Kleinen .....	74
1.5.3 Java für die ganz, ganz Kleinen .....	75

1.5.4	Java für die Großen: Java EE/Jakarta EE .....	75
1.5.5	Echtzeit-Java (Real-time Java) .....	76
<b>1.6</b>	<b>Java SE-Implementierungen</b> .....	77
1.6.1	OpenJDK .....	77
1.6.2	Oracle JDK .....	78
<b>1.7</b>	<b>Die Installation des Oracle OpenJDK</b> .....	81
1.7.1	OpenJDK unter Windows installieren .....	81
<b>1.8</b>	<b>Das erste Programm compilieren und testen</b> .....	83
1.8.1	Ein Quadratzahlen-Programm .....	84
1.8.2	Der Compilerlauf .....	85
1.8.3	Die Laufzeitumgebung .....	85
1.8.4	Häufige Compiler- und Interpreter-Probleme .....	86
<b>1.9</b>	<b>Entwicklungsumgebungen im Allgemeinen</b> .....	87
1.9.1	Eclipse IDE .....	87
1.9.2	IntelliJ IDEA .....	88
1.9.3	NetBeans .....	89
<b>1.10</b>	<b>Eclipse IDE im Speziellen</b> .....	89
1.10.1	Eclipse IDE entpacken und starten .....	91
1.10.2	Das erste Projekt anlegen .....	96
1.10.3	Verzeichnisstruktur für Java-Projekte * .....	98
1.10.4	Eine Klasse hinzufügen .....	99
1.10.5	Übersetzen und ausführen .....	100
1.10.6	Projekt einfügen, Workspace für die Programme wechseln .....	100
1.10.7	Plugins für Eclipse .....	101
<b>1.11</b>	<b>Zum Weiterlesen</b> .....	101

## **2 Imperative Sprachkonzepte** 103

---

<b>2.1</b>	<b>Elemente der Programmiersprache Java</b> .....	103
2.1.1	Token .....	103
2.1.2	Textkodierung durch Unicode-Zeichen .....	104
2.1.3	Bezeichner .....	104
2.1.4	Literale .....	107
2.1.5	(Reservierte) Schlüsselwörter .....	107
2.1.6	Zusammenfassung der lexikalischen Analyse .....	108
2.1.7	Kommentare .....	109
<b>2.2</b>	<b>Von der Klasse zur Anweisung</b> .....	111
2.2.1	Was sind Anweisungen? .....	111

2.2.2	Klassendeklaration .....	112
2.2.3	Die Reise beginnt am main(String[]) .....	113
2.2.4	Der erste Methodenaufruf: println(..) .....	113
2.2.5	Atomare Anweisungen und Anweisungssequenzen .....	114
2.2.6	Mehr zu print(..), println(..) und printf(..) für Bildschirmausgaben .....	115
2.2.7	Die API-Dokumentation .....	117
2.2.8	Ausdrücke .....	118
2.2.9	Ausdrucksanweisung .....	119
2.2.10	Erste Idee der Objektorientierung .....	120
2.2.11	Modifizierer .....	121
2.2.12	Gruppieren von Anweisungen mit Blöcken .....	121
<b>2.3</b>	<b>Datentypen, Typisierung, Variablen und Zuweisungen</b> .....	122
2.3.1	Primitive Datentypen im Überblick .....	125
2.3.2	Variablen Deklarationen .....	127
2.3.3	Automatisches Feststellen der Typen mit var .....	130
2.3.4	Konsoleneingaben .....	131
2.3.5	Fließkommazahlen mit den Datentypen float und double .....	133
2.3.6	Ganzzahlige Datentypen .....	135
2.3.7	Wahrheitswerte .....	137
2.3.8	Unterstriche in Zahlen *	137
2.3.9	Alphanumerische Zeichen .....	138
2.3.10	Gute Namen, schlechte Namen .....	138
2.3.11	Initialisierung von lokalen Variablen .....	139
<b>2.4</b>	<b>Ausdrücke, Operanden und Operatoren</b> .....	140
2.4.1	Zuweisungsoperator .....	141
2.4.2	Arithmetische Operatoren .....	142
2.4.3	Unäres Minus und Plus .....	146
2.4.4	Präfix- oder Postfix-Inkrement und -Dekrement .....	147
2.4.5	Zuweisung mit Operation (Verbundoperator) .....	149
2.4.6	Die relationalen Operatoren und die Gleichheitssymbole .....	150
2.4.7	Logische Operatoren: Nicht, Und, Oder, XOR .....	152
2.4.8	Kurzschluss-Operatoren .....	153
2.4.9	Der Rang der Operatoren in der Auswertungsreihenfolge .....	155
2.4.10	Die Typumwandlung (das Casting) .....	158
2.4.11	Überladenes Plus für Strings .....	164
2.4.12	Operator vermisst *	165
<b>2.5</b>	<b>Bedingte Anweisungen oder Fallunterscheidungen</b> .....	166
2.5.1	Verzweigung mit der if-Anweisung .....	166
2.5.2	Die Alternative mit einer if-else-Anweisung wählen .....	169
2.5.3	Der Bedingungsoperator .....	173
2.5.4	Die switch-Anweisung bietet die Alternative .....	175

<b>2.6</b>	<b>Immer das Gleiche mit den Schleifen</b>	182
2.6.1	Die while-Schleife	182
2.6.2	Die do-while-Schleife	184
2.6.3	Die for-Schleife	186
2.6.4	Schleifenbedingungen und Vergleiche mit == *	190
2.6.5	Schleifenabbruch mit break und zurück zum Test mit continue	192
2.6.6	break und continue mit Marken *	195
<b>2.7</b>	<b>Methoden einer Klasse</b>	199
2.7.1	Bestandteil einer Methode	200
2.7.2	Signatur-Beschreibung in der Java-API	201
2.7.3	Aufruf einer Methode	203
2.7.4	Methoden ohne Parameter deklarieren	203
2.7.5	Statische Methoden (Klassenmethoden)	204
2.7.6	Parameter, Argument und Wertübergabe	205
2.7.7	Methoden vorzeitig mit return beenden	207
2.7.8	Nicht erreichbarer Quellcode bei Methoden *	208
2.7.9	Methoden mit Rückgaben	209
2.7.10	Methoden überladen	214
2.7.11	Gültigkeitsbereich	216
2.7.12	Vorgegebener Wert für nicht aufgeführte Argumente *	217
2.7.13	Finale lokale Variablen	218
2.7.14	Rekursive Methoden *	219
2.7.15	Die Türme von Hanoi *	224
<b>2.8</b>	<b>Zum Weiterlesen</b>	226

## 3 Klassen und Objekte

---

<b>3.1</b>	<b>Objektorientierte Programmierung (OOP)</b>	227
3.1.1	Warum überhaupt OOP?	227
3.1.2	Denk ich an Java, denk ich an Wiederverwendbarkeit	228
<b>3.2</b>	<b>Eigenschaften einer Klasse</b>	229
3.2.1	Klassenarbeit mit Point	230
<b>3.3</b>	<b>Natürlich modellieren mit der UML (Unified Modeling Language) *</b>	230
3.3.1	Hintergrund und Geschichte der UML *	231
3.3.2	Wichtige Diagrammtypen der UML *	232
3.3.3	UML-Werkzeuge *	233
<b>3.4</b>	<b>Neue Objekte erzeugen</b>	234
3.4.1	Ein Exemplar einer Klasse mit dem Schlüsselwort new anlegen	235

3.4.2	Der Zusammenhang von new, Heap und Garbage-Collector .....	235
3.4.3	Deklarieren von Referenzvariablen .....	236
3.4.4	Jetzt mach mal 'nen Punkt: Zugriff auf Objektattribute und -methoden .....	238
3.4.5	Überblick über Point-Methoden .....	242
3.4.6	Konstruktoren nutzen .....	245
<b>3.5</b>	<b>ZZZZnake .....</b>	<b>246</b>
<b>3.6</b>	<b>Pakete schnüren, Importe und Kompilationseinheiten .....</b>	<b>249</b>
3.6.1	Java-Pakete .....	249
3.6.2	Pakete der Standardbibliothek .....	249
3.6.3	Volle Qualifizierung und import-Deklaration .....	249
3.6.4	Mit import p1.p2.* alle Typen eines Pakets erreichen .....	251
3.6.5	Hierarchische Strukturen über Pakete .....	251
3.6.6	Die package-Deklaration .....	252
3.6.7	Unbenanntes Paket (default package) .....	253
3.6.8	Klassen mit gleichen Namen in unterschiedlichen Paketen * .....	254
3.6.9	Kompilationseinheit (Compilation Unit) .....	254
3.6.10	Statischer Import * .....	255
<b>3.7</b>	<b>Mit Referenzen arbeiten, Identität und Gleichheit (Gleichwertigkeit) .....</b>	<b>256</b>
3.7.1	null-Referenz und die Frage der Philosophie .....	256
3.7.2	Alles auf null? Referenzen testen .....	259
3.7.3	Zuweisungen bei Referenzen .....	260
3.7.4	Methoden mit Referenztypen als Parametern .....	261
3.7.5	Identität von Objekten .....	265
3.7.6	Gleichheit (Gleichwertigkeit) und die Methode equals(..) .....	266
<b>3.8</b>	<b>Zum Weiterlesen .....</b>	<b>268</b>

## 4 Arrays und ihre Anwendungen

---

<b>4.1</b>	<b>Arrays .....</b>	<b>269</b>
4.1.1	Grundbestandteile .....	269
4.1.2	Deklaration von Array-Variablen .....	270
4.1.3	Array-Objekte mit new erzeugen .....	271
4.1.4	Arrays mit Inhalt .....	272
4.1.5	Die Länge eines Arrays über das Attribut length auslesen .....	273
4.1.6	Zugriff auf die Elemente über den Index .....	274
4.1.7	Typische Array-Fehler .....	276
4.1.8	Arrays als Methodenparameter .....	277
4.1.9	Vorinitialisierte Arrays .....	278
4.1.10	Die erweiterte for-Schleife .....	279

4.1.11	Arrays mit nichtprimitiven Elementen .....	281
4.1.12	Methode mit variabler Argumentanzahl (Varargs) .....	283
4.1.13	Mehrdimensionale Arrays * .....	285
4.1.14	Nichtrechteckige Arrays * .....	289
4.1.15	Die Wahrheit über die Array-Initialisierung * .....	291
4.1.16	Mehrere Rückgabewerte * .....	292
4.1.17	Klonen kann sich lohnen – Arrays vermehren * .....	293
4.1.18	Array-Inhalte kopieren * .....	294
4.1.19	Die Klasse Arrays zum Vergleichen, Füllen, Suchen, Sortieren nutzen .....	295
4.1.20	Eine lange Schlange .....	308
<b>4.2</b>	<b>Der Einstiegspunkt für das Laufzeitsystem: main(..)</b> .....	311
4.2.1	Korrekte Deklaration der Startmethode .....	311
4.2.2	Kommandozeilenargumente verarbeiten .....	312
4.2.3	Der Rückgabetyp von main(..) und System.exit(int)* .....	313
<b>4.3</b>	<b>Zum Weiterlesen</b> .....	315

## 5 Der Umgang mit Zeichenketten 317

---

<b>5.1</b>	<b>Von ASCII über ISO-8859-1 zu Unicode</b> .....	317
5.1.1	ASCII .....	317
5.1.2	ISO/IEC 8859-1 .....	318
5.1.3	Unicode .....	319
5.1.4	Unicode-Zeichenkodierung .....	321
5.1.5	Escape-Sequenzen/Fluchtsymbole .....	322
5.1.6	Schreibweise für Unicode-Zeichen und Unicode-Escapes .....	323
5.1.7	Java-Versionen gehen mit Unicode-Standard Hand in Hand * .....	325
<b>5.2</b>	<b>Die Character-Klasse</b> .....	326
5.2.1	Ist das so? .....	326
5.2.2	Zeichen in Großbuchstaben/Kleinbuchstaben konvertieren .....	329
5.2.3	Vom Zeichen zum String .....	330
5.2.4	Von char in int: vom Zeichen zur Zahl * .....	330
<b>5.3</b>	<b>Zeichenfolgen</b> .....	332
<b>5.4</b>	<b>Die Klasse String und ihre Methoden</b> .....	334
5.4.1	String-Literale als String-Objekte für konstante Zeichenketten .....	334
5.4.2	Konkatenation mit + .....	334
5.4.3	String-Länge und Test auf Leer-String .....	335
5.4.4	Zugriff auf ein bestimmtes Zeichen mit charAt(int) .....	336
5.4.5	Nach enthaltenen Zeichen und Zeichenfolgen suchen .....	337

5.4.6	Das Hangman-Spiel .....	340
5.4.7	Gut, dass wir verglichen haben .....	342
5.4.8	String-Teile extrahieren .....	346
5.4.9	Strings anhängen, zusammenfügen, Groß-/Kleinschreibung und Weißraum	350
5.4.10	Gesucht, gefunden, ersetzt .....	354
5.4.11	String-Objekte mit Konstruktoren und aus Wiederholungen erzeugen * .....	356
<b>5.5</b>	<b>Veränderbare Zeichenketten mit StringBuilder und StringBuffer</b> .....	360
5.5.1	Anlegen von StringBuilder-Objekten .....	361
5.5.2	StringBuilder in andere Zeichenkettenformate konvertieren .....	362
5.5.3	Zeichen(folgen) erfragen .....	362
5.5.4	Daten anhängen .....	362
5.5.5	Zeichen(folgen) setzen, löschen und umdrehen .....	364
5.5.6	Länge und Kapazität eines StringBuilder-Objekts *	367
5.5.7	Vergleich von StringBuilder-Exemplaren und String mit StringBuilder .....	368
5.5.8	hashCode() bei StringBuilder *	370
<b>5.6</b>	<b>CharSequence als Basistyp</b> .....	370
<b>5.7</b>	<b>Konvertieren zwischen Primitiven und Strings</b> .....	373
5.7.1	Unterschiedliche Typen in String-Repräsentationen konvertieren .....	373
5.7.2	String-Inhalt in einen primitiven Wert konvertieren .....	375
5.7.3	String-Repräsentation im Format Binär, Hex, Oktal *	377
5.7.4	parseXXX(..)- und printXXX()-Methoden in DatatypeConverter *	381
<b>5.8</b>	<b>Strings zusammenhängen (konkatenieren)</b> .....	382
5.8.1	Strings mit StringJoiner zusammenhängen .....	383
<b>5.9</b>	<b>Zerlegen von Zeichenketten</b> .....	384
5.9.1	Splitten von Zeichenketten mit split(..)	385
5.9.2	Yes we can, yes we scan – die Klasse Scanner .....	386
<b>5.10</b>	<b>Ausgaben formatieren</b> .....	390
5.10.1	Formatieren und Ausgeben mit format()	390
<b>5.11</b>	<b>Zum Weiterlesen</b> .....	396

---

## 6 Eigene Klassen schreiben

---

<b>6.1</b>	<b>Eigene Klassen mit Eigenschaften deklarieren</b> .....	397
6.1.1	Attribute deklarieren .....	398
6.1.2	Methoden deklarieren .....	400
6.1.3	Verdeckte (shadowed) Variablen .....	404
6.1.4	Die this-Referenz .....	405

<b>6.2</b>	<b>Privatsphäre und Sichtbarkeit .....</b>	409
6.2.1	Für die Öffentlichkeit: public .....	409
6.2.2	Kein Public Viewing – Passwörter sind privat .....	410
6.2.3	Wieso nicht freie Methoden und Variablen für alle? .....	411
6.2.4	Privat ist nicht ganz privat: Es kommt darauf an, wer's sieht * .....	412
6.2.5	Zugriffsmethoden für Attribute deklarieren .....	412
6.2.6	Setter und Getter nach der JavaBeans-Spezifikation .....	413
6.2.7	Paketsichtbar .....	415
6.2.8	Zusammenfassung zur Sichtbarkeit .....	417
<b>6.3</b>	<b>Eine für alle – statische Methoden und statische Attribute .....</b>	419
6.3.1	Warum statische Eigenschaften sinnvoll sind .....	420
6.3.2	Statische Eigenschaften mit static .....	421
6.3.3	Statische Eigenschaften über Referenzen nutzen? * .....	422
6.3.4	Warum die Groß- und Kleinschreibung wichtig ist * .....	423
6.3.5	Statische Variablen zum Datenaustausch * .....	423
6.3.6	Statische Eigenschaften und Objekteigenschaften * .....	425
<b>6.4</b>	<b>Konstanten und Aufzählungen .....</b>	426
6.4.1	Konstanten über statische finale Variablen .....	426
6.4.2	Typsichere Aufzählungen .....	427
6.4.3	Aufzählungstypen: typsichere Aufzählungen mit enum .....	429
<b>6.5</b>	<b>Objekte anlegen und zerstören .....</b>	434
6.5.1	Konstruktoren schreiben .....	434
6.5.2	Verwandtschaft von Methode und Konstruktor .....	436
6.5.3	Der Standard-Konstruktor (default constructor) .....	436
6.5.4	Parametrisierte und überladene Konstruktoren .....	438
6.5.5	Copy-Konstruktor .....	440
6.5.6	Einen anderen Konstruktor der gleichen Klasse mit this(...) aufrufen .....	441
6.5.7	Ihr fehlt uns nicht – der Garbage-Collector .....	444
<b>6.6</b>	<b>Klassen- und Objektinitialisierung * .....</b>	446
6.6.1	Initialisierung von Objektvariablen .....	446
6.6.2	Statische Blöcke als Klasseninitialisierer .....	448
6.6.3	Initialisierung von Klassenvariablen .....	449
6.6.4	Eincompilierte Belegungen der Klassenvariablen .....	450
6.6.5	Exemplarinitialisierer (Instanzinitialisierer) .....	451
6.6.6	Finale Werte im Konstruktor und in statischen Blöcken setzen .....	454
<b>6.7</b>	<b>Zum Weiterlesen .....</b>	456

## 7 Objektorientierte Beziehungsfragen

457

<b>7.1</b>	<b>Assoziationen zwischen Objekten</b>	457
7.1.1	Unidirektionale 1:1-Beziehung .....	458
7.1.2	Zwei Freunde müsst ihr werden – bidirektionale 1:1-Beziehungen .....	459
7.1.3	Unidirektionale 1:n-Beziehung .....	460
<b>7.2</b>	<b>Vererbung</b> .....	464
7.2.1	Vererbung in Java .....	464
7.2.2	Spielobjekte modellieren .....	465
7.2.3	Die implizite Basisklasse <code>java.lang.Object</code> .....	467
7.2.4	Einfach- und Mehrfachvererbung * .....	467
7.2.5	Sehen Kinder alles? Die Sichtbarkeit <code>protected</code> .....	468
7.2.6	Konstruktoren in der Vererbung und <code>super(..)</code> .....	469
<b>7.3</b>	<b>Typen in Hierarchien</b> .....	475
7.3.1	Automatische und explizite Typumwandlung .....	475
7.3.2	Das Substitutionsprinzip .....	478
7.3.3	Typen mit dem <code>instanceof</code> -Operator testen .....	480
<b>7.4</b>	<b>Methoden überschreiben</b> .....	482
7.4.1	Methoden in Unterklassen mit neuem Verhalten ausstatten .....	482
7.4.2	Mit <code>super</code> an die Eltern .....	487
7.4.3	Finale Klassen und finale Methoden .....	489
7.4.4	Kovariante Rückgabetypen .....	491
7.4.5	Array-Typen und Kovarianz * .....	492
<b>7.5</b>	<b>Drum prüfe, wer sich dynamisch bindet</b> .....	493
7.5.1	Gebunden an <code>toString()</code> .....	494
7.5.2	Implementierung von <code>System.out.println(Object)</code> .....	496
7.5.3	Nicht dynamisch gebunden bei privaten, statischen und finalen Methoden .....	497
7.5.4	Dynamisch gebunden auch bei Konstruktoraufrufen * .....	498
7.5.5	Eine letzte Spielerei mit Javas dynamischer Bindung und überdeckten Attributen * .....	500
<b>7.6</b>	<b>Abstrakte Klassen und abstrakte Methoden</b> .....	502
7.6.1	Abstrakte Klassen .....	502
7.6.2	Abstrakte Methoden .....	504
<b>7.7</b>	<b>Schnittstellen</b> .....	510
7.7.1	Schnittstellen sind neue Typen .....	510
7.7.2	Schnittstellen deklarieren .....	510
7.7.3	Abstakte Methoden in Schnittstellen .....	511

7.7.4	Implementieren von Schnittstellen .....	512
7.7.5	Ein Polymorphie-Beispiel mit Schnittstellen .....	514
7.7.6	Die Mehrfachvererbung bei Schnittstellen .....	515
7.7.7	Keine Kollisionsgefahr bei Mehrfachvererbung * .....	520
7.7.8	Erweitern von Interfaces – Subinterfaces .....	521
7.7.9	Konstantendeklarationen bei Schnittstellen .....	522
7.7.10	Nachträgliches Implementieren von Schnittstellen * .....	524
7.7.11	Statische ausprogrammierte Methoden in Schnittstellen .....	525
7.7.12	Erweitern und Ändern von Schnittstellen .....	527
7.7.13	Default-Methoden .....	529
7.7.14	Erweiterte Schnittstellen deklarieren und nutzen .....	530
7.7.15	Öffentliche und private Schnittstellenmethoden .....	533
7.7.16	Erweiterte Schnittstellen, Mehrfachvererbung und Mehrdeutigkeiten * .....	533
7.7.17	Bausteine bilden mit Default-Methoden * .....	537
7.7.18	Initialisierung von Schnittstellenkonstanten * .....	543
7.7.19	Markierungsschnittstellen * .....	547
7.7.20	(Abstrakte) Klassen und Schnittstellen im Vergleich .....	548
<b>7.8</b>	<b>SOLIDe Modellierung .....</b>	549
7.8.1	DRY, KISS und YAGNI .....	549
7.8.2	SOLID .....	549
7.8.3	Sei nicht STUPID .....	551
<b>7.9</b>	<b>Zum Weiterlesen .....</b>	552

## 8 Ausnahmen müssen sein

---

<b>8.1</b>	<b>Problembereiche einzäunen .....</b>	553
8.1.1	Exceptions in Java mit try und catch .....	554
8.1.2	Eine NumberFormatException auffangen .....	554
8.1.3	Bitte nicht schlucken – leere catch-Blöcke .....	558
8.1.4	Wiederholung abgebrochener Bereiche * .....	558
8.1.5	Mehrere Ausnahmen auffangen .....	559
8.1.6	Ablauf einer Ausnahmesituation .....	562
8.1.7	throws im Methodenkopf angeben .....	562
8.1.8	Abschlussbehandlung mit finally .....	564
<b>8.2</b>	<b>Die Klassenhierarchie der Ausnahmen .....</b>	569
8.2.1	Eigenschaften des Exception-Objekts .....	569
8.2.2	Basistyp Throwable .....	570

8.2.3	Die Exception-Hierarchie .....	571
8.2.4	Oberausnahmen auffangen .....	572
8.2.5	Schon gefangen? .....	574
8.2.6	Alles geht als Exception durch .....	574
8.2.7	Zusammenfassen gleicher catch-Blöcke mit dem multi-catch .....	576
<b>8.3</b>	<b>RuntimeException muss nicht aufgefangen werden</b> .....	579
8.3.1	Beispiele für RuntimeException-Klassen .....	580
8.3.2	Kann man abfangen, muss man aber nicht .....	581
<b>8.4</b>	<b>Harte Fehler – Error *</b> .....	581
<b>8.5</b>	<b>Auslösen eigener Exceptions</b> .....	582
8.5.1	Mit throw Ausnahmen auslösen .....	582
8.5.2	Vorhandene Runtime-Ausnahmetypen kennen und nutzen .....	584
8.5.3	Parameter testen und gute Fehlermeldungen .....	587
8.5.4	Neue Exception-Klassen deklarieren .....	589
8.5.5	Eigene Ausnahmen als Unterklassen von Exception oder RuntimeException? .....	590
8.5.6	Ausnahmen abfangen und weiterleiten *	593
8.5.7	Aufruf-Stack von Ausnahmen verändern *	594
8.5.8	Präzises rethrow *	595
8.5.9	Geschachtelte Ausnahmen *	599
<b>8.6</b>	<b>Automatisches Ressourcen-Management (try mit Ressourcen)</b> .....	602
8.6.1	try mit Ressourcen .....	602
8.6.2	Die Schnittstelle AutoCloseable .....	605
8.6.3	Mehrere Ressourcen nutzen .....	607
8.6.4	try mit Ressourcen auf null-Ressourcen .....	608
8.6.5	Unterdrückte Ausnahmen *	608
<b>8.7</b>	<b>Besonderheiten bei der Ausnahmebehandlung *</b> .....	611
8.7.1	Rückgabewerte bei ausgelösten Ausnahmen .....	612
8.7.2	Ausnahmen und Rückgaben verschwinden – das Duo return und finally .....	612
8.7.3	throws bei überschriebenen Methoden .....	613
8.7.4	Nicht erreichbare catch-Klauseln .....	615
<b>8.8</b>	<b>Assertions *</b> .....	617
8.8.1	Assertions in eigenen Programmen nutzen .....	617
8.8.2	Assertions aktivieren und Laufzeit-Errors .....	617
8.8.3	Assertions feiner aktivieren oder deaktivieren .....	620
<b>8.9</b>	<b>Zum Weiterlesen</b> .....	621

<b>9 Geschachtelte Typen</b>	623
<b>9.1 Geschachtelte Klassen, Schnittstellen, Aufzählungen</b>	623
<b>9.2 Statische geschachtelte Typen</b>	624
<b>9.3 Nichtstatische geschachtelte Typen</b>	626
9.3.1 Exemplare innerer Klassen erzeugen .....	626
9.3.2 Die this-Referenz .....	627
9.3.3 Vom Compiler generierte Klassendateien * .....	628
9.3.4 Erlaubte Modifizierer bei äußeren und inneren Klassen .....	629
<b>9.4 Lokale Klassen</b>	629
9.4.1 Beispiel mit eigener Klassendeklaration .....	630
9.4.2 Lokale Klasse für einen Timer nutzen .....	631
<b>9.5 Anonyme innere Klassen</b>	631
9.5.1 Nutzung einer anonymen inneren Klasse für den Timer .....	632
9.5.2 Umsetzung innerer anonymer Klassen * .....	633
9.5.3 Konstruktoren innerer anonymer Klassen .....	634
<b>9.6 Zugriff auf lokale Variablen aus lokalen und anonymen Klassen *</b>	636
<b>9.7 this in Unterklassen *</b>	637
9.7.1 Geschachtelte Klassen greifen auf private Eigenschaften zu .....	638
<b>9.8 Nester</b>	640
<b>9.9 Zum Weiterlesen</b>	641
<b>10 Besondere Typen der Java SE</b>	643
<b>10.1 Object ist die Mutter aller Klassen</b>	644
10.1.1 Klassenobjekte .....	644
10.1.2 Objektidentifikation mit <code>toString()</code> .....	645
10.1.3 Objektgleichheit mit <code>equals(..)</code> und Identität .....	647
10.1.4 Klonen eines Objekts mit <code>clone()</code> * .....	653
10.1.5 Hashwerte über <code>hashCode()</code> liefern * .....	658
10.1.6 <code>System.identityHashCode(..)</code> und das Problem der nicht eindeutigen Objektverweise * .....	665
10.1.7 Aufräumen mit <code>finalize()</code> * .....	666
10.1.8 Synchronisation * .....	668
<b>10.2 Schwache Referenzen und Cleaner</b>	669

<b>10.3 Die Utility-Klasse <code>java.util.Objects</code></b> .....	670
10.3.1 Eingebaute null-Tests für <code>equals(...)</code> / <code>hashCode()</code> .....	670
10.3.2 <code>Objects.toString(..)</code> .....	671
10.3.3 null-Prüfungen mit eingebauter Ausnahmebehandlung .....	672
10.3.4 Tests auf null .....	673
10.3.5 Indexbezogene Programmargumente auf Korrektheit prüfen .....	673
<b>10.4 Vergleichen von Objekten und Ordnung herstellen</b> .....	674
10.4.1 Natürlich geordnet oder nicht? .....	674
10.4.2 <code>compareXXX()</code> -Methode der Schnittstellen Comparable und Comparator ....	675
10.4.3 Rückgabewerte kodieren die Ordnung .....	676
10.4.4 Beispiel-Comparator: den kleinsten Raum einer Sammlung finden .....	677
10.4.5 Tipps für Comparator und Comparable-Implementierungen .....	679
10.4.6 Statische und Default-Methoden in Comparator .....	680
<b>10.5 Wrapper-Klassen und Autoboxing</b> .....	683
10.5.1 Wrapper-Objekte erzeugen .....	685
10.5.2 Konvertierungen in eine String-Repräsentation .....	687
10.5.3 Von einer String-Repräsentation parsen .....	688
10.5.4 Die Basisklasse Number für numerische Wrapper-Objekte .....	688
10.5.5 Vergleiche durchführen mit <code>compareXXX(..)</code> , <code>compareTo(..)</code> , <code>equals(..)</code> und Hashwerten .....	690
10.5.6 Statische Reduzierungsmethoden in Wrapper-Klassen .....	693
10.5.7 Konstanten für die Größe eines primitiven Typs .....	694
10.5.8 Behandeln von vorzeichenlosen Zahlen * .....	694
10.5.9 Die Klasse Integer .....	696
10.5.10 Die Klassen Double und Float für Fließkommazahlen .....	697
10.5.11 Die Long-Klasse .....	697
10.5.12 Die Boolean-Klasse .....	697
10.5.13 Autoboxing: Boxing und Unboxing .....	699
<b>10.6 Iterator, Iterable *</b> .....	703
10.6.1 Die Schnittstelle Iterator .....	703
10.6.2 Wer den Iterator liefert .....	706
10.6.3 Die Schnittstelle Iterable .....	707
10.6.4 Erweitertes for und Iterable .....	708
10.6.5 Interne Iteration .....	708
10.6.6 Einen eigenen Iterable implementieren * .....	709
<b>10.7 Die Spezial-Oberklasse Enum</b> .....	710
10.7.1 Methoden auf Enum-Objekten .....	711
10.7.2 Aufzählungen mit eigenen Methoden und Initialisierern * .....	714
10.7.3 enum mit eigenen Konstruktoren * .....	717

<b>10.8 Annotationen in der Java SE .....</b>	720
10.8.1 Orte für Annotationen .....	721
10.8.2 Annotationstypen aus java.lang .....	721
10.8.3 @Deprecated .....	722
10.8.4 Annotationen mit zusätzlichen Informationen .....	722
10.8.5 @SuppressWarnings .....	723
<b>10.9 Zum Weiterlesen .....</b>	726

---

## 11 Generics<T>

---

<b>11.1 Einführung in Java Generics .....</b>	727
11.1.1 Mensch versus Maschine – Typprüfung des Compilers und der Laufzeitumgebung .....	727
11.1.2 Raketen .....	728
11.1.3 Generische Typen deklarieren .....	730
11.1.4 Generics nutzen .....	731
11.1.5 Diamonds are forever .....	734
11.1.6 Generische Schnittstellen .....	737
11.1.7 Generische Methoden/Konstruktoren und Typ-Inferenz .....	739
<b>11.2 Umsetzen der Generics, Typlösung und Raw-Types .....</b>	743
11.2.1 Realisierungsmöglichkeiten .....	743
11.2.2 Typlösung (Type Erasure) .....	743
11.2.3 Probleme der Typlösung .....	745
11.2.4 Raw-Type .....	750
<b>11.3 Einschränken der Typen über Bounds .....</b>	752
11.3.1 Einfache Einschränkungen mit extends .....	753
11.3.2 Weitere Obertypen mit & .....	755
<b>11.4 Typparameter in der throws-Klausel * .....</b>	756
11.4.1 Deklaration einer Klasse mit Typvariable <E extends Exception> .....	756
11.4.2 Parametrisierter Typ bei Typvariable <E extends Exception> .....	756
<b>11.5 Generics und Vererbung, Invarianz .....</b>	759
11.5.1 Arrays sind kovariant .....	759
11.5.2 Generics sind nicht kovariant, sondern invariant .....	760
11.5.3 Wildcards mit ? .....	761
11.5.4 Bounded Wildcards .....	763
11.5.5 Bounded-Wildcard-Typen und Bounded-Typvariablen .....	767
11.5.6 Das LESS-Prinzip .....	769
11.5.7 Enum<E extends Enum<E>> * .....	771

<b>11.6 Konsequenzen der Typlöschung: Typ-Token, Arrays und Brücken *</b> .....	773
11.6.1 Typ-Token .....	773
11.6.2 Super-Type-Token .....	775
11.6.3 Generics und Arrays .....	776
11.6.4 Brückenmethoden .....	777
<b>11.7 Zum Weiterlesen .....</b>	783

## 12 Lambda-Ausdrücke und funktionale Programmierung

785

<b>12.1 Code = Daten .....</b>	785
<b>12.2 Funktionale Schnittstellen und Lambda-Ausdrücke im Detail .....</b>	788
12.2.1 Funktionale Schnittstellen .....	789
12.2.2 Typ eines Lambda-Ausdrucks ergibt sich durch Zieltyp .....	790
12.2.3 Annotation @FunctionalInterface .....	794
12.2.4 Syntax für Lambda-Ausdrücke .....	795
12.2.5 Die Umgebung der Lambda-Ausdrücke und Variablenzugriffe .....	800
12.2.6 Ausnahmen in Lambda-Ausdrücken .....	804
12.2.7 Klassen mit einer abstrakten Methode als funktionale Schnittstelle? * .....	808
<b>12.3 Methodenreferenz .....</b>	809
12.3.1 Varianten von Methodenreferenzen .....	811
<b>12.4 Konstruktorreferenz .....</b>	813
12.4.1 Parameterlose und parametrisierte Konstruktoren .....	815
12.4.2 Nützliche vordefinierte Schnittstellen für Konstruktorreferenzen .....	815
<b>12.5 Implementierung von Lambda-Ausdrücken *</b> .....	816
<b>12.6 Funktionale Programmierung mit Java .....</b>	817
12.6.1 Programmierparadigmen: imperativ oder deklarativ .....	817
12.6.2 Funktionale Programmierung und funktionale Programmiersprachen .....	818
12.6.3 Funktionale Programmierung in Java am Beispiel vom Comparator .....	819
12.6.4 Lambda-Ausdrücke als Funktionen sehen .....	820
<b>12.7 Funktionale Schnittstelle aus dem java.util.function-Paket .....</b>	821
12.7.1 Blöcke mit Code und die funktionale Schnittstelle Consumer .....	822
12.7.2 Supplier .....	824
12.7.3 Prädikate und java.util.function.Predicate .....	824
12.7.4 Funktionen über die funktionale Schnittstelle java.util.function.Function ....	826
12.7.5 Ein bisschen Bi ... .....	830
12.7.6 Funktionale Schnittstellen mit Primitiven .....	833

<b>12.8 Optional ist keine Nullnummer .....</b>	836
12.8.1 Optional-Typ .....	838
12.8.2 Primitive optionale Typen .....	841
12.8.3 Erstmal funktional mit Optional .....	842
<b>12.9 Was ist jetzt so funktional? .....</b>	847
<b>12.10 Zum Weiterlesen .....</b>	849

## 13 Architektur, Design und angewandte Objektorientierung

851

<b>13.1 Architektur, Design und Implementierung .....</b>	851
<b>13.2 Design-Patterns (Entwurfsmuster) .....</b>	852
13.2.1 Motivation für Design-Patterns .....	852
13.2.2 Singleton .....	853
13.2.3 Fabrikmethoden .....	854
13.2.4 Das Beobachter-Pattern mit Listener realisieren .....	856
<b>13.3 Zum Weiterlesen .....</b>	860

## 14 Komponenten, JavaBeans und Module

861

<b>14.1 JavaBeans .....</b>	861
14.1.1 Properties (Eigenschaften) .....	862
14.1.2 Einfache Eigenschaften .....	863
14.1.3 Indizierte Eigenschaften .....	863
14.1.4 Gebundene Eigenschaften und PropertyChangeListener .....	863
14.1.5 Veto-Eigenschaften – dagegen! .....	867
<b>14.2 Klassenlader (Class Loader) und Modul-/Klassenpfad .....</b>	870
14.2.1 Klassenladen auf Abruf .....	871
14.2.2 Klassenlader bei der Arbeit zusehen .....	871
14.2.3 JMOD-Dateien und JAR-Dateien .....	872
14.2.4 Woher die kleinen Klassen kommen: die Suchorte und spezielle Klassenlader .....	873
14.2.5 Setzen des Modulpfades .....	874
<b>14.3 Module entwickeln und einbinden .....</b>	876
14.3.1 Wer sieht wen .....	876
14.3.2 Plattform-Module und JMOD-Beispiel .....	878

14.3.3	Interne Plattformeigenschaften nutzen, --add-exports .....	878
14.3.4	Neue Module einbinden, --add-modules und --add-opens .....	881
14.3.5	Projektabhängigkeiten in Eclipse .....	882
14.3.6	Benannte Module und module-info.java .....	884
14.3.7	Automatische Module .....	887
14.3.8	Unbenanntes Modul .....	888
14.3.9	Lesbarkeit und Zugreifbarkeit .....	889
14.3.10	Modul-Migration .....	890
<b>14.4</b>	<b>Zum Weiterlesen .....</b>	<b>891</b>

---

## **15 Die Klassenbibliothek** 893

<b>15.1</b>	<b>Die Java-Klassenphilosophie .....</b>	<b>893</b>
15.1.1	Modul, Paket, Typ .....	893
15.1.2	Übersicht über die Pakete der Standardbibliothek .....	896
<b>15.2</b>	<b>Einfache Zeitmessung und Profiling *</b> .....	<b>901</b>
<b>15.3</b>	<b>Die Klasse Class .....</b>	<b>904</b>
15.3.1	An ein Class-Objekt kommen .....	904
15.3.2	Eine Class ist ein Type .....	906
<b>15.4</b>	<b>Klassenlader .....</b>	<b>907</b>
15.4.1	Die Klasse java.lang.ClassLoader .....	908
<b>15.5</b>	<b>Die Utility-Klassen System und Properties .....</b>	<b>908</b>
15.5.1	Speicher der JVM .....	910
15.5.2	Anzahl CPUs/Kerne .....	911
15.5.3	Systemeigenschaften der Java-Umgebung .....	911
15.5.4	Eigene Properties von der Konsole aus setzen * .....	913
15.5.5	Zeilenumbruchzeichen, line.separator .....	915
15.5.6	Umgebungsvariablen des Betriebssystems .....	916
<b>15.6</b>	<b>Sprachen der Länder .....</b>	<b>917</b>
15.6.1	Sprachen in Regionen über Locale-Objekte .....	917
<b>15.7</b>	<b>Wichtige Datum-Klassen im Überblick .....</b>	<b>922</b>
15.7.1	Der 1.1.1970 .....	922
15.7.2	System.currentTimeMillis() .....	923
15.7.3	Einfache Zeitumrechnungen durch TimeUnit .....	923
<b>15.8</b>	<b>Date-Time-API .....</b>	<b>924</b>
15.8.1	Menschenzeit und Maschinenzeit .....	926
15.8.2	Datumsklasse LocalDate .....	929

<b>15.9 Logging mit Java .....</b>	930
15.9.1 Logging-APIs .....	930
15.9.2 Logging mit java.util.logging .....	931
<b>15.10 Maven: Build-Management und Abhängigkeiten auflösen .....</b>	933
15.10.1 Beispieldprojekt in Eclipse mit Maven .....	934
15.10.2 Properties hinzunehmen .....	934
15.10.3 Dependency hinzunehmen .....	935
15.10.4 Lokales und Remote-Repository .....	936
15.10.5 Lebenszyklus, Phasen und Maven-Plugins .....	936
15.10.6 Archetypes .....	937
<b>15.11 Zum Weiterlesen .....</b>	937

---

## 16 Einführung in die nebenläufige Programmierung

---

<b>16.1 Nebenläufigkeit und Parallelität .....</b>	939
16.1.1 Multitasking, Prozesse, Threads .....	940
16.1.2 Threads und Prozesse .....	940
16.1.3 Wie nebenläufige Programme die Geschwindigkeit steigern können .....	941
16.1.4 Was Java für Nebenläufigkeit alles bietet .....	943
<b>16.2 Laufende Threads, neue Threads erzeugen .....</b>	943
16.2.1 Main-Thread .....	944
16.2.2 Wer bin ich? .....	944
16.2.3 Die Schnittstelle Runnable implementieren .....	944
16.2.4 Thread mit Runnable starten .....	946
16.2.5 Runnable parametrisieren .....	947
16.2.6 Die Klasse Thread erweitern .....	948
<b>16.3 Thread-Eigenschaften und Zustände .....</b>	950
16.3.1 Der Name eines Threads .....	951
16.3.2 Die Zustände eines Threads * .....	951
16.3.3 Schläfer gesucht .....	952
16.3.4 Mit yield() und onSpinWait() auf Rechenzeit verzichten .....	954
16.3.5 Wann Threads fertig sind .....	955
16.3.6 Einen Thread höflich mit Interrupt beenden .....	955
16.3.7 Unbehandelte Ausnahmen, Thread-Ende und UncaughtExceptionHandler .....	958
16.3.8 Der stop() von außen und die Rettung mit ThreadDeath * .....	959
16.3.9 Ein Rendezvous mit join(...) * .....	960
16.3.10 Arbeit niederlegen und wieder aufnehmen * .....	962
16.3.11 Priorität * .....	963

<b>16.4 Der Ausführer (Executor) kommt .....</b>	964
16.4.1 Die Schnittstelle Executor .....	965
16.4.2 Glücklich in der Gruppe – die Thread-Pools .....	966
16.4.3 Threads mit Rückgabe über Callable .....	968
16.4.4 Erinnerungen an die Zukunft – die Future-Rückgabe .....	970
16.4.5 Mehrere Callable-Objekte abarbeiten .....	973
16.4.6 CompletionService und ExecutorCompletionService .....	975
16.4.7 ScheduledExecutorService für wiederholende Aufgaben und Zeitsteuerungen nutzen .....	976
<b>16.5 Zum Weiterlesen .....</b>	977

## 17 Einführung in Datenstrukturen und Algorithmen 979

---

<b>17.1 Listen .....</b>	979
17.1.1 Erstes Listen-Beispiel .....	980
17.1.2 Auswahlkriterium ArrayList oder LinkedList .....	981
17.1.3 Die Schnittstelle List .....	981
17.1.4 ArrayList .....	988
17.1.5 LinkedList .....	989
17.1.6 Der Array-Adapter Arrays.asList(..) .....	991
17.1.7 ListIterator * .....	993
17.1.8 toArray(..) von Collection verstehen – die Gefahr einer Falle erkennen .....	994
17.1.9 Primitive Elemente in Datenstrukturen verwalten .....	998
<b>17.2 Mengen (Sets) .....</b>	998
17.2.1 Ein erstes Mengen-Beispiel .....	999
17.2.2 Methoden der Schnittstelle Set .....	1001
17.2.3 HashSet .....	1003
17.2.4 TreeSet – die sortierte Menge .....	1003
17.2.5 Die Schnittstellen NavigableSet und SortedSet .....	1005
17.2.6 LinkedHashSet .....	1008
<b>17.3 Java Stream API .....</b>	1009
17.3.1 Deklaratives Programmieren .....	1009
17.3.2 Interne versus externe Iteration .....	1009
17.3.3 Was ist ein Stream? .....	1010
<b>17.4 Stream erzeugen .....</b>	1012
17.4.1 Parallele oder sequenzielle Streams .....	1015
<b>17.5 Terminale Stream-Operationen .....</b>	1016
17.5.1 Anzahl Elemente .....	1016

17.5.2	Und jetzt alle – forEachXXX(..) .....	1017
17.5.3	Einzelne Elemente aus dem Strom holen .....	1017
17.5.4	Existenztests mit Prädikaten .....	1018
17.5.5	Strom reduzieren auf kleinstes/größtes Element .....	1018
17.5.6	Strom mit eigenen Funktionen reduzieren .....	1019
17.5.7	Ergebnisse in einen Container schreiben, Teil 1: collect(..) .....	1021
17.5.8	Ergebnisse in einen Container schreiben, Teil 2: Collector und Collectors .....	1022
17.5.9	Ergebnisse in einen Container schreiben, Teil 3: Gruppierungen .....	1024
17.5.10	Stream-Elemente in Array oder Iterator übertragen .....	1026
<b>17.6</b>	<b>Intermediäre Stream-Operationen .....</b>	<b>1027</b>
17.6.1	Element-Vorschau .....	1028
17.6.2	Filtrern von Elementen .....	1028
17.6.3	Statusbehaftete intermediäre Operationen .....	1028
17.6.4	Präfix-Operation .....	1030
17.6.5	Abbildungen .....	1031
<b>17.7</b>	<b>Zum Weiterlesen .....</b>	<b>1033</b>

## **18 Einführung in grafische Oberflächen**

---

<b>18.1</b>	<b>GUI-Frameworks .....</b>	<b>1035</b>
18.1.1	Kommandozeile .....	1035
18.1.2	Grafische Benutzeroberfläche .....	1035
18.1.3	Abstract Window Toolkit (AWT) .....	1036
18.1.4	Java Foundation Classes und Swing .....	1036
18.1.5	JavaFX .....	1036
18.1.6	SWT (Standard Widget Toolkit) * .....	1038
<b>18.2</b>	<b>Deklarative und programmierte Oberflächen .....</b>	<b>1039</b>
18.2.1	GUI-Beschreibungen in JavaFX .....	1040
18.2.2	Deklarative GUI-Beschreibungen für Swing? .....	1040
<b>18.3</b>	<b>GUI-Builder .....</b>	<b>1041</b>
18.3.1	GUI-Builder für JavaFX .....	1041
18.3.2	GUI-Builder für Swing .....	1042
<b>18.4</b>	<b>Mit dem Eclipse WindowBuilder zur ersten Swing-Oberfläche .....</b>	<b>1042</b>
18.4.1	WindowBuilder installieren .....	1042
18.4.2	Mit WindowBuilder eine GUI-Klasse hinzufügen .....	1044
18.4.3	Layoutprogramm starten .....	1046
18.4.4	Grafische Oberfläche aufbauen .....	1047
18.4.5	Swing-Komponenten-Klassen .....	1049
18.4.6	Funktionalität geben .....	1051

<b>18.5 Grundlegendes zum Zeichnen .....</b>	1054
18.5.1 Die paint(Graphics)-Methode für das AWT-Frame .....	1054
18.5.2 Die ereignisorientierte Programmierung ändert Fensterinhalte .....	1056
18.5.3 Zeichnen von Inhalten auf einen JFrame .....	1057
18.5.4 Auffordern zum Neuzeichnen mit repaint(..) .....	1058
18.5.5 Java 2D-API .....	1059
<b>18.6 Zum Weiterlesen .....</b>	1060

---

## 19 Einführung in Dateien und Datenströme 1061

<b>19.1 Alte und neue Welt in java.io und java.nio .....</b>	1061
19.1.1 java.io-Paket mit File-Klasse .....	1061
19.1.2 NIO.2 und java.nio-Paket .....	1062
19.1.3 java.io.File oder java.nio.*? .....	1062
<b>19.2 Dateisysteme und Pfade .....</b>	1063
19.2.1 FileSystem und Path .....	1063
19.2.2 Die Utility-Klasse Files .....	1069
<b>19.3 Dateien mit wahlfreiem Zugriff .....</b>	1072
19.3.1 Ein RandomAccessFile zum Lesen und Schreiben öffnen .....	1072
19.3.2 Aus dem RandomAccessFile lesen .....	1073
19.3.3 Schreiben mit RandomAccessFile .....	1076
19.3.4 Die Länge des RandomAccessFile .....	1076
19.3.5 Hin und her in der Datei .....	1077
<b>19.4 Basisklassen für die Ein-/Ausgabe .....</b>	1078
19.4.1 Die vier abstrakten Basisklassen .....	1078
19.4.2 Die abstrakte Basisklasse OutputStream .....	1079
19.4.3 Die abstrakte Basisklasse InputStream .....	1081
19.4.4 Die abstrakte Basisklasse Writer .....	1084
19.4.5 Die Schnittstelle Appendable * .....	1086
19.4.6 Die abstrakte Basisklasse Reader .....	1087
19.4.7 Die Schnittstellen Closeable, AutoCloseable und Flushable .....	1090
<b>19.5 Lesen aus Dateien und Schreiben in Dateien .....</b>	1092
19.5.1 Byteorientierte Datenströme über Files beziehen .....	1092
19.5.2 Zeichenorientierte Datenströme über Files beziehen .....	1093
19.5.3 Funktion von OpenOption bei den Files.newXXX(..)-Methoden .....	1095
19.5.4 Ressourcen aus dem Modulpfad und aus JAR-Dateien laden .....	1096
<b>19.6 Zum Weiterlesen .....</b>	1098

<b>20 Einführung ins Datenbankmanagement mit JDBC</b>	1099
<b>20.1 Relationale Datenbanken und Datenbankmanagementsysteme</b> .....	1099
20.1.1 Das relationale Modell .....	1099
20.1.2 H2-Datenbank .....	1100
20.1.3 Weitere Datenbank-Management-Systeme * .....	1100
<b>20.2 JDBC und Datenbanktreiber</b> .....	1102
20.2.1 H2-Datenbank und JDBC-Treiber .....	1103
20.2.2 JDBC-Versionen * .....	1103
<b>20.3 Eine Beispielabfrage</b> .....	1104
20.3.1 Schritte zur Datenbankabfrage .....	1104
20.3.2 Mit Java auf die relationale Datenbank zugreifen .....	1105
<b>20.4 Zum Weiterlesen</b> .....	1106
<b>21 Einführung in &lt;XML&gt;</b>	1107
<b>21.1 Auszeichnungssprachen</b> .....	1107
21.1.1 Extensible Markup Language (XML) .....	1108
<b>21.2 Eigenschaften von XML-Dokumenten</b> .....	1108
21.2.1 Elemente und Attribute .....	1108
21.2.2 Beschreibungssprache für den Aufbau von XML-Dokumenten .....	1111
21.2.3 Schema – die moderne Alternative zu DTD .....	1115
21.2.4 Namensraum (Namespace) .....	1118
21.2.5 XML-Applikationen * .....	1119
<b>21.3 Die Java-APIs für XML</b> .....	1120
21.3.1 Das Document Object Model (DOM) .....	1121
21.3.2 Pull-API StAX .....	1121
21.3.3 Simple API for XML Parsing (SAX) .....	1121
21.3.4 Java Document Object Model (JDOM) .....	1122
21.3.5 JAXP als Java-Schnittstelle zu XML .....	1122
21.3.6 DOM-Bäume einlesen mit JAXP * .....	1123
<b>21.4 Java Architecture for XML Binding (JAXB)</b> .....	1124
21.4.1 JAXB raus aus der Java SE, Abhängigkeit rein in die POM .....	1124
21.4.2 Bean für JAXB aufbauen .....	1125
21.4.3 Utility-Klasse JAXB .....	1126

21.4.4	Ganze Objektgraphen schreiben und lesen .....	1127
21.4.5	JAXBContext und Marshaller/Unmarshaller nutzen .....	1129
<b>21.5</b>	<b>Zum Weiterlesen .....</b>	<b>1131</b>

---

## **22 Bits und Bytes, Mathematisches und Geld**

1133

<b>22.1</b>	<b>Bits und Bytes .....</b>	<b>1133</b>
22.1.1	Die Bit-Operatoren Komplement, Und, Oder und XOR .....	1134
22.1.2	Repräsentation ganzer Zahlen in Java – das Zweierkomplement .....	1135
22.1.3	Das binäre (Basis 2), oktale (Basis 8), hexadezimale (Basis 16) Stellenwertsystem .....	1137
22.1.4	Auswirkung der Typumwandlung auf die Bit-Muster .....	1138
22.1.5	Vorzeichenlos arbeiten .....	1140
22.1.6	Die Verschiebeoperatoren .....	1143
22.1.7	Ein Bit setzen, löschen, umdrehen und testen .....	1146
22.1.8	Bit-Methoden der Integer- und Long-Klasse .....	1146
<b>22.2</b>	<b>Fließkomma-Arithmetik in Java .....</b>	<b>1148</b>
22.2.1	Spezialwerte für Unendlich, Null, NaN .....	1148
22.2.2	Standardnotation und wissenschaftliche Notation bei Fließkommazahlen * .....	1152
22.2.3	Mantisse und Exponent * .....	1152
<b>22.3</b>	<b>Die Eigenschaften der Klasse Math .....</b>	<b>1154</b>
22.3.1	Attribute .....	1154
22.3.2	Absolutwerte und Vorzeichen .....	1154
22.3.3	Maximum/Minimum .....	1155
22.3.4	Runden von Werten .....	1156
22.3.5	Rest der ganzzahligen Division * .....	1158
22.3.6	Division mit Rundung Richtung negativ unendlich, alternativer Restwert * .....	1159
22.3.7	Multiply-Accumulate .....	1161
22.3.8	Wurzel- und Exponentialmethoden .....	1161
22.3.9	Der Logarithmus * .....	1162
22.3.10	Winkelmethoden * .....	1163
22.3.11	Zufallszahlen .....	1165
<b>22.4</b>	<b>Genauigkeit, Wertebereich eines Typs und Überlaufkontrolle *</b>	<b>1165</b>
22.4.1	Der größte und der kleinste Wert .....	1165
22.4.2	Überlauf und alles ganz exakt .....	1166
22.4.3	Was bitte macht eine ulp? .....	1169

<b>22.5 Zufallszahlen: Random, SecureRandom, SplittableRandom</b> .....	1170
22.5.1 Die Klasse Random .....	1171
22.5.2 Random-Objekte mit dem Samen aufbauen .....	1171
22.5.3 Einzelne Zufallszahlen erzeugen .....	1172
22.5.4 Pseudo-Zufallszahlen in der Normalverteilung * .....	1172
22.5.5 Strom von Zufallszahlen generieren * .....	1173
22.5.6 Die Klasse SecureRandom * .....	1174
22.5.7 SplittableRandom * .....	1175
<b>22.6 Große Zahlen *</b> .....	1175
22.6.1 Die Klasse BigInteger .....	1176
22.6.2 Beispiel: ganz lange Fakultäten mit BigInteger .....	1183
22.6.3 Große Fließkommazahlen mit BigDecimal .....	1184
22.6.4 Mit MathContext komfortabel die Rechengenauigkeit setzen .....	1187
22.6.5 Noch schneller rechnen durch mutable Implementierungen .....	1188
<b>22.7 Mathe bitte strikt *</b> .....	1188
22.7.1 Strikte Fließkommaberechnungen mit strictfp .....	1189
22.7.2 Die Klassen Math und StrictMath .....	1189
<b>22.8 Geld und Währung</b> .....	1190
22.8.1 Geldbeträge repräsentieren .....	1190
22.8.2 ISO 4217 .....	1190
22.8.3 Währungen in Java repräsentieren .....	1191
<b>22.9 Zum Weiterlesen</b> .....	1192
<b>23 Testen mit JUnit</b> .....	1193
<b>23.1 Softwaretests</b> .....	1193
23.1.1 Vorgehen beim Schreiben von Testfällen .....	1194
<b>23.2 Das Test-Framework JUnit</b> .....	1194
23.2.1 Test-Driven Development und Test-First .....	1195
23.2.2 Testen, implementieren, testen, implementieren, testen, freuen .....	1196
23.2.3 JUnit-Tests ausführen .....	1198
23.2.4 assertXXX(...)-Methoden der Klasse Assert .....	1198
23.2.5 Hamcrest .....	1201
23.2.6 Exceptions testen .....	1205
23.2.7 Tests ignorieren .....	1207
23.2.8 Grenzen für Ausführungszeiten festlegen .....	1207
23.2.9 Mit Methoden der Assumptions-Klasse Tests abbrechen .....	1208
<b>23.3 Wie gutes Design das Testen ermöglicht</b> .....	1208

<b>23.4 Aufbau größerer Testfälle .....</b>	1210
23.4.1 Fixtures .....	1211
23.4.2 Sammlungen von Testklassen und Klassenorganisation .....	1213
<b>23.5 Dummy, Fake, Stub und Mock .....</b>	1214
<b>23.6 JUnit-Erweiterungen, Testzusätze .....</b>	1215
<b>23.7 Zum Weiterlesen .....</b>	1216

---

## 24 Die Werkzeuge des JDK

---

<b>24.1 Übersicht .....</b>	1217
24.1.1 Aufbau und gemeinsame Schalter .....	1218
<b>24.2 Java-Quellen übersetzen .....</b>	1218
24.2.1 Java-Compiler vom JDK .....	1218
24.2.2 Alternative Compiler .....	1219
24.2.3 Native Compiler .....	1220
24.2.4 Java-Programme in ein natives ausführbares Programm einpacken .....	1220
<b>24.3 Die Java-Laufzeitumgebung .....</b>	1221
24.3.1 Schalter der JVM .....	1221
24.3.2 Der Unterschied zwischen java.exe und javaw.exe .....	1224
<b>24.4 jlink: der Java Linker .....</b>	1224
<b>24.5 Dokumentationskommentare mit Javadoc .....</b>	1225
24.5.1 Einen Dokumentationskommentar setzen .....	1226
24.5.2 Mit dem Werkzeug javadoc eine Dokumentation erstellen .....	1228
24.5.3 HTML-Tags in Dokumentationskommentaren * .....	1229
24.5.4 Generierte Dateien .....	1229
24.5.5 Dokumentationskommentare im Überblick * .....	1229
24.5.6 Javadoc und Doclets * .....	1231
24.5.7 Veraltete (deprecated) Typen und Eigenschaften .....	1231
24.5.8 Javadoc-Überprüfung mit DocLint .....	1234
<b>24.6 Das Archivformat JAR .....</b>	1235
24.6.1 Das Dienstprogramm jar benutzen .....	1236
24.6.2 Das Manifest .....	1238
24.6.3 Applikationen in JAR-Archiven starten .....	1239
<b>24.7 Zum Weiterlesen .....</b>	1240
 Anhang .....	1241
Index .....	1261



# 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 Anfang war das Wort. Viel später, am 23. Mai 1995, stellten auf der SunWorld in San Francisco der Chef des 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, plattformunabhängig zu programmieren und objektorientiert unsere Gedanken auszudrücken. Die Möglichkeiten der Sprache und Bibliotheken sind an sich nichts Neues, aber so gut verpackt, dass Java angenehm und flüssig zu programmieren ist und Java heute zur populärsten Programmiersprache auf unserem Planeten zählt. Dieses Buch beschäftigt sich in 24 Kapiteln mit der Java-Technologie und intensiv mit der Programmiersprache Java. Wichtige Themen sind objektorientierte Programmierung, Design von Klassen und der Aufbau der Java-Standardbibliotheken.

## 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! Objektorientiertes Vorwissen ist von Vorteil, aber nicht notwendig.

### 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 ver-

ständlichen Tiefe, doch sie ersetzt nicht die *Java Language Specification* (JLS: <http://tutego.de/go/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 JavaFX oder Swing beschäftigt – beides Teile von Standard-Java –, ist genauso umfangreich wie die jetzige Insel. Nicht anders verhält es sich mit den anderen Spezialthemen, wie etwa objektorientierter Analyse/Design, UML, paralleler oder verteilter Programmierung, Enterprise JavaBeans, Datenbankanbindung, OR-Mapping, Webservices, 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 auf 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 geht über 15 Jahre zurück und hängt 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 hatte ich rund zehn Jahre in Assembler kodiert, 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 hatte. Doch bis heute macht mir Java immer noch viel Spaß, und ich freue mich auf jede neue Version. Und mit meinem Optimismus bin ich nicht allein: Die Prognosen für Java stehen ungebrochen gut, weil der Einsatz von Java mittlerweile so gefestigt ist wie der von COBOL bei Banken und Versicherungen. Daher wird immer wieder davon gesprochen, Java sei das neue COBOL.

Nach über fünfzehn Jahren sehe ich die Insel heute 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.<sup>1</sup> Die Erweiterung der Insel macht mir Spaß, auch wenn viele Themen kaum in einem normalen Java-Kurs angesprochen werden.



## Software und Versionen

Als Grundlage für dieses Buch dient die *Java Platform Standard Edition* (Java SE) in der Version 11 in der freien Implementierung OpenJDK, wobei JDK für *Java Development Kit* steht. Das JDK besteht im Wesentlichen aus einem Compiler und einer Laufzeitumgebung (JVM) und ist für die Plattformen Windows, macOS, Linux und Solaris erhältlich. Ist das System keines der genannten, gibt es Laufzeitumgebungen von anderen Unternehmen bzw. vom Hersteller der Plattform: IBM bietet für IBM System i (ehemals iSeries) zum Beispiel eine eigene 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.

---

<sup>1</sup> Und vermutlich gibt es weltweit kein anderes IT-Fachbuch, das so viele unanständige Wörter im Text versteckt.

### Welche Java-Version verwenden wir?

Seit Oracle (damals noch von Sun geführt) die Programmiersprache Java 1995 mit Version 1.0 vorstellte, hat sich die Versionsspirale bis Version 11 gedreht. Besonders für Java-Buch-Autoren stellt sich die Frage, auf welcher Java-Version ihr Text aufbauen muss und welche Bibliotheken er beschreiben soll. Ich beschreibe immer die Möglichkeiten der neuesten Version, was zur Drucklegung die Java SE 11 war. Für die Didaktik der objektorientierten Programmierung ist die Versionsfrage glücklicherweise unerheblich.

Da viele Unternehmen noch mit der Java-Version 8 entwickeln, wirft die breite Nutzung von Features der Java-Version 11 unter Umständen Probleme auf, denn nicht jedes Beispielprogramm aus der Insel lässt sich per Copy & Paste fehlerfrei in das eigene Projekt übertragen. Da Java 10 und Java 11 kleine Änderungen an der Sprache mitbringen, kann ein Java 8-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 11 zwar ausführlich in den einzelnen Kapiteln beschrieben, aber die Beispielprogramme in anderen Kapiteln bleiben auf dem Sprachniveau von Java 8 und nutzen zum Beispiel nicht die verkürzte Variablen-deklaration mit var. So laufen mehrheitlich alle Programme unter der verbreiteten Version Java 8. Auch wenn die aktuellen Sprachänderungen zu einer Programmverkürzung führen, ist es unrealistisch, anzunehmen, dass jedes Unternehmen sofort auf Java 11 wechselt – dafür bereitet das Modulsystem aus Java 9 noch zu viele Sorgen, zudem ist Java 11 nach Java 8 das erste Release mit Langzeitunterstützung. Es ist daher nur naheliegend, das Buch für eine breite Entwicklergemeinde auszulegen anstatt für ein paar wenige, die sofort mit der neuesten Version arbeiten können.

### Das Buch in der Lehre einsetzen

Die »Insel« eignet sich ideal zum Selbststudium. Das erste Kapitel dient dem 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, Compilieren und Laufenlassen eines Java-Programms sind 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 kommen dazu, spielen im Buch aber keine Rolle.

In der Softwareentwicklung ist die Dokumentation der API-Schnittstellen 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# objektorientiert programmiert hat, kann [Kapitel 3](#) überfliegen und dann einsteigen. Objektorientierter Mittelpunkt des Buchs sind [Kapitel 6](#) und [Kapitel 7](#): Sie vermitteln die OO-Begriffe Klasse, Methode, Assoziation, Vererbung, dynamisches Binden. Nach [Kapitel 7](#) ist die objektorientierte Grundausbildung abgeschlossen, und nach [Kapitel 14](#) sind die Grundlagen von Java bekannt. Es folgen Überblicke in einzelne Bereiche der Java-Standardbibliothek, die der zweite Band vertieft.

Mit diesem Buch und einer Entwicklungsumgebung des Vertrauens lassen sich die ersten Programme entwickeln. Um eine neue Programmiersprache zu erlernen, reicht das Lesen aber nicht aus. Mit den Übungsaufgaben kann jeder Leser deshalb die eigene Fingerfertigkeit trainieren. Da viele Lösungen unter [www.rheinwerk-verlag.de/4804](http://www.rheinwerk-verlag.de/4804) verfügbar sind, lassen sich die eigenen Ergebnisse 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 Büchereibuch ist, sollten Sie Passagen, die Ihnen wichtig erscheinen, mit vielen Farben hervorheben und mit Randbemerkungen versehen. Das Gleiche 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 Beispiele 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, über mein Wissen zu reflektieren, doch sollte dies jeder auf seine Weise tun. Allemal 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 ([ullenboom@gmail.com](mailto:ullenboom@gmail.com)) angebracht.

Die PQ4R-Methode erleichtert den Wissenserwerb. Allerdings ist es so wie mit allen gelerten Dingen: Wer Wissen nicht regelmäßig auffrischt, vergisst es. Der PQ4R-Methode sollte daher für einen langfristigen Erfolg ein zusätzliches »R« verpasst werden: »R« für »repeat« oder »refresh«. In der Regel überliest man etwas am Anfang oder schenkt Details kaum Bedeutung. Erst später, wenn mehr Hintergrundwissen vorhanden ist, kann auch Grundlegendes in neuem Licht erscheinen und sich gut zum Gesamtbild fügen. Leser sollten sich daher immer wieder die Insel vornehmen und querblättern.

### Persönliche Bemerkung

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 teilten mir mit, dass die Zeit der Homecomputer vorbei sei.<sup>2</sup> Mit Java war das anders, denn hier war ich zur richtigen Zeit am richtigen Ort.

---

<sup>2</sup> Damit verlor ich eine Wette gegen Georg und Thomas – 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 zur Magenleerung anhalten mussten. Jetzt ist das Buch online unter <http://retrobude.de/dokumente/amiga-assembler-buch>.

## Fokus auf das Wesentliche

Einige Abschnitte 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 JavaScript, C++, C# oder andere Programmiersprachen dienen aber nicht wesentlich dem Verständnis, sondern nur dem Vergleich.

### \*-Abschnitte

Das Java-Universum ist reich an Feinheiten und verwirrt Einsteiger oft. 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 ein JDK zu beziehen und zu installieren ist, damit die ersten Programme übersetzt und gestartet werden können. Mit einer Entwicklungsumgebung macht es mehr Spaß, daher gibt es auch dazu eine Einführung.

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 Schleiftypen 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, denn im Mittelpunkt stehen die grundlegenden Konzepte, wie das Schlüsselwort new, Referenzen, null-Referenz, Referenzvergleiche. Die Bibliothek ist so reichhaltig, dass allein mit den vordefinierten Klassen und diesem Basiswissen schon viele Programme entwickelt werden können. Speziell die bereitgestellten Datenstrukturen lassen sich vielfältig einsetzen.

Mehrere Datentypen lassen sich in einem Array zusammenfassen, das Konzept stellt Kapitel 4 vor. Arrays stecken auch hinter einigen Java-Konzepten wie variablen Argumentlisten und dem erweiterten for.

Essenziell für viele Probleme ist der gewandelte, in [Kapitel 5](#) vorgestellte, »Umgang mit Zeichenketten«. Die wichtigen Typen `Character` (Datentyp für einzelne Zeichen) und `String`, `StringBuilder` (Datentypen für Zeichenfolgen) werden eingeführt. 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 6](#) werden wir »Eigene Klassen schreiben«. Anhand von Spielen und Räumen modellieren wir Objekteigenschaften. Wichtige Konzepte – wie Konstruktoren, statische Eigenschaften, Aufzählungen – finden dort ihren Platz.

Objektbeziehungen, also Assoziationen zwischen Objekten, auch genannt Benutzt-Beziehungen sowie Vererbungsbeziehungen, zeigt [Kapitel 7](#), »Objektorientierte Beziehungsfragen«, auf. Das Kapitel führt in die »echte« objektorientierte Programmierung ein und erhellt dynamisches Binden, abstrakte Klassen und Schnittstellen (Interfaces) sowie Sichtbarkeit. Da Klassen in Java auch innerhalb anderer Klassen liegen können (geschachtelte Klassen), setzt sich ein eigener Abschnitt damit auseinander.

Exceptions, die wir in [Kapitel 8](#), »Ausnahmen müssen sein«, 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 9](#), »Geschachtelte.Typen«, beschreibt, wie sich Klassen ineinander verschachteln lassen. Das verbessert die Kapselung, denn auch Implementierungen können dann sehr lokal sein.

[Kapitel 10](#), »Besondere Typen 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 11](#), »Generics<T>«, gibt einen Einblick in die Technik.

Lambda-Ausdrücke gibt es seit Java 8; sie vereinfachen insbesondere das funktionale Programmieren. Wir widmen den Spracheigenschaften [Kapitel 12](#), »Lambda-Ausdrücke und funktionale Programmierung«, genau wie vielen Standardtypen, die bei funktionaler Programmierung auftauchen.

Danach sind die Fundamente gelegt, und die verbleibenden Kapitel dienen dazu, das bereits erworbene Wissen auszubauen. [Kapitel 13](#), »Architektur, Design und angewandte Objektorientierung«, zeigt Anwendungen guter objektorientierter Programmierung und stellt Entwurfsmuster (Design-Patterns) vor. An unterschiedlichen Beispielen demonstriert das Kapi-

tel, wie Schnittstellen und Klassenhierarchien gewinnbringend in Java eingesetzt werden. Es ist der Schlüssel dazu, nicht nur im Kleinen zu denken, sondern auch große Applikationen zu schreiben.

**Kapitel 14**, »Komponenten, JavaBeans und Module«, beantwortet die Frage, wie in Java einzelne Komponenten oder Systeme von Komponenten flexibel für andere Anwendungen eingesetzt werden können. Neu in Java 9 ist das Modulsystem.

Nach den ersten 14 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 Abschnitt »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 4.200 Klassen, Schnittstellen, Aufzählungen, Ausnahmen und Annotationen. **Kapitel 15**, »Die Klassenbibliothek«, (und auch der Anhang) 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.

**Kapitel 16** bis **Kapitel 21** geben einen Überblick über spezielle APIs. Die Bibliotheken sind sehr umfangreich, und das aufbauende Expertenbuch »Java SE 9 Standard-Bibliothek. Das Handbuch für Java-Entwickler« vertieft die APIs weiter. **Kapitel 16** gibt eine »Einführung in die nebenläufige Programmierung«. **Kapitel 17**, »Einführung in Datenstrukturen und Algorithmen«, zeigt praxisnah geläufige Datenstrukturen wie Listen, Mengen und Assoziativspeicher. Einen Kessel Buntes bietet **Kapitel 18**, »Einführung in grafische Oberflächen«, wo es um die Swing-Bibliothek geht. Im Kern steht eine bildgeführte Anleitung, wie sich mit einer Entwicklungsumgebung einfach grafische Oberflächen entwickeln lassen. Darüber, wie aus Dateien gelesen und geschrieben wird, gibt **Kapitel 19**, »Einführung in Dateien und Datenströme«, einen Überblick. Wer zum Abspeichern von Daten eine relationale Datenbank nutzen möchte, der findet in **Kapitel 20**, »Einführung ins Datenbankmanagement mit JDBC«, Hilfestellungen. Alle genannten Kapitel geben aufgrund der Fülle nur einen Einblick, und ihre Themen werden im zweiten Band tiefer aufgefächert. Ein neues Thema spannt **Kapitel 21** mit einer »Einführung in <XML>« auf. Java als plattformunabhängige Programmiersprache und XML als dokumentenunabhängige Beschreibungssprache sind ein ideales Paar, und die Kombination dieser beiden Technologien ist der Renner der letzten Jahre. Das Kapitel beginnt auf der höchsten Abstraktionsebene, beschreibt, wie XML-Daten auf Objekte übertragen werden, und geht dann weiter zur elementaren Verarbeitung der XML-Dokumente, wie sie zum Beispiel als Objektbaum aufgebaut und modifiziert werden.

**Kapitel 22** stellt »Bits, Bytes, Mathematisches und Geld« 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.

Ein automatischer Test von Anwendungen hilft, Fehler in Code aufzuspüren, insbesondere wenn später dumme Codeänderungen gemacht wurden und sich das Programm nicht mehr konform mit der Spezifikation verhält. [Kapitel 23](#), »Testen mit JUnit«, stellt die Idee der Testfälle vor und wie diese einfach und automatisch ausführbar werden. Zudem setzt das Kapitel den Schwerpunkt auf das Design von testbaren Anwendungen.

Abschließend liefert [Kapitel 24](#), »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.

Der Anhang, »Java SE-Module und Paketübersicht«, erklärt alle Java-Module und -Pakete kurz mit einem Satz und erläutert 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 *Dateinamen*, *HTTP-Adressen*, *Namen ausführbarer Programme*, *Programmoptionen* und *Dateiendungen (.txt)* *kursiv*. Einige Links führen nicht direkt zur Ressource, sondern werden über <http://tutego.de/go> zur tatsächlichen Quelle umgeleitet, was nachträgliche URL-Änderungen erleichtert.
- ▶ Begriffe der Benutzeroberfläche stehen in **KAPITÄLCHEN**.
- ▶ Listings, Methoden und sonstige Programmelemente sind in *nichtproportionaler Schrift* gesetzt. An einigen Stellen wurde hinter einer Listingzeile ein abgeknickter Pfeil (↗) als Sonderzeichen gesetzt, der den Zeilenumbruch markiert. Der Code aus der nächsten Zeile gehört also noch zur vorigen.
- ▶ Bei Methodennamen und Konstruktoren folgt immer ein Klammerpaar, um Methoden/Konstruktoren von Attributen abgrenzen zu können. So ist bei der Schreibweise `System.out` und `System.gc()` klar, dass Ersteres ein Attribut ist und Letzteres eine Methode.
- ▶ Hat eine Methode/ein Konstruktor einen Parameter, so steht der Typ in Klammern. Beispiele: `Math.abs(double a)`, `Point(int x, int y)`. Oft wird auch der Parametername ausgelassen. Beispiel: »`System.exit(int)` beendet das Programm mit einem Rückgabewert.« Die Java-API-Dokumentation macht das genauso. Hat eine Methode/ein Konstruktor eine Parameterliste, ist diese aber im Text gerade nicht relevant, wird sie mit »...« abgekürzt. Beispiel: »Die Methode `System.out.print(...)` konvertiert die übergebenen Argumente in Strings und gibt sie aus.« Ein leeres Klammerpaar bedeutet demnach, dass eine Methode/



ein Konstruktor wirklich keine Parameterliste hat. Um die Angabe kurz und dennoch präzise zu machen, gibt es im Text teilweise Angaben wie `getProperty(String key[, String def])`, was eine Abkürzung für `getProperty(String key)` und `getProperty(String key, String def)` ist.

- ▶ Um eine Gruppe von Methoden anzugeben, symbolisiert die Kennung **XXX** einen Platzhalter. So steht zum Beispiel `printXXX(...)` für die Methoden `println(...)`, `print(...)` und `printf(...)`. Aus dem Kontext geht hervor, welche Methoden gemeint sind.
- ▶ Lange Paketnamen werden teilweise abgekürzt, sodass `com.tutego.insel.game` etwa zu `c.t.i.g` wird.
- ▶ 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:

```
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. im Kommandozeilencompiler `javac`. Aber natürlich führen beide Compiler zu ähnlichen Fehlermeldungen.
- ▶ 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. Sun 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** 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** 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 im Internet verfügbar (<http://www.rheinwerk-verlag.de/4804>). Wird ein Ausschnitt einer Datei *Person.java* abgebildet, steht »Ausschnitt« oder »Teil 1«, »Teil 2« ... dabei:

#### **Listing 1** Person.java, Ausschnitt

#### **Listing 2** Listing 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, *DateFormatDemo*.

#### **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*) bzw. der Name vom Annotations- oder Aufzählungstyp. 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 an-

deres 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 Programmierersprache in diesem Buch ist Englisch, um ein Vorbild für »echte« Programme zu sein. Bezeichner wie Klassennamen, Methodennamen und auch eigene API-Dokumentationen sind auf Englisch, um eine Homogenität mit der englischsprachigen 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

Ausführliche Informationen zu diesem Buch liefert das Internet unter den Adressen <http://tutego.de/javabuch> und <http://www.rheinwerk-verlag.de/4804>. 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 oben genannten Buch-Webseite erhältlich. 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://tutego.de/javabuch/aufgaben>. 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 Webs 2.0. Neue Kapitel und Abschnitte des Buchs werden immer im Java-Insel-Blog <http://tutego.de/blog/javainsel> veröffentlicht. So kann jeder Leser – mehrere Monate bevor die Texte überhaupt zum Lektorat und schlussendlich ins Buch wandern – seine Meinung äußern und Kritik üben. Es gibt weltweit nur wenige Autoren, die so konsequent alle Texte ihres Buches vorveröffentlichen und bereit sind, sie in aller Öffentlichkeit zu diskutieren. Der Vorteil für die Leser zahlt sich aus: In den über zehn Insel-Jahren liegt die Anzahl fachlicher Fehler bei fast null.



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

Neben den reinen Buchaktualisierungen publiziert der Blog auch tagesaktuelle Nachrichten über die Java-Welt und Java-Tools. Google zählt den Insel-Blog weltweit zu den Top-Java-Blogs und platziert ihn seit Jahren auf Platz 1.

Facebook-Nutzer können ein Fan der Insel werden (<http://tutego.de/go/fb>), 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://tutego.de/seminare> werfen. tutego bietet über dreihundert IT-Seminare zu Java-Themen, C(++), C#/.NET, Softwaredesign, 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, Eclipse
- ▶ JavaServer Faces (JSF), JavaServer Pages (JSP), Servlets und weitere Webtechnologien
- ▶ Spring Framework, Spring Boot
- ▶ Java EE, EJB, OR-Mapping mit JPA, Hibernate
- ▶ grafische Oberflächen mit Swing und JFC, JavaFX, Eclipse RPC, SWT
- ▶ Java und XML, JAXB
- ▶ Android

- ▶ Build-Management mit Maven, Hudson/Jenkins, Ant
- ▶ .NET, C#, iOS-Programmierung, HTML5/CSS3-Frameworks

## Danksagungen

Ich war 9 Jahre alt, als mir meine Eltern mit dem C64 den Weg in die Welt der Computer öffneten. Ein Medium zum Speichern der Programme und Daten fehlte mir, und so blieb mir nichts anderes übrig, als nach dem Neustart alles wieder neu einzugeben – auf diesem Weg lernte ich das Programmieren. An meine Eltern einen großen Dank für den Computer, für die Liebe und das Vertrauen.

Dank gebührt Sun Microsystems, die 1991 mit der Entwicklung an Java begannen. Ohne Sun gäbe es kein Java, und ohne Java gäbe es auch nicht dieses 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. Ein weiteres Dankeschön geht an verschiedene treue Leser, deren Namen aufzulisten viel Platz kosten würde; ihnen ist die Webseite <http://tutego.de/javabuch/korrekteure.html> 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, -Bibliotheken und -Frameworks wie Eclipse, Maven, JUnit, Spring, Tomcat, WildFly schreiben und warten: Ohne sie wäre Java heute nicht da, wo es ist.

Abschließend möchte ich dem Rheinwerk Verlag 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, sind bei über 1.200 Seiten Unstimmigkeiten wahrscheinlich, so wie auch jede Software rein statistisch Fehler aufweist. 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 [ullenboom@gmail.com](mailto:ullenboom@gmail.com) zu senden. Ich bin für Anregung, Lob und Tadel stets empfänglich.

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

Dortmund im Jahr 2019, Jahr 9 nach Oracles Übernahme<sup>3</sup>

**Christian Ullenboom**

---

<sup>3</sup> Das war am 17. Januar 2010 zum Preis von 7,4 Milliarden US\$.

## Vorwort zur 14. Auflage

Diese Ausgabe ist insofern neu, als sie gleich zwei Java-Versionen auf einmal abdeckt. Bisher gab es für jede neue Java-Version entweder ein oder sogar zwei Insel-Auflagen, doch das ändert sich ab Java 10, denn es gibt zwei Releases im Jahr. So schnell kommen wir mit der Insel nicht nach ... In der Zukunft umfasst jede neue Auflage also höchstwahrscheinlich zwei Java-Releases, für die jetzige Auflage sind es Java 10 und Java 11.

Auch von Oracle selbst gibt es Neues, denn Oracle möchte mit seiner JDK-Implementierung Geld verdienen, und so ist das Oracle JDK nur noch für Testzwecke und Entwicklungen frei und sonst kostenpflichtig. Diese Auflage beschreibt daher nicht mehr das Oracle JDK, sondern das freie und komplett quelloffene OpenJDK. Die Unterschiede sind unbedeutend, daher macht der Wechsel keinen Unterschied. Unter Linux war es sowieso immer die meistverbreitete Implementierung.

Sprachlich wurden einige Ausdrücke präzisiert. An einigen Stellen hieß es früher Zeichensatz, wenn aber Schriftart gemeint war. Und statt Kontrollzeichen heißt es nun Steuerzeichen. Bei einem try-catch handelt es sich nicht um eine catch-Anweisung, sondern um eine catch-Klausel. Innere Klassen sind laut JLS nur nichtstatische Klassen, das wurde korrigiert. Es heißt nun allgemein geschachtelte Klassen.

Kapitel 3 enthielt bisher immer die Einführung in die Objektorientierung und auch das Konzept der Arrays. Diese unterschiedlichen Themen sind nunmehr auf zwei Kapitel aufgeteilt. Das Generics-Kapitel nutzte Pocket als Beispielklasse, doch Taschen sind langweilig, Raketen sind cooler, daher heißt es nun Rocket statt Pocket. Zudem habe ich die Nutzung der Begriffe Typparameter und Typargument vereinheitlicht.

Alle Programme der Insel sind im Internet frei verfügbar und als Eclipse-Workspace verpackt. Bisher ist jedes Kapitel ein eigenes Eclipse-Java-Projekt. Da die Paketnamen aber sowieso eindeutig waren und sich Band 1 und Band 2 auch Programme teilen, gibt es für die Beispiele nur noch ein Java-Projekt im Workspace und ein Java-Projekt für die Lösungen der Übungsaufgaben.

## Vorwort zur 13. Auflage

Das Bibliothekskapitel beschreibt alle Module. Das Compact Profil, das erst in Java 8 eingezogen ist, ist wieder rausgeflogen. Der Abschnitt über Annotationen und Generics aus Kapitel 3 ist in Kapitel 6 verschoben worden, dort sind die Themen besser im Kontext verknüpft.

Bei den fortgeschrittenen Kapiteln liegt die größere Änderung in der Einführung von Maven; die Java-Insel-Eclipse-Projekte wurden in Maven-Projekte konvertiert. Das heißt auch, dass Java-Archive nicht mit Teil der Projekte sind, sondern Eclipse sie beim Start erst vom zentralen Maven-Server beziehen muss.

Nach dem Vorwort ist es jetzt an der Zeit, das Leben zwischen den geschweiften Klammern zu genießen 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 Geräte führen Java-Programme aus,<sup>1</sup> 1,1 Milliarden Desktops und alle Blu-ray-Player.<sup>2</sup> 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. Java<sup>3</sup> hätte einfach nur eine schöne Insel, eine reizvolle Wandfarbe oder eine Pinte mit brasiliianischen Rhythmen in Paris sein können, so wie Heuschrecken einfach nur grüne Hüpfer 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, als die Hippies noch zu Jimi Hendrix tanzten, wollte Bill Joy eine Programmiersprache schaffen, die alle Vorteile von *MESA*<sup>4</sup> und *C* vereinigen sollte. Diesen Wunsch konnte sich Joy, Mitbegründer von Sun Microsystems, 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.

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

---

1 Zahlen von Oracle: <http://emeapressoffice.oracle.com/Press-Releases/Oracle-Outlines-Plans-to-Make-the-Future-Java-During-JavaOne-2012-Strategy-Keynote-31b0.aspx>

2 So verkündet es Thomas Kurian auf der JavaOne-Konferenz 2010 (<http://www.oracle.com/us/corporate/press/193190>). Die Zahlen sind relativ konstant geblieben.

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

4 Die Programmiersprache MESA wurde in den 1970er Jahren am Forschungszentrum Xerox Palo Alto Research Center (Xerox PARC) entwickelt. Damit wurde der Xerox Alto programmiert, der erste Computer mit kompletter grafischer Oberfläche. Die Syntax von MESA weist Ähnlichkeiten mit Algol und Pascal auf.

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.<sup>5</sup>

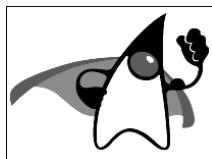


Abbildung 1.1 Der Duke, Symbol für Java

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, die etwa an einen Fernseher angegeschlossen werden). 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). Die Programmiersprache sollte Programmcode über das Netzwerk empfangen können, und fehlerhafte Programme sollten keinen Schaden anrichten. 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 1994 den Namen *Java*, da der Name Oak von Oak Technology schon angemeldet war. Nach der Überlieferung fiel die Entscheidung für den Namen Java in einem Coffeeshop. In Java führte Patrick Naughton den Prototyp des Brow-

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

sers *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. Am 23. Januar 1996 wurde das JDK 1.0 freigegeben, das 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 Wertpapierbörsen 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 seitens 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, OpenOffice, 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, vermied es aber, Dinge aufzunehmen, die nur von sehr wenigen Entwicklern eingesetzt werden oder die öfter zu Fehlern führen. In den Anfängen stand C++ als Vorbild da, heute schielt 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 einen bestimmten Prozessor (zum Beispiel für x86er-Mikroprozessoren oder Prozessoren der ARM-Architektur) und eine spezielle Plattform (etwa Linux oder Windows) generieren, erzeugt der Java-Compiler aus den Quellcode-Dateien den so genannten *Bytecode*. Dieser Programmcode ist binär und Ausgangspunkt für die virtuelle Maschine zur Ausführung. Bytecode ist vergleichbar mit Mikroprozessorcode für einen erdachten Prozessor, der Anweisungen wie arithmetische Operationen, Sprünge und Weiteres kennt.

## 1.2.2 Ausführung des Bytecodes durch eine virtuelle Maschine

Damit der Programmcode des virtuellen Prozessors ausgeführt werden kann, kümmert sich nach der Übersetzungsphase von Programmcode in Bytecode eine *Laufzeitumgebung*, genannt die *Java Virtual Machine (JVM)*, um den Bytecode.<sup>6</sup> Die Laufzeitumgebung (auch *Runtime-Interpreter* genannt) lädt den Bytecode, prüft ihn und führt ihn in einer kontrollierten Umgebung aus. Die JVM bietet eine ganze Reihe von Zusatzdiensten wie eine automatische Speicherbereinigung (Garbage-Collector), die Speicher aufräumt, sowie eine starke Typprüfung unter einem klar definierten Speicher- und Threading-Modell.



<sup>6</sup> 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 Zwischenencode – kurz *p-code*.

### 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 zehn Jahren kamen mehr und mehr Programmiersprachen hinzu. Neben Java sind plattformunabhängige Programmiersprachen und Laufzeitumgebungen .NET-Sprachen wie C# auf der CLR (*Common Language Runtime* – entspricht der Java VM), Perl, Python oder Ruby. Plattformunabhängigkeit ist schwer, denn die Programmiersprache und ein Bytecode produzierender Compiler sind nur ein Teil – der größere Teil sind 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 änderte sich die Richtung etwas, was sich besonders an der 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.<sup>7</sup> Mit Java 7 gab es einen Wechsel: Nun sind plattformspezifische Dateieigenschaften zugänglich. Es bleibt abzuwarten, ob in 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-lang.org>), *JRuby* (<http://jruby.org>), *Jython* (<http://jython.org>)

---

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

[www.jython.org](http://www.jython.org)), [Kotlin](http://kotlinlang.org/) (<http://kotlinlang.org/>) oder [Scala](http://www.scala-lang.org) (<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 Datenstrukturen, Zeichenkettenverarbeitung, Datum-/Zeit-Verarbeitung, grafische Oberflächen, Ein-/Ausgabe, Netzwerkoperationen und mehr. Das bildet die Basis für höherwertige Dienste wie Datenbankanbindungen oder Webservices. 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-Prozessoren sorgt das Betriebssystem für eine optimale Ausnutzung der Rechenleistung, da Threads wirklich nebenläufig arbeiten können.

## 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 Softwareproblems 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 existieren für numerische Zahlen oder Unicode-Zeichen und 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

waren, 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 Ausbildungswägen, woher bekommt ein Team einen Entwickler oder Consultant, wenn es brennt? Dies sind nicht unbedingt Punkte, die Informatiker beim Sprachvergleich an die erste Stelle setzen, sie sind aber letztendlich für den Erfolg einer Softwareplattform 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 Supersprache 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.

Heute ist Java die Basis sehr erfolgreicher Produkte, viele auf der Serverseite. Dazu zählen Facebook, LinkedIn, Twitter, Amazon, eBay. Auf der Client-Seite ist Java seltener zu finden, das Spiel Minecraft ist eher eine Ausnahme.

#### Java-Entwickler sind glücklich

Andrew Vos hat sich Kommentare angeschaut, mit denen Entwickler ihre Programme in die Versionsverwaltung einpflegen.<sup>8</sup> Dabei zählte 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 Sprache 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 ein-

---

<sup>8</sup> <http://tutego.de/go/profanity>

mal 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.

Die Technik der *Just-in-time-(JIT-)Compiler*<sup>9</sup> war der 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. Eine Übersetzung zur Laufzeit hat enormes Optimierungspotential, da die JVM natürlich weiß, was für eine CPU eingesetzt ist, und für die CPU bestmöglichen Maschinencode erzeugen kann. Ein Beispiel ist die Befehlssatzerweiterung SSE2 (Streaming SIMD Extensions 2) für x86-Prozessoren – findet die JVM diesen Prozessortyp vor, kann SSE2-Code erzeugt werden, andernfalls eben nicht.

Nach der Übersetzung 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 Compilermodus, 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. Und das Ergebnis ist sehr gut, wie sich Java im Vergleich zu anderen Sprachen schlägt.<sup>10</sup>

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.<sup>11</sup> 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.

---

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

<sup>10</sup> <https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fast.html>

<sup>11</sup> Dynamische Methodenaufrufe sind in der Regel sehr schnell, weil die JVM die Typhierarchie kennt und sehr aggressiv optimieren kann, aber Optimierungen auch wieder zurücknehmen kann, wenn sich die Typhierarchie etwa durch nachgeladene Klassen ändert. Auch weiß die JVM, welche Programmteile nebenläufig ausgeführt werden und gesichert werden müssen, oder ob die Synchronisation entfallen kann.

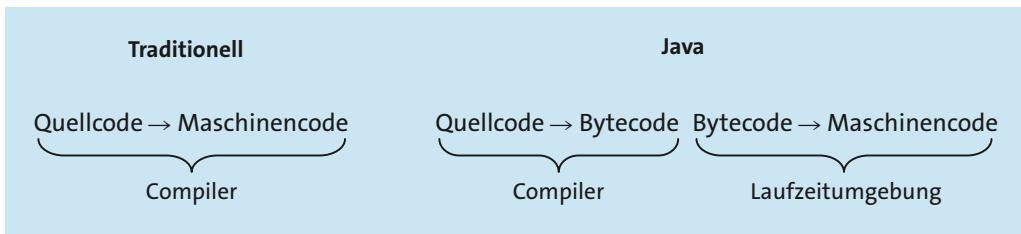


Abbildung 1.2 Traditioneller Compiler und Java-Compiler mit Laufzeitumgebung

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 einfach ist. Die HotSpot-VM hat eine eigene Entwicklung und Versionsnummer.

### 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 sie 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.

Java kämpfte Anfang 2013 mit einigen bösen Sicherheitsproblemen. Applets, also Java-Programme, die in Webseiten eingebettet sind, konnten sich Zugriff zum System verschaffen, Anwendungen starten und so Blödsinn anstellen. Die Debatte kochte so hoch, dass auch Massenmedien<sup>12</sup> die Meldung aufgriffen und zur Deinstallation von Java rieten. Zwei Dinge müssen jedoch berücksichtigt werden: Erstens betraf das Problem nur Applets, die fast voll-

<sup>12</sup> Ja, selbst in der BILD-Zeitung ist es angekommen: <http://www.bild.de/digital/internet/datenschutz/java-sicherheitsluecke-25899768.bild.html>.

ständig aus dem Netz verschwunden sind (bis auf ELSTER<sup>13</sup>, das uns immer noch zu Java zwingt). Auf der Serverseite, wo Java in der Regel zu Hause ist, gefährden diese Angriffe die Sicherheit nicht. Zweitens – allerdings ist das kein wirklicher Trost<sup>14</sup> – kann es durch Programmierfehler der Sandbox immer wieder zu Ausbrüchen aus der Umgebung kommen. Vergleichbare Meldungen vom Acrobat Reader, der einbettete Skripte ausführt, sind nicht selten. Da ist es gut, dass Java eine so hohe Verbreitung hat, dass Fehler schnell gefunden und gefixt werden. Leider machte Oracle hier aber keine sehr gute Figur, und es dauerte eine paar Tage, bis der gefährliche Zero-Day-Exploit<sup>15</sup> gefixt wurde. Selbst Facebook-Mitarbeiter waren von der Java-Sicherheitslücke betroffen.<sup>16</sup>

### 1.2.9 Zeiger und Referenzen

In Java gibt es keine Zeiger (engl. *pointer*) auf Speicherbereiche, wie sie aus anderen Programmiersprachen bekannt und gefürchtet sind. Da eine objektorientierte Programmiersprache ohne Verweise aber nicht funktioniert, führt Java *Referenzen* ein. Eine Referenz repräsentiert ein Objekt, und eine Variable speichert diese Referenz – sie wird *Referenzvariable* genannt. Während Programmierer nur mit Referenzen arbeiten, verbindet die JVM die Referenz mit einem Speicherbereich; der Zugriff, *Dereferenzierung* genannt, ist indirekt. Referenz und Speicherblock sind also getrennt; das ist sehr flexibel, da Java das Objekt im Speicher bewegen kann.

Das im Speicher aufgebaute Objekt hat einen Typ, der sich nicht ändern kann; wir sprechen vom *Objekttyp*: Ein Auto bleibt ein Auto und ist kein Laminiersystem. Eine Referenz unter Java kann verschiedene Typen annehmen, wir nennen das *Referenztyp*: Ein Java-Programm kann ein Auto auch als Fortbewegungsmittel ansehen.



#### Beispiel \*

Das folgende Programm zeigt, dass das Pfuschen in C++ leicht möglich ist und wir über eine Zeigerarithmetik Zugriff auf private Elemente bekommen können.<sup>17</sup> Für uns Java-Programmierer ist dies ein abschreckendes Beispiel.

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

<sup>13</sup> Siehe <http://www.elster.de>, das es ermöglicht, »verschiedene Steuererklärungen elektronisch via Internet an das Finanzamt zu übermitteln«.

<sup>14</sup> Und die Frage ist auch, was peinlicher ist: gegen eine gut gemachte Attacke schwach zu werden oder bei der Eingabe von »File:///« im Editor mit Rechtschreibkorrektur abzustürzen.

<sup>15</sup> Ein Fehler, der nach dem Entdecken sofort ausgenutzt wird, um Schadsoftware in Systeme einzuschleusen

<sup>16</sup> <http://www.facebook.com/notes/facebook-security/protecting-people-on-facebook/10151249208250766>

<sup>17</sup> Auch ohne Compiler lässt sich das online prima testen: <http://ideone.com>.

```

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.<sup>18</sup> Der Compiler bricht mit einer Fehlermeldung ab – bzw. das Laufzeitsystem löst eine Ausnahme (Exception) aus –, wenn das Programm einen Zugriff auf eine private Variable versucht.

### 1.2.10 Bring den Müll raus, Garbage-Collector!

In Programmiersprachen wie C(++) lässt sich etwa die Hälfte der Fehler auf falsche Speicherallokation zurückführen. Mit Objekten und Strukturen 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, die *automatische Speicherfreigabe* von Java (engl. *garbage collector*, kurz GC) entfernt sie. Nach dem expliziten Aufbauen eines Objekts überwacht das Laufzeitsystem von Java permanent, ob das Objekt noch gebraucht, 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 so genannten *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 Funk-

---

<sup>18</sup> Ganz stimmt das allerdings nicht. Mit Reflection lässt sich da schon etwas machen, wenn die Sicherheits-einstellungen das nicht verhindern.

tionen `free(...)` aus C oder `delete(...)` aus C++ überflüssig. Wir können uns über diese Technik freuen, da damit viele Probleme 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(++) gibt.<sup>19</sup>

Eine automatische Speicherbereinigung ist nicht ganz unproblematisch, da sie immer dann zuschlagen kann, wenn das System gerade etwas ganz Zeitkritisches machen möchte und Unterbrechungen nicht gelegen kommen. Doch die modernen Garbage-Collectors sind gut darin, wenig aktive Zyklen zu erkennen und die Arbeit gleichmäßig und auch auf mehrere Prozessorkerne zu verteilen. Doch automatische Speicherbereinigung ist nichts Neues, und die Verfahren sind lange erprobt. Schon frühere Programmiersprachen wie LISP (1958) und Smalltalk (1972) brachten einen Garbage-Collector mit, und alle modernen Programmiersprachen (bzw. deren Laufzeitumgebungen) besitzen eine automatische Speicherbereinigung.

### 1.2.11 Ausnahmebehandlung

Java unterstützt ein modernes System, mit Laufzeitfehlern umzugehen. In die Programmiersprache wurden *Ausnahmen* (engl. *exceptions*) eingeführt: Fehlerobjekte, 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(<sup>20</sup>) die Array-Grenzen (engl. *range checking*) standardmäßig nicht, was ein Grund für viele Sicherheitsprobleme ist. Ein fehlerhafter Zugriff auf das Element  $n + 1$  eines Arrays 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; Array-Überprüfungen kosten daher nicht mehr die Welt und machen sich nicht automatisch in einer schlechteren Performance bemerkbar.

---

<sup>19</sup> Ein bekannter Garbage-Collector stammt von Hans-J. Boehm, Alan J. Demers und Mark Weiser. 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 wurden die Funktionen `malloc(...)` und `free(...)` einfach durch neue Methoden ausgetauscht.

<sup>20</sup> In C++ ließe sich eine Variante mit einem überladenen Operator lösen.

### 1.2.12 Angebot an Bibliotheken und Werkzeugen

Java ist mittlerweile schon so lange im Einsatz, dass es eine große Anzahl von Werkzeugen gibt, angefangen bei den Entwicklungsumgebungen mit unterstützenden Editoren über gute Debugger bis hin zu Werkzeugen zum Build-Management. Neben den Standardbibliotheken kommen 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.

Neue Programmiersprachen haben es schwer, da mitzuhalten. Die Anzahl der Programmiersprachen ist in den letzten Jahren explodiert, doch so richtig können sie die breite Masse nicht anziehen, da eben Werkzeuge und Bibliotheken fehlen. Interessanter sind da schon die neuen Sprachen auf der Basis der JVM, denn das erlaubt weiterhin die Nutzung der Werkzeuge und Bibliotheken mit einer neuen Programmiersprache.

Werkzeuge und Bibliotheken sind es aber auch manchmal, die Teams vom Wechsel auf Java abhalten. Das trifft besonders die Spieleentwickler, die viel Erfahrung in C(++) haben und jahrelang Energie in Spiele-Frameworks gesteckt haben – dort sind die Tools einfach vorhanden, in Java (noch) nicht.

### 1.2.13 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, verhinderte 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 – je kompakter der Code, desto länger benötigt die menschliche Dekodierung. Auch wenn das folgende Perl-Beispiel aus »The Fifth Obfuscated Perl Contest Results«<sup>21</sup> 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";
#.: ::||-| ..|-| ::||-| ::||-| ..|-: ::||-| ..|
```

<sup>21</sup> [https://de.wikipedia.org/wiki/Obfuscated\\_Perl\\_Contest](https://de.wikipedia.org/wiki/Obfuscated_Perl_Contest)

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 auch hätte einfacher sein können (Stichwort *Generics*). Dennoch 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 == bzw. != 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 – ein matrix1.add(matrix2) ist mit längerem Methodennamen 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 es, denn es ist eine Sprachbeschränkung und keine Beschränkung der virtuellen Maschine.

### Neue Entwicklungen

Auch Java geht mit der Zeit und verändert sich. Das führt dazu, dass die Syntax erweitert wird und Entwickler Neues lernen müssen. Bisher gibt es keine überladenen Operatoren, aber vielleicht gibt es sie in zehn Jahren? Und da die Anforderung an statische Typsicherheit steigt, genauso wie der Wunsch, im Bereich funktionaler Programmierung aufzuholen, ist einiges passiert. Java war schon immer ein bisschen »geschwätzig«, doch die Symboldichte nahm ab Java 5 immer mehr zu und erreichte ihren (vorläufigen) Höhepunkt in Java 8 – in Java 9 kommen keine neuen Symbole hinzu.

### 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 Compilierung 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.<sup>22</sup>

#### 1.2.14 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 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 bekantere 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*<sup>23</sup> unter der Open-Source-Lizenz GPL 2 an. Dem war Ende 2006 die Freigabe des Compilers und der virtuellen Maschine vorausgegangen.

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

23 <http://openjdk.java.net>

Ein paar Statistiken zeigt <https://www.openhub.net/p/openjdk> an:

- ▶ rund 9 Millionen Zeilen Code insgesamt
- ▶ >50.000 Commits in die Versionsverwaltung insgesamt seit dem Bestehen vom OpenJDK
- ▶ 60 % vom OpenJDK ist Java-Code, 20 % C und C++.
- ▶ Es stecken nach dem COCOMO-Model (Constructive Cost Model) über 2.700 Jahre Entwicklungsarbeit in dem Code.

Im Prinzip kann mit dem OpenJDK jeder Entwickler sein eigenes Java zusammenstellen und beliebige Erweiterungen veröffentlichen. Mit der Lizenzform GPL kann Java auf Linux-Distributionen Platz finden, die Java vorher aus Lizenzgründen nicht integrieren wollten.

## 1.2.15 Wofür sich Java weniger eignet

Java wurde als Programmiersprache für allgemeine Probleme entworfen 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 wurde plattformunabhängig entworfen, sodass alle Methoden auf allen Systemen lauffähig sein sollten. 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:

- ▶ Bildschirm auf der Textkonsole löschen, Cursor positionieren und Farben setzen
- ▶ auf niedrige Netzwerkprotokolle wie ICMP zugreifen
- ▶ Microsoft Office fernsteuern
- ▶ Zugriff auf USB<sup>24</sup> 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.

---

<sup>24</sup> 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.

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. Die Standard-JVM ist in C++ geschrieben und wird vom GCC-Compiler bzw. Microsoft Visual Studio und XCode übersetzt.

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.

## 1.3 Java im Vergleich zu anderen Sprachen \*

Beschäftigen sich Entwickler mit dem Design neuer Programmiersprachen – oder Spracherweiterungen –, werden häufig andere Programmiersprachenkonstrukte auf ihre Tauglichkeit hin überprüft und dann bei Erfolg in das Konzept aufgenommen. Auch Java ist eine sich entwickelnde Sprache, die Merkmale anderer Sprachen aufweist.

### 1.3.1 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.

Das Klassenkonzept – und damit der objektorientierte Ansatz – wurde nicht unweentlich 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 alles zu einem großen Binärklumpen verbindet, ist in Java jeder Typ eine eigene Klassendatei. Alle Klassen – optional auch von einem anderen Rechner über das Netzwerk – lädt die JVM zur Laufzeit. Selbst Methodenaufrufe sind über das Netz möglich.<sup>25</sup> 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.

---

<sup>25</sup> Diese Möglichkeit ist unter dem Namen RMI (*Remote Method Invocation*) bekannt. Bestimmte Objekte können über das Netz miteinander kommunizieren.

### 1.3.2 Java und JavaScript

Obacht ist beim Gebrauch des Namens »Java« geboten. Nicht alles, was Java im Wortstamm hat, hat tatsächlich mit Java zu tun: JavaScript hat keinen großen Bezug zu Java – bis auf Ähnlichkeiten bei den imperativen Konzepten. Die Programmiersprache JavaScript wurde 1995 vom Netscape-Entwickler Brendan Eich entwickelt. 1997 hat die *European Computer Manufacturers Association* Teile von JavaScript im Standard ECMA-262 festgeschrieben und die Programmiersprache *ECMAScript* genannt. Die aktuelle Version ist ECMAScript 2018 (ES2018). Die bekannten Browserhersteller Google (Chrome), Microsoft (Edge) und Mozilla Foundation (Firefox) implementieren ECMAScript, fügen aber in der Regel Erweiterungen hinzu.<sup>26</sup>

Java und ECMAScript/JavaScript unterscheiden sich in vielen Punkten, genauso wie Schlüssel und Schüsselbein wenig miteinander zu tun haben. Die Klassennutzung ist mit einem Prototyp-Ansatz in JavaScript völlig anders als in Java, und JavaScript lässt sich ebenso zu den funktionalen Programmiersprachen zählen, was Java nun wahrlich nicht ist.

### 1.3.3 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 Kits*) und mit der *Microsoft Java Virtual Machine* (MSJVM) eine eigene schnelle Laufzeitumgebung. Das Problem war nur, dass Dinge wie RMI und JNI absichtlich fehlten<sup>27</sup> – JNI wurde 1998 nachgereicht. Entgegen allen Standards führte der J++-Compiler neue Schlüsselwörter wie `multicast` und `delegate` ein. Außerdem 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 multimedialfähig gemacht werden. Das führte natürlich zu dem Problem, dass Applikationen, die 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 bzw. die

---

<sup>26</sup> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/New\\_in\\_JavaScript/ECMAScript\\_Next\\_support\\_in\\_Mozilla](https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_Next_support_in_Mozilla)

<sup>27</sup> <http://www.microsoft.com/presspass/legal/charles.mspx>

*Java Foundation Classes*, man müsse sie »bei jeder sich bietenden Gelegenheit anpassen« (»pissing on at every opportunity«).<sup>28</sup>

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 ausführte, 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. Bedauerlicherweise wird das Projekt nicht weiterentwickelt.

Microsoft hat sich lange Zeit aus der Java-Entwicklung nahezu vollständig zurückgezogen. Es waren eher überschaubare Projekte wie der *Microsoft JDBC Driver for SQL Server*. Das Verhältnis ist heute auch deutlich entspannter, und Microsoft geht wieder einen Schritt auf Java zu. Zu erkennen ist das am Beitritt in die Jakarta EE Working Group und an der Unterstützung von Java-Applikationen in der Windows Cloud Azure.<sup>29</sup> Vielleicht gratuliert Microsoft irgendwann einmal Oracle, wie es auch Linux zum 20. Geburtstag gratuliert hat.<sup>30</sup>

#### 1.3.4 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 waren, könnte leicht angenommen werden, dass Java Pate für die Programmiersprache C# stand.<sup>31</sup> 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. Bei den Lambda-Ergänzungen von Java 8 ist ausdrücklich die Übernahme der C#-Syntax hervorgehoben,<sup>32</sup> anders als bei der Microsoft-Dokumentation, die jegliche Ähnlichkeit zwischen C# und Java leugnet.

Während Oracle für Java eine Aufteilung in das *Java SE* für die »allgemeinen« Programme und das *Jakarta EE* (ehemals Java EE) als Erweiterung für die »großen« Enterprise-Systeme

---

<sup>28</sup> 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.

<sup>29</sup> <https://azure.microsoft.com/de-de/develop/java/>

<sup>30</sup> <http://www.youtube.com/watch?v=ZA2kqAIoZM>

<sup>31</sup> 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.

<sup>32</sup> <http://mail.openjdk.java.net/pipermail/lambd-dev/2011-September/003936.html>

vornimmt, fließt bei Microsoft alles in *ein* Framework, genannt .NET. Das 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.

Eine neue Richtung entsteht durch .NET Core, einer quelloffenen, an das »große« .NET-Framework angelehnten Alternative, die für Windows, macOS und Linux x64 bereitsteht. Alle Änderungen an der Codebasis sind unter <https://github.com/dotnet/core> sichtbar. Da sich bestehende .NET-Anwendungen nicht ohne Anpassungen migrieren lassen, hat .NET Core noch nicht so richtig an Fahrt aufgenommen.

Etwas zynisch lässt sich bemerken, dass Java vielleicht nur deshalb noch lebt, weil Microsoft mit .NET ausschließlich auf Windows gesetzt hat, die Welt aber etwas anderes wollte. Mit .NET Core hat Microsoft zu lange gewartet, weil es nicht andere Plattformen stärken wollte.

## 1.4 Weiterentwicklung und Verluste

### 1.4.1 Die Entwicklung von Java und seine Zukunftsaussichten

Vor 20 Jahren waren zwei große Pluspunkte die Einfachheit im Vergleich zum Vorgänger C++ und das Fehlen von »gefährlichen« syntaktischen Konstrukten. So schrieb einer der Sprachväter, James Gosling – der nach der Übernahme von Sun durch Oracle das Unternehmen verließ –, 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.«<sup>33</sup>*

Der Wunsch nach einer einfachen Sprache besteht bis heute, allerdings ist in den letzten 20 Jahren viel passiert und Java deutlich komplexer geworden. Bedeutende Sprachänderungen gab es in Java 5 (also etwa zehn Jahre nach der Einführung von Java) und in Java 8. Das Modulsystem in Java 9 bringt ebenfalls neue Herausforderungen.

Bei der Dreifaltigkeit der Java-Plattform – 1. Java als Programmiersprache, 2. den Standardbibliotheken und 3. 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 JavaScript, Groovy, Kotlin,

---

<sup>33</sup> 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>)

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 man C#- oder VB.NET-Klassen deklariert oder nutzt.

Als die Übernahme von Sun vor der Tür stand, zeigte Oracle sich sehr engagiert gegenüber den Sun-Technologien. Nach der Übernahme 2010 wandelte sich das Bild etwas, und Oracle hat eher für negative Schlagzeilen gesorgt, etwa als es die Unterstützung für OpenSolaris eingestellt hat, seitdem es MySQL zurückfahrt und als Gefahr für die eigene Datenbank sieht und weil es OpenOffice erst spät an Apache übergeben hat (als sich LibreOffice schon ver-selbstständigt hatte). Auch was die Informationspolitik und Unterstützung von Usergroups angeht, verhält sich Oracle ganz anders als Sun. 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. Anlässlich der Sicherheitslücken in Java machte das Unternehmen ebenfalls keine gute Figur, insgesamt dürfte auf einer Bewertung zu finden sein: »Oracle hat sich bemüht, den Anforderungen gerecht zu werden.«

Fast 10 Jahre nach der Übernahme gibt es neue radikale Änderungen. Erstmals entfernte Oracle Teile der Java SE-Bibliothek, wie CORBA-Unterstützung, JAXB, JAX-RS (Web-Services), Applets, Java Web Start und JavaFX, auch die JavaScript-Engine soll verschwinden. Damit ist die bisher heilige Abwärtskompatibilität aufgegeben – Java 11 kann nicht in jedem Fall ältere Java-Software ausführen. Ein weiter Schock ist die Kommerzialisierung. Oracle bringt mit dem Oracle JDK alle halbe Jahre ein neues Release heraus, das jedoch ab Java 11 nur für »development, testing, prototyping or demonstrating purposes« genutzt werden darf, also nicht mehr kommerziell; für eine kommerzielle Nutzung fallen Gebühren an.<sup>34</sup> Da das Oracle JDK allerdings nur eine Java SE-Implementierung von vielen ist, gibt es gute Alternativen, wie das OpenJDK.

#### 1.4.2 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.

---

<sup>34</sup> <http://www.oracle.com/technetwork/java/eol-135779.html>

*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 Website 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, wie es die Seite <https://jcp.org/en/jsr/overview> zeigt.

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 und OpenJDK realisiert – frühere Alternativen wie *Apache Harmony*<sup>35</sup> oder *GNU Classpath*<sup>36</sup> sind beendet. Das Oracle JDK (kurz JDK) basiert auf dem OpenJDK, und das OpenJDK steht unter der GPL-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.) Die Java SE 11 Platform wird im JSR 384 beschrieben,<sup>37</sup> der ein so genanntes *Umbrella-JSR* ist, weil er andere JSRs referenziert.

### 1.4.3 Applets

Es ist nicht untertrieben, dem Web eine Schlüsselposition bei der Verbreitung von Java zuzuschreiben. Populär wurde Java Mitte der 1990er Jahre durch *Applets* – Java-Programme, die ein Browser ausführt. Eine HTML-Datei referenzierte das Applet, es bekam auf der Webseite einen Platz zugewiesen, der Browser holte sich eigenständig die Klassendateien und Ressourcen über das Netz und führte sie in der JVM aus.

Im Laufe der Zeit ging die Bedeutung für Java-Applets immer weiter zurück, und das lag an der Sprache, die ebenfalls 1995 erschien: JavaScript. Java-Applets brachten erstmals Dynamik und bewegte Grafik in die bis dahin statischen Webseiten, doch als dann die Webstandards CSS (1996) und SVG auftauchten (2001), setzten immer mehr Webentwickler auf eine Kombination von JavaScript mit diesen Standards. Denn Java-Applets haben, genauso wie Flash oder Silverlight, alle ein Problem: Sie benötigen ein Browser-Plugin. Das macht sie unattraktiv für Unternehmen, da im Internet kein Kunde ausgeschlossen werden soll. Früher, als die Webstandards noch nicht so weit entwickelt waren, brachten Flash und Silverlight Dynamik auf die Webseite, doch heute sind aufwändige Webanwendungen mit JavaScript und HTML 5/CSS3 realisierbar. Auch Microsoft stoppte bei Silverlight 5 die Entwicklung und be-

---

<sup>35</sup> <http://harmony.apache.org>. Seit Ende 2011 »retired«, also »in Pension« und somit tot.

<sup>36</sup> <http://www.gnu.org/s/classpath>. Seit Anfang 2009 keine neue Version mehr. 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.

<sup>37</sup> <https://www.jcp.org/en/jsr/detail?id=384>

vorzugt nun Lösungen auf der Basis von JavaScript + HTML 5 + CSS3, insbesondere für mobile Endgeräte, da den Redmontern klar ist, dass es nie Silverlight auf dem iPhone oder Android geben wird.<sup>38</sup> Adobe selbst beginnt mit Konvertern von Flash nach HTML 5/CSS3/JavaScript und zeigt damit auch die Zukunft auf.

Es ist lange her, dass die frühen Browser Netscape und Internet Explorer eine JVM integrierten – irgendwann haben die Hersteller die Java-Laufzeitumgebung entfernt. Eine Zeit lang lieferte Oracle das *Java Plug-in* (in der Schreibweise) aus, um die eigene JVM in den Browser zu integrieren. Moderne Browser unterstützen das Java Plug-in jedoch nicht mehr. Wegen fehlender Unterstützung in den Browsern – und den guten Webstandards – markierte Oracle in Java 9 Applets als veraltet und entfernte sie komplett in Java 10. *Java Web Start* war eine Alternative ähnlich den Applets, mit der sich Programme aus dem Internet laden ließen, allerdings ist auch Java Web Start nicht mehr Teil von Java 11.

#### 1.4.4 JavaFX

*Rich Internet Applications* (RIA) sind grafisch anspruchsvolle Webanwendungen, die in der Regel Daten aus dem Internet beziehen. Viele Jahre beherrschte *Adobe Flash* hier fast zu 100 % das Feld, und Microsoft spielte mit Silverlight zeitweilig mit. Auch Oracle wollte aus strategischen Gründen das Feld nicht den Mitbewerbern überlassen. Als sich abzeichnete, dass Applets zu unflexibel für komplexe GUI-Anwendungen und unbeliebt sind, veröffentlichte Oracle nach längerer interner Projektphase Ende 2008 die *JavaFX*-Plattform. JavaFX ist ein ganz neuer GUI-Stack und komplett von Swing und AWT entkoppelt.

In der ersten JavaFX-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. Mit Skriptsprachen auf der Java-VM ist jedoch ein vergleichbares »Feeling« gewährleistet.

Eine Zeit lang sah es so aus, als ob JavaFX den GUI-Stack AWT/Swing beerben könnte. Swing ist eine GUI-Bibliothek, die schon seit Java 1.2 (1998) zum Standard gehört, doch seit 10 Jahren keinen nennenswerten Features mehr bekommen hat, ließen aber regelmäßig Bugs gefixt werden. Doch nach anfänglichem Hype um JavaFX hat Oracle viele Entwickler abgezogen und JavaFX auf das Abstellgleis gestellt. Für Oracle spielen Desktop-Technologien keine Rolle mehr, weshalb sich Oracle 2018 ganz von JavaFX verabschiedet hat: JavaFX wurde aus der Java SE entfernt und ist kein Teil mehr von Java 11. Oracle hat JavaFX in das Projekt *OpenJFX* (<https://wiki.openjdk.java.net/display/OpenJFX/Main>) überführt, wo es als Open-Source-Projekt weiterentwickelt wird – ein Mirror ist <https://github.com/teamfx>. Für JavaFX-Fans ist das eher ein Vorteil als ein Nachteil, denn eine Entkopplung ermöglicht eine flexiblere Weiterentwicklung; OpenJFX ist dann ein Modul wie jedes andere auch.

---

<sup>38</sup> Mit <https://xamarin.com/platform> gibt es interessante Ansätze.

## 1.5 Java-Plattformen: Java SE, Jakarta EE, Java ME, Java Card

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/Jakarta EE.

### 1.5.1 Die Java SE-Plattform

Die *Java Platform, Standard Edition* (Java SE) 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. Es gibt unterschiedliche Implementierungen, etwa das OpenJDK.

#### 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. 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.<sup>39</sup>

Version	Datum	Einige Neuerungen oder Besonderheiten
1.0	Januar 1995	Erste Version. Folgende 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 geschachtelte 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.

Tabelle 1.1 Neuerungen und Besonderheiten der verschiedenen Java-Versionen

<sup>39</sup> 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.«

Version	Datum	Einige Neuerungen oder Besonderheiten
1.3	Mai 2000	Namensdienste mit JNDI, verteilte Programmierung mit RMI/IIOP, 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	Webservices, Skript-Unterstützung, Compiler-API, Java-Objekte an XML-Dokumente binden, System Tray
7	Juli 2011	Kleine Sprachänderungen, NIO2, erste freie Version unter der GNU General Public License (GPL)
8	März 2014	Sprachänderungen Lambda-Ausdrücke, Stream-API
9	September 2017	Modularisierung von Anwendungen
10	März 2018	Lokale Variablen Deklarationen mit var
11	September 2018	Entfernung des java.ee-Moduls

Tabelle 1.1 Neuerungen und Besonderheiten der verschiedenen Java-Versionen (Forts.)

Die Produktzyklen zeigen einige Sprünge, besonders Java 9 wurde zweimal verschoben.

Oracle und Sun waren sehr lange konservativ darin, das Bytecodeformat zu ändern, sodass eine ältere JVM im Prinzip Programme einer neuen Java-Version ausführen konnte. Aber gerade in den Versionen Java 7 und Java 8 gab es doch einige Neuerungen, die die Aufwärtskompatibilität brechen; eine JVM der Version 7 kann also keine Programme mehr ausführen, die ein Java 8-Compiler übersetzt hat. Da einige Teile aus den Java 11-Bibliotheken entfernt wurden, ist auch dadurch die Abwärtskompatibilität eingeschränkt.

### Feature-Release vs. zeitorientiertes Release

20 Jahre lang bestimmten Features die Freigabe von neuen Java-Versionen; die Entwickler setzten bestimmte Neuerungen auf die Wunschliste, und wenn alle Features realisiert und getestet waren, erfolgte die allgemeine Verfügbarkeit (eng. *general availability*, kurz GA). Hauptproblem dieses Feature-basierten Vorgehensmodells waren die Verzögerungen, die mit Problemen bei der Implementierung einhergingen. Vieldiskutiert war das Java 9-Release, weil es unbedingt ein Modulsystem enthalten sollte.

Die Antwort auf diese Probleme, und den Wusch der Java-Community nach häufigeren Releases, beantwortet Oracle mit der »JEP 322: Time-Based Release Versioning«<sup>40</sup>. Vier Releases sind im Jahr geplant:

- ▶ Im März und September erscheinen Haupt-Releases, wie Java 10, Java 11.
- ▶ Updates erscheinen einen Monat nach einem Haupt-Release und dann im Abstand von drei Monaten. Im April und Juli erscheinen folglich die Updates 10.0.1 und 10.0.2. Für Java 11 sind das Oktober 2018 und Januar 2019.

Anders gesagt: Im Halbjahresrhythmus gibt es Updates, die es Oracle erlauben, in der schnelllebigen IT-Zeit die Sprache und Bibliotheken weiterzuentwickeln und neue Spracheigenschaften zu integrieren. Kommt es zu Verzögerungen, hält das nicht gleich das ganze Release auf. Java 10 war im März 2018 das erste Release nach diesem Zeitplan, Java 11 folgte im September 2018.

### Codenamen, Namensänderungen und Vendor-Versionsnummer

Die ersten Java-Versionen waren Java 1.0, Java 1.1 usw. Mit Java 5 entfiel das Präfix »1.« in der Versionskennung des Produkts, sodass es einfach nur Java 5, Java 6 etc. hieß; in den Entwicklerversionen blieb die Schreibweise mit der »1.« aber weiterhin bis Java 9 gültig.<sup>41</sup> In Java 10 kommt durch das »Time-Based Release Versioning« eine Vendor-Kennung hinzu, sodass alternativ zu Java 11 auch von 18.9 die Rede ist; gut lassen sich daran die Jahreszahl und der Monat vom Release ablesen.

### 1.5.2 Java ME: Java für die Kleinen

Die *Java Platform, Micro Edition* (Java ME) ist ein Standard für Geräte mit limitierten Ressourcen, also PDAs, Organizer, Telefone und eingebettete Systeme, die unter 1 MB Speicher haben. Als Laufzeitumgebung gibt es die Oracle Java ME Embedded für die Plattformen: STM32429I-EVAL (Cortex-M4/RTX), STM 32F746GDISCOVERY (Cortex-M7/RTX), Intel Galileo Gen. 2 and Raspberry Pi (ARM 11/Linux). Die Bedeutung der Java ME liegt in Embedded Geräten, wobei Systeme wie ein Raspberry Pi schon wieder so leistungsfähig sind, dass sie eine normale JVM laufen lassen können.

Kräftigen 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 Softwareplattform, sondern auch ein Betriebssystem. Statt einer JVM mit standardisiertem Java-Bytecode nutzt Android einen völlig anderen Bytecode und führt ihn auf der *Dalvik Virtual Machine* aus. Die Klage von Oracle gegen Google wegen Anschuldigungen der Patentverletzung gibt einen Eindruck von der Wichtigkeit des mobilen Markts. Zwar wurde im Mai 2016 die Schadensersatzforderung von Oracle abgewiesen, doch Oracle ging offiziell in

---

<sup>40</sup> <http://openjdk.java.net/jeps/322>

<sup>41</sup> Siehe dazu <http://docs.oracle.com/javase/1.5.0/docs/relnotes/version-5.0.html>.

Berufung gegen das Urteil; die Entscheidung steht noch aus. Die Java ME hat gegen Android in dem Bereich der Smartphones aber verloren, und die Bedeutung wird weiter abnehmen.

Auf der Basis von Java ME spezifiziert Oracle die *Java TV*-Laufzeitumgebung, allerdings stehen alle diese Lösungen nicht im Rampenlicht.

### 1.5.3 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 Smartcard (etwa einer SIM-Karte) Platz findet. Da es jedoch ganz andere Speicheranforderungen an so ein winziges System gibt, ist die Laufzeitumgebung nicht mit der Standard-JVM vergleichbar. Es gibt keine Threads und keine automatische Speicherbereinigung. 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 Standard *Java Card 3.0* 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 automatische Speicherbereinigung.

Mit dem Standard *Java Card* 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.

Oracle schenkt Java Card keine große Aufmerksamkeit. Ein paar Anbieter und Auswahlkriterien listet <https://github.com/martinpaljak/GlobalPlatformPro/tree/master/docs/JavaCard-BuyersGuide> auf.

### 1.5.4 Java für die Großen: Java EE/Jakarta EE

Die *Jakarta EE* – ehemals *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)*, *CDI* für Dependency-Injection, *Servlets*, *JSP*, *JSF* für dynamische Webseiten, die *JavaMail-API* und weitere. Die Implementierung der Enterprise-Spezifikation übernimmt ein Application-Server. Es ist *GlassFish*, die Referenz-Implementierung für die Jakarta EE.

Interessant ist zu beobachten, dass im Laufe der letzten Jahre Teile aus der Enterprise Edition in die Java SE gewandert sind, etwa JAX-WS (Webservices), JNDI (Verzeichnisservice), JAXB (Objekt-XML-Mapping). Das zeigt, dass die APIs heute zum Standard gehören und nicht

mehr nur zwingend zu großen Geschäftsanwendungen. In Java 11 gab es dann wieder einen Schritt zurück, und Technologien wie JAB wurden aus der Java SE entfernt, um sie unabhängig weiterentwickeln zu können.

Die erste Version geht auf das Jahr 1999 zurück, damals noch *J2EE* genannt. In den letzten Jahren gab es um die Enterprise Edition einigen Wirbel. Lange Zeit definierte Sun, dann Oracle den Standard. Dann wurde es ruhig, und Oracle versäumte es, eine leichtgewichtige Alternative zu etablieren, die Microservices und Containervirtualisierung optimal unterstützte. Das stärkte alternative Enterprise-Frameworks wie Spring (Boot). Im September 2017 verkündete Oracle dann, Java EE abzugeben – etwa zur gleichen Zeit wie das Java EE 8 Release – und überließ es der Eclipse Foundation, die es *Eclipse Enterprise for Java (EE4J)* nannten. Allerdings war Oracle gegen die Nutzung des Namens »Java«, sodass der finale Name *Jakarta EE* ist. Die Eclipse Foundation ist heute eine der wichtigsten Gesellschaften, die Java-Standards und -Lösungen herstellerneutral weiterentwickeln.

### 1.5.5 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 Jahr 2001 – von der Java-Community die JSR 1, »Real-time Specification for Java« (kurz *RTSJ*), definiert (mittlerweile JSR 282 für den Nachfolger *RTSJ 1.1*).

Echtzeit-Anwendungen zeichnen sich dadurch aus, dass es eine maximale deterministische Wartezeit gibt, die das System zum Beispiel bei der automatischen Speicherbereinigung 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 automatische Speicherbereinigung ist bei normalen Laufzeitumgebungen eher unbestimmt, auch andere Aktionen unbestimmter Dauer kommen dazu: Läßt Java eine Klasse, dann zur Laufzeit. Das kann zu beliebig vielen weiteren Abhängigkeiten und Laufzyklen 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 Real-Time-Spezifikation. 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, Testtools und Bibliotheken statt. In den letzten Jahren ist es ruhig um Real-time Java geworden.

## 1.6 Java SE-Implementierungen

Die Java SE-Plattform ist im Grunde nur eine Spezifikation und keine Implementierung. Damit Programme übersetzt und ausgeführt werden können, brauchen wir einen konkreten Compiler, eine Laufzeitumgebung und eine Implementierung der Bibliotheken. Es gibt unterschiedliche Implementierungen verschiedener Hersteller. Bestehen die Implementierungen eine Reihe von Tests, die *Technology Compatibility Kit (TCK)* genannt wird, dürfen sie sich »Java SE compatible« nennen.

### 1.6.1 OpenJDK

Das freie und unter der GPL stehende *OpenJDK* (<http://openjdk.java.net/>) bildet die Referenzimplementierung für Java SE. Alle Entwicklungen finden dort statt. Der Fortschritt ist live zu beobachten, regelmäßig fixen und erweitern Hunderte von Entwicklern die Codebasis. Die Quellen für das OpenJDK lassen sich im Mercurial Repository unter <http://hg.openjdk.java.net/jdk/jdk11> einsehen, ein Wechsel auf GitHub wird diskutiert. Viele Technologien, die Oracle vorher nur im Oracle JDK hatte, wurden in das OpenJDK übertragen, etwas Java Flight Recorder, Java Mission Control, Application Class-Data Sharing und ZGC (Zero-Garbage-Collector).

#### OpenJDK-Builds von Oracle

Oracle selbst compiliert das OpenJDK und bietet es an. Aktuelle Versionen sind über <http://jdk.java.net/> verlinkt. Es gibt von Oracle OpenJDK x64-Builds für Windows, Linux und macOS.

Das Oracle OpenJDK steht unter der *GNU General Public License v2* mit der *Classpath Exception* (GPLv2+CPE). Oracle selbst hat angekündigt, bei neuen Versionen keine Updates mehr für die alten Versionen zu veröffentlichen.

#### OpenJDK-Builds von AdoptOpenJDK

Auch unter <https://adoptopenjdk.net/> lässt sich für unterschiedliche Betriebssysteme eine Version herunterladen. Angeboten werden x64-Builds für

- ▶ Windows
- ▶ Linux
- ▶ macOS
- ▶ Linux s390x
- ▶ Linux ppc64le
- ▶ Linux aarch64
- ▶ Linux arm32
- ▶ AIX ppc64

AdoptOpenJDK ist eine Stiftung und betreibt eine Serverfarm, die regelmäßig Builds vom OpenJDK baut und dazu weitere Software wie die JavaFX-Implementierung OpenJFX und alternative Laufzeitumgebungen wie *Eclipse OpenJ9* einbindet. Es wird angekündigt, auch ältere Versionen mit Bugfixes zu versorgen. Zu den Unterstützern zählen IBM und Microsoft.

### Weitere OpenJDK-Builds

Das Unternehmen Azul bietet ebenfalls Builds an, auch lässt sich ein Support-Vertrag abschließen: <http://www.azul.com/downloads/zulu/>. Neben den Plattformen Windows, Linux und macOS gibt es von Azul ebenfalls Docker-Images.

Red Hat bietet neben Linux auch eine Windows-Version vom OpenJDK an: <http://developers.redhat.com/products/openjdk/overview/>. Die Integration in Linux ist sehr gut, und Red Hat pflegt auch noch Sicherheitsupdates in Java 6 und Java 7 ein.

Recht neu von Amazon ist das auf dem OpenJDK basierende Projekt Amazon Corretto (<https://aws.amazon.com/de/corretto/>), allerdings zum Zeitpunkt der Drucklegung nur für Java 8.

Apple pflegte lange Zeit eine komplett eigene JVM, bis Apple den Code an Oracle für das OpenJDK übergab. Auch Google setzt bei Android neuerdings auf das OpenJDK.

Große Unternehmen wie Amazon<sup>42</sup> und Alibaba<sup>43</sup> pflegen ihre eigenen angepassten Java-Implementierungen – für die Allgemeinheit zugänglich sind sie nicht.

### 1.6.2 Oracle JDK

Oracle vermarktet auf der Basis vom OpenJDK sein eigenes Projekt, *Oracle JDK*. Ab Java 11 sind das Oracle JDK und OpenJDK vom Code her (nahezu) identisch. Das Oracle JDK ist die »offizielle« Version, die die Java-Download-Seite von Oracle anbietet. Einer der wenigen klei-

---

<sup>42</sup> Die Amazon EMR Clusters laufen mit dem Amazon Open Java Development Kit (Amazon JDK), sieht <http://www.ateam-oracle.com/using-oracle-data-integrator-odi-with-amazon-elastic-mapreduce-emr/>

<sup>43</sup> <https://www.youtube.com/watch?v=94eTZsNYYBE>

nen Unterschiede sind die Paketierung (das Oracle JDK hat einen Installer, das Oracle OpenJDK ist nur ein ZIP), Versionskennung, ein paar weitere Module.<sup>44</sup>

### Long Term Support (LTS)

Die halbjährlichen Java-Releases haben zur Folge, dass Versionen immer dann veraltet sind, wenn eine neue Version erscheint. In dem Moment, in dem Java 10 kam, war Java 9 veraltet; das Gleiche bei Java 11 – es machte sofort Java 10 zur alten Version. Das allein wäre kein Problem, wenn die älteren Versionen mit Sicherheitsupdates versorgt würden, aber Oracle investiert für die Allgemeinheit in die alten Versionen keine Zeit und Mühe mehr.

Für Unternehmen ist das ein Problem, denn es erzeugt Stress, mit den Änderungen mitziehen zu müssen. Aus diesem Grund bietet Oracle alle drei Jahre eine Java-Version mit Long Term Support (LTS) und versorgt sie mit Updates und Sicherheitspatches. Die nächsten LTS-Versionen nach Java 8 sind Java 11 (September 2018) und dann nach 3 Jahren Java 17 (September 2021). Das ist für weniger agile Unternehmen gut. Oracle will ihre Java SE 8-Implementierung noch bis mindestens Dezember 2020 pflegen, geplant ist bis März 2025<sup>45</sup> und Java 11 bis 2023, beides Releases mit LTS.

### Kommerzialisierung des Oracle JDK

Auf den ersten Blick sieht das gut aus: Es gibt regelmäßige Updates für agile Unternehmen, und die konservativen Unternehmen setzen auf eine LTS-Version. Das Problem ist allerdings, dass alle Oracle JDK-Versionen nicht kommerziell eingesetzt werden dürfen; der Hersteller erlaubt die Nutzung nur für »development, testing, prototyping or demonstrating purposes«. Für Java 8 endet die Schonfrist im Januar 2019. Das dürfte vielen Entwicklern gar nicht bewusst sein, denn seit 20 Jahren sind Unternehmen daran gewöhnt, das Oracle JDK für alle einzusetzen. Und wir wissen alle, wie viele Menschen wirklich die Lizenzbedingungen lesen ...

Wer das Oracle JDK kommerziell einsetzen möchte und nicht nur in einer Entwicklungs- oder Testumgebung, muss eine Lizenz von Oracle erwerben. Es wird monatlich abgerechnet, die Vertragszeit ist mindestens ein Jahr. Es stehen zwei Modelle zur Auswahl:

Java SE Subscription	Java SE Desktop Subscription
Für Serveranwendungen	Für Client-Anwendungen
Abrechnung pro Prozessor	Abrechnung pro Benutzer
Bis 25 US\$/Monat, für 1–99 Benutzer	Bis 2,50 US\$/Monat für 1–999 Benutzer/Clients

Tabelle 1.2 Zwei Lizenzmodelle für Oracle Java SE

44 <https://blogs.oracle.com/java-platform-group/oracle-jdk-releases-for-java-11-and-later>

45 <http://www.oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf>

Oracle wendet bei der Java SE Subscription das gleiche Geschäftsmodell wie bei der Oracle-Datenbank an. Wie genau ein Rechner in der Cloud mit einer unbestimmten Anzahl der Prozessoren abgerechnet werden soll, ist noch unklar.<sup>46</sup> Interessenten sollten die »Oracle Java SE Subscription FAQ« unter <http://www.oracle.com/technetwork/java/javaseproducts/overview/javasesubscriptionfaq-4891443.html> studieren und Oracle-Berater hinzuziehen. Wer Client- und Serveranwendungen nutzt, muss zweimal bezahlen, statt »write once, run anywhere« nun »write once, pay everywhere«.

Die Kosten können sich schnell summieren, doch bekommen Unternehmen damit Support und insbesondere für Java 8 immer noch für einige Jahre Unterstützung. Nachteil ist, dass es das Subscription-Modell nur für die LTS-Versionen gibt, Unternehmen also gezwungen werden, größere Versionssprünge zu machen. Nach Java 11 kommt erst im September 2021 die Version Java 17 mit dem nächsten LTS.

### Java Platform, Standard Edition in drei Paketen: JDK, JRE, Server JRE

In der Oracle 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 Pakete geschnürt:

- ▶ 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 *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 wie Management-Tools.
- ▶ Das *Server JRE* ist für 64-Bit-Server-Umgebungen gedacht und enthält anders als das herkömmliche JRE nicht den Auto-Updater, das Plugin-Tool oder einen Installer. Wie das JDK enthält es hingegen eine Management-Konsole.

Die drei Produkte können von der Webseite <http://www.oracle.com/technetwork/java/javase/downloads/> bezogen werden – die Lizenzbestimmungen sind einzuhalten.

### Oracle JDK 11 Certified System Configurations

Eine ideale, perfekt getestete und unterstützte Umgebung gilt als *Oracle Certified System Configurations*. Das ist eine Kombination aus Betriebssystem mit installierten Service-Packs; das beschreibt das Unternehmen unter <https://www.oracle.com/technetwork/java/javase/>

---

<sup>46</sup> <http://houseofbrick.com/the-oracle-parking-garage/> ist dann gar nicht zum Lachen ...

[documentation/jdk11certconfig-5069638.html](http://documentation/jdk11certconfig-5069638.html). Dazu zählen etwa die 64-Bit-Betriebssysteme (32 Bit gibt es nicht mehr):

- ▶ Windows 10
- ▶ Windows Server 2016
- ▶ Red Hat Enterprise Linux x64 Version 6 und 7
- ▶ macOS x64 ab Version 10.11
- ▶ Solaris SPARC ab Version 11.3

Bei gemeldeten Bugs auf nicht zertifizierten Plattformen kann dann schnell ein »Sorry, das ist eine nicht unterstützte Plattform, das schauen wir uns nicht weiter an« folgen. Bei Linux ist zum Beispiel die Gentoo-Distribution nicht in der Liste, wäre also »Not certified on Oracle VM«. Das heißt nicht, dass Java dort nicht zu 100 % läuft, sondern nur, dass es im Fehlerfall eben keinen Fix geben muss.

## 1.7 Die Installation des Oracle OpenJDK

Im Folgenden wollen wir das Oracle OpenJDK, die Basisimplementierung der Java Platform, Standard Edition (Java SE), installieren. Unter der Webseite <http://jdk.java.net/11/> finden wir Downloads für

- ▶ Linux
- ▶ macOS
- ▶ Windows

Wer sich für Java 12 interessiert, kann unter <http://jdk.java.net/12/> auch schon eine Vorabversion herunterladen.

### 1.7.1 OpenJDK unter Windows installieren

Während das Oracle JDK für Windows ein Installationsprogramm anbietet, kommt das OpenJDK nur als komprimiertes Archiv. Wir laden von der Webseite hinter dem Link **WINDOWS / x64** die ZIP-Datei, die etwa *openjdk-11+28\_windows-x64\_bin.zip* heißt; die Build-Nummer kann natürlich auch höher sein.

#### Programme und Ordner im Java-Verzeichnis

Nach dem Auspacken entsteht ein Ordner *jdk-11* mit dem OpenJDK. Es enthält ausführbare Programme wie Compiler und Interpreter sowie die Bibliotheken und Quellcodes, allerdings keine Javadoc. Wenn wir die Rechte haben, wollen wir den JDK-Ordner in der Windows-Programmordner setzen und für die weiteren Beispiele folgenden Ort annehmen: *C:\Program-*

*me\Java\jdk-11* (bzw. *C:\Program Files\Java\jdk-11*). Es gibt allerdings keinen vorgeschriebenen Ort.

Ordner/Datei	Bedeutung
<i>bin</i>	Hier befinden sich Entwicklungswerkzeuge, unter anderem der Interpreter <i>java</i> und beim JDK der Compiler <i>javac</i> .
<i>conf</i>	Konfigurationsdateien, Anpassungen sind hier selten nötig.
<i>include</i>	Dateien für die Anbindung von Java an C(++)-Programme
<i>jmods</i>	Java-Module vom JDK, etwa das Basis-Modul
<i>legal</i>	eine Reihe von COPYRIGHT-Textdateien
<i>lib</i>	interne JDK-Tools
<i>lib/src.zip</i>	Archiv mit dem Quellcode der öffentlichen Bibliotheken
<i>release</i>	Datei mit Schüssel-Wert-Paaren

Tabelle 1.3 Ordnerstruktur

### Testen der Installation und Pfade setzen

Gehen wir in das *bin*-Verzeichnis *C:\Program Files\Java\jdk-11\bin*, können wir aufrufen:

```
C:\Program Files\Java\jdk-11\bin>java -version
openjdk version "11.0.1" 2018-10-16
OpenJDK Runtime Environment 18.9 (build 11.0.1+13)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.1+13, mixed mode)
```

### Den Suchpfad für die aktuelle Sitzung 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 *C:\Program Files\Java\jdk-11\bin* an den Anfang der Suchliste, damit im Fall von Altinstallationen immer das neue JDK verwendet wird:

```
set PATH=C:\Program Files\Java\jdk-11\bin;%PATH%
```

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

### Den Suchpfad in der Systemsteuerung setzen

Damit die Pfadangabe auch nach einem Neustart des Rechners noch verfügbar ist, müssen wir abhängig vom System unterschiedliche Einstellungen vornehmen: Wir suchen in Windows 10 nach UMGEBUNGSVARIABLEN FÜR DIESES KONTO BEARBEITEN. Es öffnet sich ein Dialog UMGEBUNGSVARIABLEN. Im oberen Teil, BENUTZERVARIABLEN, suche nach PATH. Mit BEARBEITEN... verändere den Eintrag, und füge dem Pfad hinter einem Semikolon das JDK-bin-Verzeichnis (`C:\Program Files\Java\jdk-11\bin`) hinzu. Bestätige die Änderung mit OK.

Wenn wir jetzt eine neue Eingabeaufforderung öffnen, können wir `javac` für den Java-Compiler aufrufen oder `java` für die Laufzeitumgebung. Auch kann eine Entwicklungsumgebung wie Eclipse gestartet werden.

### OpenJDK deinstallieren

Nach dem Löschen des Ordners ist Java deinstalliert. Falls wir den PATH gesetzt haben, sollten wir ihn wieder aus den Systemeigenschaften löschen.

### JDK 11-Dokumentation

Die Hauptseite für die Dokumentation ist <https://docs.oracle.com/en/java/javase/11/>. Die API-Dokumentationen der Standardbibliothek und die der Tools sind kein Teil des JDK; eine Trennung ist sinnvoll, denn sonst würde der Download nur unnötig größer, die Dokumentation kann schließlich auch online angeschaut werden. Die API-Beschreibung kann online unter <https://docs.oracle.com/en/java/javase/11/docs/api> eingesehen werden.

## 1.8 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. Die Beispiele im Buch basieren auf Windows.

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.

### 1.8.1 Ein Quadratzahlen-Programm

Das erste Java-Programm soll einen Algorithmus ausführen, der die Quadrate (engl. *squares*) der Zahlen von 1 bis 4 ausgibt. Wir setzen den Quellcode (engl. *source code*) beispielhaft in das Verzeichnis C:\firstlove. Der Name der Quellcodedatei ist *Squares.java*. Da der Quellcode reiner Text ist, kann er mit jedem Texteditor angelegt und editiert werden.

Hinweis für Windows-Nutzer: Unter Windows findet sich der Editor *Notepad* unter START • PROGRAMME • ZUBEHÖR • EDITOR. Beim Abspeichern mit Notepad unter DATEI • SPEICHERN UNTER... muss bei DATEINAME *Squares.java* stehen und beim Dateityp ALLE DATEIEN ausgewählt sein, damit der Editor nicht automatisch die Dateiendung .txt vergibt.

**Listing 1.1** C:\firstlove\Squares.java

```
/*
 * Erstes Java-Beispielprogramm.
 * @version 1.02    26 Dez 2013
 * @author Christian Ullenboom
 */
public class Squares {

    static int quadrat( int n ) {
        return n * n;
    }

    static void ausgabe( int n ) {
        for ( int i = 1; i <= n; i = i + 1 ) {
            String s = "Quadrat(" + i + ") = " + quadrat( i );
            System.out.println( s );
        }
    }

    public static void main( String[] args ) {
        ausgabe( 4 );
    }
}
```

Die ganze Programmlogik sitzt in einer Klasse Squares, die drei Methoden enthält. Alle Methoden in einer objektorientierten Programmiersprache wie Java müssen in Klassen platziert werden. Die erste Methode, *quadrat(int)*, 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(int)*. Zum Schluss muss es noch ein be-

sonderes Unterprogramm `main(String[])` geben, das für den Java-Interpreter den Einstiegs-punkt bietet. Die Methode `main(String[])` ruft dann die Methode `ausgabe(int)` auf.

### 1.8.2 Der Compilerlauf

Wir wechseln zur Eingabeaufforderung (Konsole) und in das Verzeichnis mit dem Quellcode. Liegt die Quellcodedatei vor, übersetzt der Compiler sie in Bytecode. Für unsere Java-Klasse in der Datei `Squares.java` heißt das:

```
C:\firstlove>javac Squares.java
```

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

Bei Aufruf von `javac` muss die zu übersetzende Datei komplett mit Dateiendung angeben werden. Doch auch Wildcards sind möglich: so übersetzt `javac *.java` alle Java-Klassen im aktuellen Verzeichnis.

Die Beachtung der Groß- und Kleinschreibung ist zwar unter Windows nicht wichtig, doch sollte sie eingehalten werden.

### Compilerfehler

Findet der Compiler in einer Zeile einen syntaktischen 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(int)`-Methode einen Fehler in Zeile 9 ein (das Semikolon fällt der Löschtaste zum Opfer). Der Compilerdurchlauf meldet:

```
Squares.java:9: ';' expected.  
    return n * n  
           ^  
1 error
```

Wenn der Compiler aufgrund eines syntaktischen Fehlers eine Übersetzung in Java-Bytecode nicht durchführen kann, sprechen wir von einem *Compilerfehler* (engl. *compile-time error*) oder *Übersetzungsfehler*. Auch wenn der Begriff »Compilerfehler« so klingt, als ob der Compiler selbst einen Fehler hat, ist es doch unser Programm.

### 1.8.3 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 Laufzeit-

umgebung 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:\firstlove>java Squares
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.



#### Tipp

Ab Java 11 lassen sich so genannte Single-File Source-Code Programs auch ohne Übersetzung von der Laufzeitumgebung ausführen.<sup>47</sup>

#### 1.8.4 Häufige Compiler- und Interpreter-Probleme

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 Squares.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
```

Die Lösung ist hier also, *javac* in den Suchpfad aufzunehmen, wie wir es vorher schon beschrieben haben.

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:

---

<sup>47</sup> <https://openjdk.java.net/jeps/330>

```
$ java Q
```

Fehler: Hauptklasse Q konnte nicht gefunden oder geladen werden.

Ursache: java.lang.ClassNotFoundException: Q

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 Squares, folgt die Fehlermeldung:

```
$ java Squares
```

Fehler: Hauptmethode in Klasse Squares nicht gefunden. Definieren Sie die Hauptmethode als:

`public static void main(String[] args):`

oder eine JavaFX-Anwendung muss `javafx.application.Application` erweitern

### Hinweis

Der Java-Compiler und die Java-Laufzeitumgebung haben einen Schalter `-help`, der weitere Informationen zeigt.



## 1.9 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 unterkringelt 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 Webservice-Zugriffe generieren.

### 1.9.1 Eclipse IDE

Um die alte *WebSphere*-Reihe und die Umgebung *Visual Age for Java* abzulösen, entwickelte IBM *Eclipse* (<http://www.eclipse.org>). Im November 2001 veröffentlichte IBM die IDE als Open-Source-Software, und 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. Alle halbe Jahre gibt es ein Update.

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

Da *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, auf Platz zwei steht IntelliJ, *NetBeans* ist weit abgeschlagen.

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 so genannten *inkrementellen Compiler* arbeitet. Speichert der Anwender eine Java-Quellcodedatei, übersetzt der Compiler diese Datei automatisch. Dieses Feature nennt sich *Autobuild*.

Treibende Kraft hinter Eclipse war Erich Gamma, der 2011 zu Microsoft ging und dort Teamleiter für *Microsoft Visual Studio Code* (kurz VSC) wurde. VSC ist ein neuer Editor und schlankere Entwicklungsumgebung. Für die Webentwicklung hat VSC schnell Freunde gefunden, auch Java-Entwicklung ist möglich, wobei die Refactoring-Möglichkeiten noch hinter Eclipse oder IntelliJ stehen. Das kann sich jedoch schnell ändern.

### 1.9.2 IntelliJ IDEA

Dass Unternehmen mit Entwicklungsumgebungen noch Geld verdienen können, zeigt JetBrains, ein aus Tschechien stammendes Softwarehaus. Die Java-Entwicklungsumgebung *IntelliJ IDEA* (<https://www.jetbrains.com/idea>) gibt es in zwei Editionen:

- ▶ Die freie, quelloffene Grundversion, die alles abdeckt, was zur Java SE-Entwicklung nötig ist, nennt sich *Community Edition*. Der Quellcode ist auf GitHub für jeden einsehbar.<sup>48</sup>
- ▶ Die kommerzielle *Ultimate Edition* richtet sich an Java Enterprise-Entwickler. Sie kostet 499 € im ersten Jahr, 399 € im zweiten Jahr, 299 € in den darauffolgenden Jahren.<sup>49</sup> Mit dem Abonnement-Modell hat sich JetBrains unter den Entwicklern/Unternehmen keine Freunde gemacht.

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 im Bereich Maven oder Spring sehr eingeschränkt, hier ist die Ultimate Edition unverzichtbar – die gleiche Leistung bekommen Nutzer der freien

---

<sup>48</sup> <http://github.com/JetBrains/intellij-community>

<sup>49</sup> <https://www.jetbrains.com/store/?fromMenu#edition=commercial>

*Eclipse IDE for Java EE Developers* frei. JetBrains ist Entwickler der Programmiersprache Kotlin, weshalb natürlich die Unterstützung in IntelliJ optimal ist.

### 1.9.3 NetBeans

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 Enterprise-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://netbeans.org>) bietet komfortable Möglichkeiten zur Java SE- und Java Enterprise-Entwicklung mit Editoren und Wizards für die Erstellung grafischer Oberflächen 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 zum anderen unterstützt Oracle auch sehr aktiv Eclipse und bietet spezielle Java Enterprise-Plugins, wie das *Oracle Enterprise Pack for Eclipse* (OEPE).<sup>50</sup> Bekam NetBeans von Oracle nach der Übernahme von Sun Zuschuss und Unterstützung, wurde es dann ruhig. Oracle hat letztendlich die Codebasis an die Apache Foundation übergeben und sich damit von der IDE getrennt. Das neue Zuhause ist <https://netbeans.apache.org/>, und *Apache NetBeans 9.0* unterstützt Java 9 und Java 10, zur Drucklegung noch nicht Java 11.

## 1.10 Eclipse IDE im Speziellen

Die Entwicklungsumgebung Eclipse ist größtenteils in Java programmiert, und die aktuelle Version 4.9 (Eclipse 2018-9) benötigt zur Ausführung mindestens eine Java-Version 8.<sup>51</sup> Da Teile wie die grafische Oberfläche in C implementiert sind, ist Eclipse nicht 100 % pures Java, und beim Download ist auf das passende System zu achten.

---

<sup>50</sup> <http://www.oracle.com/technetwork/developer-tools/eclipse/downloads/index.html>

<sup>51</sup> Natürlich kann Eclipse auch Klassendateien für Java 1.0 erzeugen, nur die IDE selbst benötigt mindestens Java 8. Zu den Gründen siehe auch [https://wiki.eclipse.org/Eclipse/Installation#Eclipse\\_4.7\\_.28Oxygen.29](https://wiki.eclipse.org/Eclipse/Installation#Eclipse_4.7_.28Oxygen.29).

Zur Installation unter Windows, Linux und macOS gibt es zwei Möglichkeiten:

- ▶ ein nativer Installer, der Komponenten aus dem Netz lädt
- ▶ ein Eclipse-Paket: ZIP-Archiv auspacken und starten

## Eclipse IDE-Pakete

Eclipse gliedert sich in unterschiedliche *Pakete*. Die wichtigsten sind:

- ▶ **Eclipse IDE for Java Developers**: Dies ist die kleinste Version zum Entwickeln von Java SE-Anwendungen.
- ▶ **Eclipse IDE for Java EE Developers**: Diese Version enthält diverse Erweiterungen für die Entwicklung von Webanwendungen und Java Enterprise-Applikationen. Dazu zählen unter anderem Data Tools Platform, Eclipse Git Team Provider, Eclipse Java EE Developer Tools, JavaScript Development Tools, Maven Integration for Eclipse, Eclipse XML Editors and Tools.
- ▶ **Eclipse IDE for C/C++ Developers**: Eclipse als Entwicklungsumgebung für C(++)-Programmierer. Es ist das kleinste Paket, da es ausschließlich für die Programmiersprache C(++) und nicht für Java ist.

Die Pakete sind unter <https://www.eclipse.org/downloads/packages/> zum Download verlinkt.



### Hinweis

Üblicherweise arbeiten Entwickler mit dem Paket *Eclipse IDE for Java EE Developers*, das ein paar Erweiterungen (Plugins) mitbringt, etwa einen JavaScript- oder HTML-Editor. Auch wir arbeiten mit diesem Paket im Buch. Der Download ist unter <https://www.eclipse.org/downloads/packages/release/2018-09/r/eclipse-ide-java-ee-developers>.

## Download weiterer Versionen

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 erlauben. 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. Die aktuellen Releases und Builds gibt es unter <http://download.eclipse.org/eclipse/downloads>.



### Hinweis

Eclipse 4.9 unterstützt Java 10, allerdings Java 11 nur mit einem Plugin.

#### 1.10.1 Eclipse IDE entpacken und starten

Eine Installation von Eclipse im typischen Sinn mit einem Installer ist nicht unbedingt erforderlich. Die folgenden Schritte beschreiben die Benutzung unter Windows. Nach dem Download und Auspacken des ZIP-Archivs *eclipse-jee-2018-09-win32-x86\_64.zip* gibt es einen Ordner *eclipse* mit der ausführbaren Datei *eclipse.exe*. Das Eclipse-Verzeichnis lässt sich frei wählen.

Nach dem Start von *eclipse.exe* folgen ein Willkommensbildschirm und ein Dialog, in dem der *Workspace* ausgewählt werden muss. Mit einer Eclipse-Instanz ist immer 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.

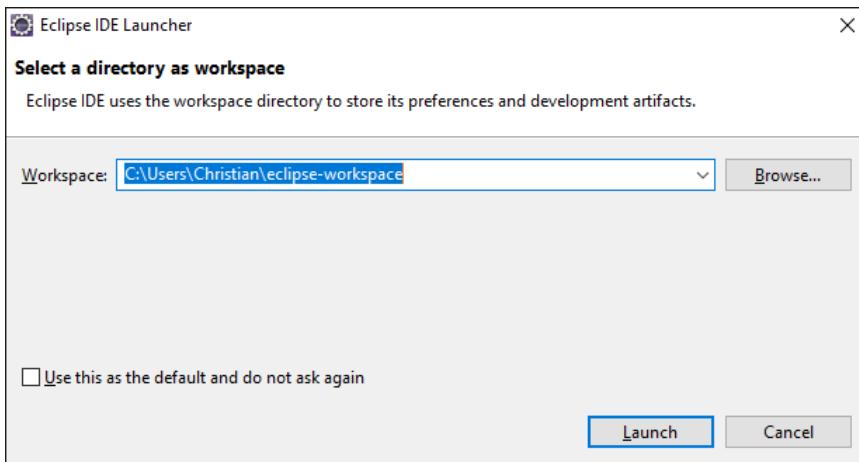


Abbildung 1.3 Workspace in Eclipse auswählen

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.

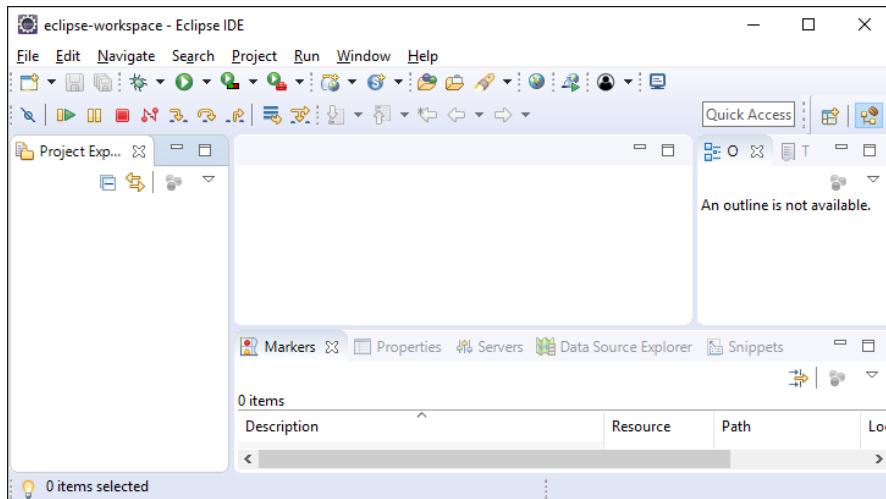


Abbildung 1.4 Eclipse IDE nach dem Start

## Java 11-Unterstützung installieren

Eclipse 4.9 unterstützt durch den eigenen Compiler unterschiedliche Java-Versionen, allerdings nicht Java 11. Natürlich lässt sich grundsätzlich ein Java 10-Projekt aufsetzen und auf das JDK 11 einstellen, doch mit einer Eclipse-Erweiterung geht das besser.

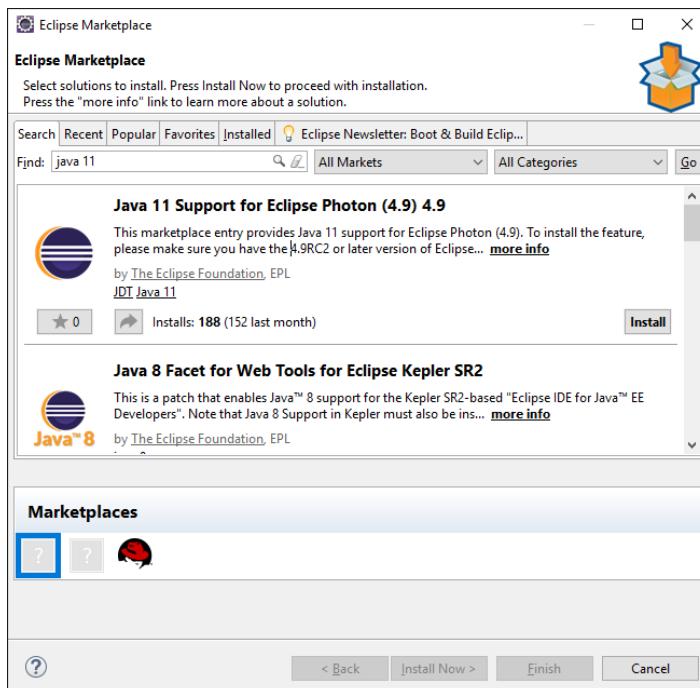


Abbildung 1.5 Marketplace nach Java 11 durchsuchen

Gehen wir auf HELP • ECLIPSE MARKETPLACE ... Dort geben wir hinter FIND den Suchbegriff »Java 11« ein. Es folgen Suchergebnisse mit JAVA 11 SUPPORT FOR ECLIPSE PHOTON (4.9) 4.9 (siehe Abbildung 1.5).

Ein Klick auf INSTALL startet den Download und die Installation. Wir starten neu, und Java 11 ist in den Menüeinträgen.

### Java 11 einstellen, »Installed JREs«

Startet Eclipse, wertet es die Umgebungsvariable PATH aus, in der die Java-Laufzeitumgebung eingetragen ist. Das ist auch die Java-Version, die Eclipse als Java SE-Implementierung annimmt und Eclipse extrahiert auch die Java-Dokumentation aus den Quellen. Die Java-Version, mit der Eclipse läuft, muss aber nicht identisch mit der Version sein, mit der wir entwickeln, daher wollen wir prüfen, dass zur Entwicklung Java 11 eingetragen ist.

Die Einstellung über das JRE/JDK und die Java-Version kann global oder lokal für jedes Projekt gesetzt werden, wir bevorzugen eine globale Einstellung. Wähle im Menü WINDOW • PREFERENCES, was einen umfassenden Konfigurationsdialog öffnet. Gehe dann im Baum links auf JAVA weiter zu INSTALLED JREs. Durch eine ältere Java-Installation kann etwas anderes als Java 11 eingetragen sein – der Dialog sieht so aus:

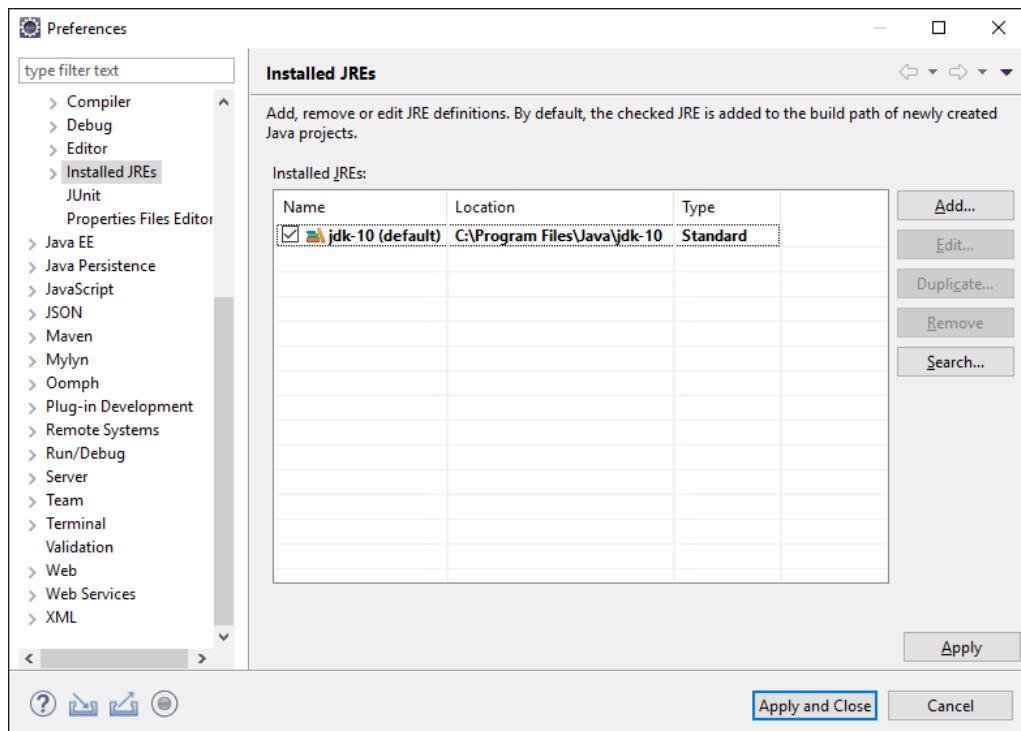


Abbildung 1.6 Es ist Java 10 eingetragen.

Klicken wir SEARCH..., wählen wir im Dialog das Verzeichnis aus, in dem das OpenJDK entpackt liegt, etwa *C:\Program Files\Java*. Nach beenden des Dialogs dauert es etwas, und Eclipse sucht uns alle Java-Installation in diesem Ordner heraus.

Die Liste kann je nach Installation länger oder kürzer sein, doch sollte sie den Eintrag **JDK-11** enthalten. Wir setzen ein Häkchen beim gewünschten JDK, in unserem Fall bei **JDK-11**, das dann fett hervorgehoben wird und **(DEFAULT)** wird. Nach dem Klick auf **APPLY** übernimmt Eclipse die Einstellungen, aber der Konfigurationsdialog bleibt offen. Wir sind mit den Einstellungen noch nicht fertig ...

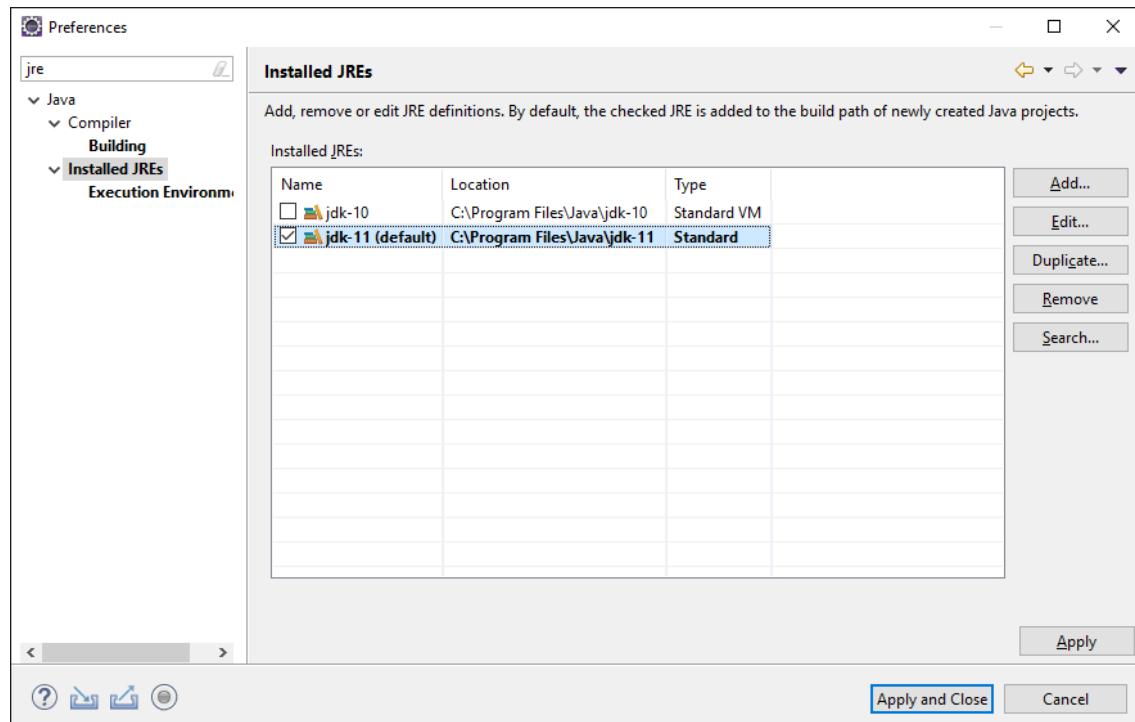


Abbildung 1.7 Java 11 ist eingetragen.

### Java 11 einstellen, »Execution Environments«

Eclipse hat eine Abstraktionsschicht für Laufzeitumgebungen, sodass Projekte ausdrücken können: »irgendeine Java 11-Version«, ohne sich binden zu müssen an OpenJDK, Oracle JDK usw. Die Abstraktion erreicht Eclipse über EXECUTION ENVIRONMENTS, das im Konfigurationsdialog direkt unter INSTALLED JREs liegt.

Nun ist die Ausführungsumgebung auf Java 11 gestellt. Wir können Java 11-Projekte mit dieser Umgebung anlegen, die durch diese Abbildung vom eingestellten OpenJDK ausgeführt werden. Sollte auf einem anderen Rechner das Projekt importiert werden und liegt die JDK-Installation in einem komplett anderen Pfad, so ist nur in der lokalen Eclipse-Installation das Verzeichnis zum JDK zu setzen, nicht aber in den Java-Projekten.

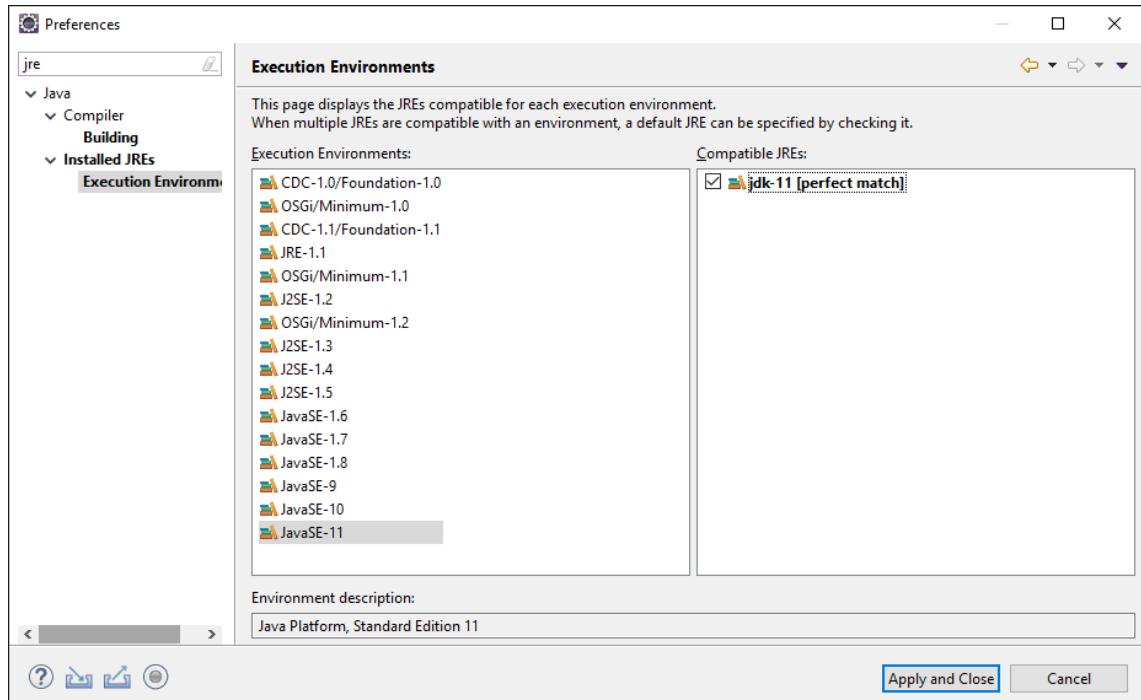


Abbildung 1.8 Ausführungsumgebung auf Java 11 setzen

### Java-Compiler auf Version 11 stellen

Eclipse nutzt einen eigenen Compiler und nicht den Compiler aus dem JDK. Wir müssen daher sicherstellen, dass der Compiler die Syntax von Java 11 unterstützt. Im Konfigurationsdialog (WINDOW > PREFERENCES) suchen wir JAVA • COMPILER. Rechts im Feld stellen wir sicher, dass bei COMPILER COMPLIANCE LEVEL auch die Version 11 eingestellt ist.

Wir haben also zwei Einstellungen vorgenommen: die passende Laufzeitumgebung konfiguriert und dann den Compiler auf die gewünschte Version gesetzt. Jetzt kann der Konfigurationsdialog mit APPLY AND CLOSE geschlossen werden.

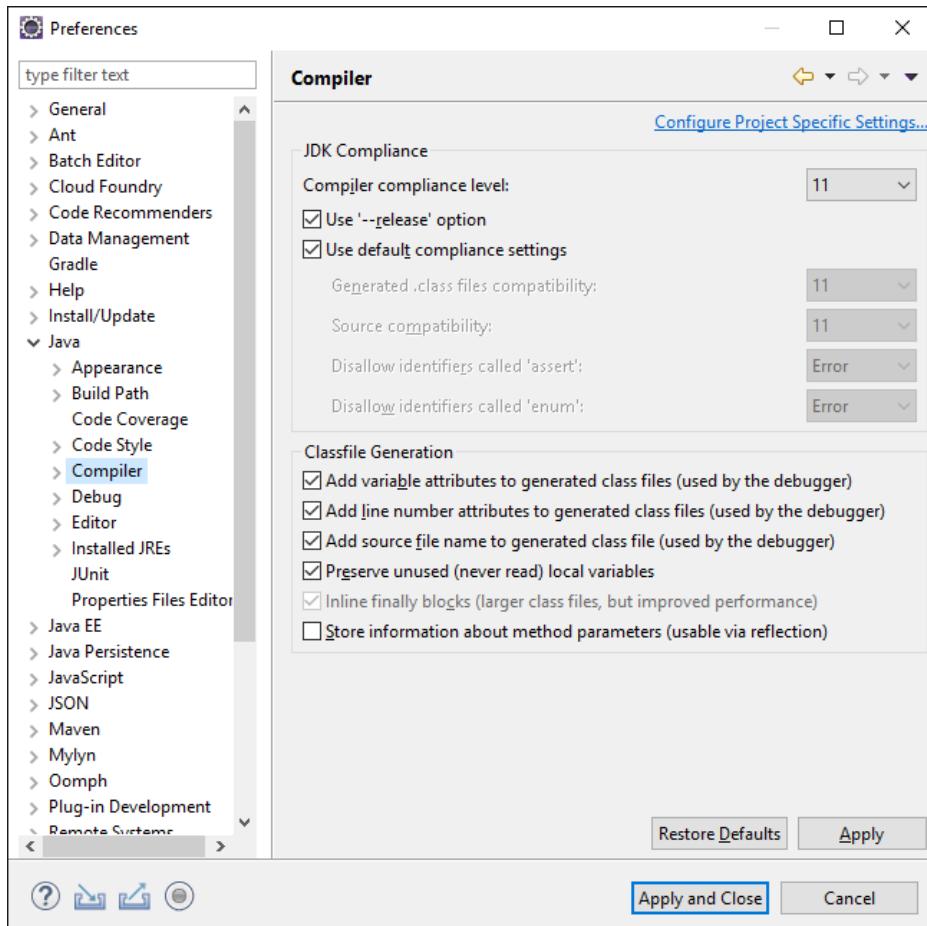


Abbildung 1.9 Java-Compiler auf Version 11 stellen

### 1.10.2 Das erste Projekt anlegen

Nach dem Start von Eclipse muss ein Projekt angelegt (oder ein existierendes eingebunden) werden – ohne dieses lässt sich kein Java-Programm ausführen. Im Menü ist dazu FILE • NEW • OTHER auszuwählen. Es öffnet sich ein Dialog mit allen möglichen Projekten, die Eclipse anlegen kann.

Wählen wir JAVA PROJECT. Der Klick auf NEXT blendet einen neuen Dialog für weitere Einstellungen ein. Unter PROJECT NAME geben wir einen Namen für unser Projekt ein, zum Beispiel: »Insel«. Mit dem Projekt ist ein Pfad verbunden, in dem die Quellcodes, Ressourcen und Klassendateien 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 sollte der erste der drei Punkte gewählt sein: USE AN EXECUTION ENVIRONMENT JRE steht auf JAVASE-11.

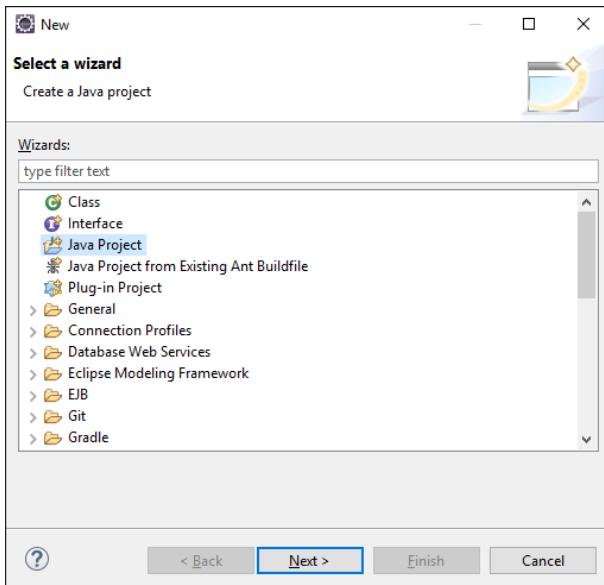


Abbildung 1.10 Neues Java-Projekt in Eclipse anlegen

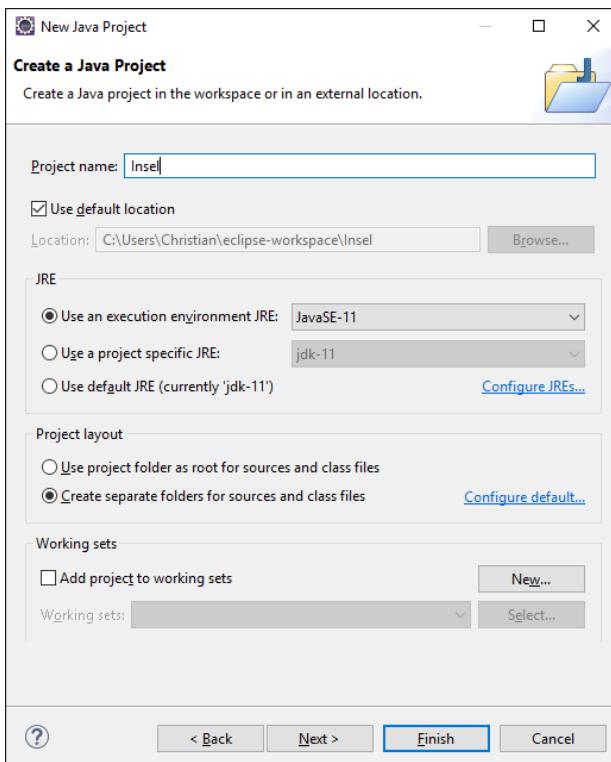


Abbildung 1.11 Einstellung des neuen Java-Projekts

Die Schaltfläche FINISH schließt das Anlegen ab. Da es sich um ein Java-Projekt handelt, möchte Eclipse je nach vorheriger Perspektive unter Umständen in eine Java-Ansicht gehen – den folgenden Dialog sollten wir mit YES bestätigen.

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.

### 1.10.3 Verzeichnisstruktur für Java-Projekte \*

Ein Java-Projekt braucht eine ordentliche Ordnerstruktur, und hier gibt es zur Organisation der Dateien unterschiedliche Ansätze. Die einfachste Form ist, Quellen, Klassendateien und Ressourcen in ein Verzeichnis zu setzen. Doch diese Mischung ist in der Praxis nicht vorteilhaft.

Im Allgemeinen finden sich zwei wichtige Hauptverzeichnisse: *src* für die Quellen und *bin* für die erzeugten Klassendateien. Diese Einteilung nutzt Eclipse standardmäßig, wenn ein neues Java-Projekt angelegt wird. Ein eigener Ordner *lib* ist sinnvoll für Java-Bibliotheken.

#### Maven Standard Directory Layout

Noch weiter in der Aufteilung geht ein Projekt nach dem Maven-Standard, es definiert ein *Standard Directory Layout* mit den folgenden wichtigen Ordnern (siehe Tabelle 1.4).<sup>52</sup>

Ordner	Inhalt
<i>src/main/java</i>	Quellen
<i>src/main/resources</i>	Ressourcen
<i>src/main/config</i>	Konfigurationen
<i>src/main/scripts</i>	Skripte
<i>src/test/java</i>	Testfälle

Tabelle 1.4 Standard Directory Layout

52 Komplett unter <http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html>

Ordner	Inhalt
<i>src/test/resources</i>	Ressourcen für Testfälle
<i>src/site</i>	Website
<i>target</i>	Ergebnis eines Builds
<i>LICENSE.txt</i>	Lizenzform des Projekts (etwa Apache, BSD, GPL)
<i>NOTICE.txt</i>	Hinweise oder Abhängigkeiten zu anderen Projekten
<i>README.txt</i>	Lies-mich-Datei

Tabelle 1.4 Standard Directory Layout (Forts.)

Einen *lib*-Ordner gibt es nicht, den verwalten Maven-Projekte an anderer Stelle, in einem lokalen Repository. Wenn das Projekt gebaut wird und zum Beispiel zu einem Webprojekt wird, werden die JAR-Dateien automatisch mit in den *target*-Ordner gesetzt.

Die Eclipse IDE for Java EE Developers enthält Maven und kann direkt Maven-Projekte anlegen und folglich auch die Verzeichnisstruktur. Dazu wähle FILE • NEW • OTHER ... • MAVEN/ MAVEN PROJECT. Für ein erstes Miniprojekt reicht es, den Schalter bei CREATE A SIMPLE PROJECT (SKIP ARCHETYPE SELECTION) zu setzen. Dann sind zwingend eine GROUP ID (globale Projektkennung wie ein Paketname) und ARTIFACT ID (Name des JARs) anzugeben – die Versionsnummer ist im Dialog vorbelegt. Dann rappelt es in der Kiste, und Maven beginnt mit dem Download einiger Dateien. Am Ende stehen vier *src*-Ordner, ein *target*-Ordner und eine Datei *pom.xml* für die Maven-Beschreibung. Vorhandene Java-Projekte lassen sich auch in Maven-Projekte konvertieren (dabei ändert sich aber nicht die Verzeichnisstruktur). Dazu wähle im PACKAGE EXPLORER das Projekt aus und im Kontextmenü CONFIGURE • CONVERT TO MAVEN PROJECT. Wir kommen in [Abschnitt 15.10](#), »Maven: Build-Management und Abhängigkeiten auflösen«, detaillierter auf Maven zurück.



#### 1.10.4 Eine Klasse hinzufügen

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

## 1.10.5 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**.

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.

### 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. 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, 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. Im Rahmen rechts – betitelt mit SAVE REQUIRED DIRTY EDITORS BEFORE LAUNCHING – aktivieren wir dann ALWAYS.

## 1.10.6 Projekt einfügen, Workspace für die Programme wechseln

Alle Beispielprogramme im Buch gibt es im Netz unter <http://tutego.de/javabuch>. Die Beispielprogramme sind in einem kompletten Eclipse-Workspace zusammengefasst, lassen sich aber natürlich in jeder anderen IDE nutzen. Da Eclipse nur einen Workspace gleichzeitig geöffnet halten kann, müssen wir mit FILE • SWITCH WORKSPACE... • OTHER den Workspace neu setzen. Eclipse beendet sich dann und startet anschließend mit dem neuen Workspace neu.

### 1.10.7 Plugins für Eclipse

Zusätzliche Anwendungen, die in Eclipse integriert werden können, werden *Plugins* genannt. 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.

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).

Durch die riesige Anzahl an Plugins ist nicht immer offensichtlich, welches Plugin gut ist; eine Auswahl stellt <http://tutego.de/java/eclipse/plugin/eclipse-plugins.html> zusammen. Auch ist nicht klar, welches Plugin mit welchem anderen Plugin gut zusammenspielt; aufeinander abgestimmte Sammlungen sind da Gold wert. Mit dem Paket *Eclipse IDE for Java EE Developers* sorgt die Eclipse Foundation für die Java Enterprise-Entwicklung schon vor, es enthält alles, um Webapplikationen oder Webservices zu entwickeln (Details etwa unter <http://tutego.de/go/webtoolscommunity>). Die *JBoss Tools* (<http://tools.jboss.org/>) bauen auch wieder darauf auf und haben JBoss-Produkte im Fokus, etwa Hibernate, aber auch jBPM oder JMX. Oder es erweitert die *Spring Tool Suite* (<https://spring.io/tools>) Eclipse um Lösungen zum Spring Framework bzw. Spring Boot.

## 1.11 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 intern für viele Lösungen, etwa *Google Mail* oder auch *Google+*. Das Auktionshaus eBay nutzt Oracle-Hardware und realisiert seine Geschäftslogik in Java. Selbst 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 verwaltet damit weltweit über 175 Millionen Mitglieder, die viele Millionen Nachrichten am Tag schicken.<sup>53</sup>

---

<sup>53</sup> [https://www.youtube.com/watch?v=qPscn\\_HWMVM](https://www.youtube.com/watch?v=qPscn_HWMVM)

Eclipse und IntelliJ sind in der Regel die Standardwerkzeuge der Softwareentwickler.<sup>54</sup> Entwickler, die Eclipse einsetzen, können in der Hilfe unter <http://tutego.de/go/eclipsehelp> viel Interessantes erfahren und sollten diverse Plugins als Ergänzung evaluieren. Die IDE mit ihren Tastenkürzeln zu beherrschen macht einen guten Softwareentwickler aus.

---

<sup>54</sup> Manche sagen, nur weil es frei ist – sonst würden sie viel lieber IntelliJ nutzen ...

# Kapitel 2

## Imperative Sprachkonzepte

»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: 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 Zeichen 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.

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

### Whitespace

Der Compiler muss die Tokens voneinander unterscheiden können. Daher fügen wir *Trennzeichen* ein; zu diesen zählt Weißraum (engl. *whitespaces*) wie Leerzeichen, Tabulatoren, Zeilenvorschub- und Seitenvorschubzeichen. Außer als Trennzeichen haben diese Zeichen keine Bedeutung. Daher können sie in beliebiger Anzahl zwischen die Tokens gesetzt werden – 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.

### Separatoren

Neben den Trennern gibt es noch zwölf aus ASCII-Zeichen geformte Tokens, die als *Separator* definiert werden:

( ) { } [ ] ; , . . . @ ::

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

```
interface ${static long __(long ___,long ___)
{return ___==0?___+1:___==0?__(_-_1,1):__(_-_1,
____,_-_1);}static void main(String[] _){int
_ =2,___=2;System.out.print("a("+_+','+___+
")="+___(_,_));System.exit(1);}}// (c)CHRIS
```

### 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-Zeichen<sup>1</sup> 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](#), »Arrays und ihre Anwendungen«.

### 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

---

<sup>1</sup> <https://en.wikipedia.org/wiki/ASCII>

den 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.<sup>2</sup> 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



Im folgenden Java-Programm sind die Bezeichner fett gesetzt:

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

Dass `String` fett 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*,<sup>3</sup> 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 oder arabische Buchstaben. Auch wenn damit viele wilde Zeichen als Bezeichnerbuchstaben 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.

<sup>2</sup> Die Java-Methoden `Character.isJavaIdentifierStart(...)/isJavaIdentifierPart(...)` stellen auch fest, ob Zeichen Java-Identifier sind.

<sup>3</sup> Ob ein Zeichen ein Buchstabe ist, stellt die statische Methode `Character.isLetter()` fest; ob er ein gültiger Bezeichnerbuchstabe ist, sagen die Funktionen `isJavaIdentifierStart()` für den Startbuchstaben und `isJavaIdentifierPart()` für den Rest.

**Tabelle 2.1** listet einige gültige Bezeichner auf.

Gültige Bezeichner	Grund
Mami	Mami besteht nur aus Alphazeichen und ist daher korrekt.
__RAPHAEL IST LIEB__	Unterstriche sind erlaubt.
Bóoléáñ	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

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.
_	Ein einzelner Unterstrich gilt ab Java 9 als reserviertes Schlüsselwort.

**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 »openFileReadOnly«. Sprachwissenschaftler nennen einen Großbuchstaben inmitten von Wörtern *Binnenmajuskel*. Programmierer und IT-affine Personen hingegen sprechen gern von der *CamelCase-Schreibweise*, wegen der zweihöckrigen Kamele. Schwierig wird die gemischte Groß-/Kleinschreibung bei großgeschriebenen Abkürzungen, wie HTTP, URL; hier sind die Java-Bibliotheken nicht einheitlich, Klassennamen wie `HttpConnection` oder `HTTPConnection` sind akzeptabel.

### 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`
- ▶ Fließkommaliterale, etwa `12.567` oder `9.999E-2`
- ▶ Zeichenliterale, etwa `'X'` oder `'\n'`
- ▶ String-Literale für Zeichenketten, wie `"Paolo Pinkas"`
- ▶ `null` steht für einen besonderen Referenztyp.

[zB]

#### Beispiel

Im folgenden Java-Programm sind die drei Literale fett gesetzt:

```
class Application {
    public static void main( String[] args ) {
        System.out.println( "Hallo Welt" );
        System.out.println( 1 + 2.65 );
    }
}
```

[zB]

### 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, und es können vom Programmierer keine eigenen Schlüsselwörter hinzugefügt werden.

#### Beispiel

Schlüsselwörter sind im Folgenden fett gesetzt:

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

## Schlüsselwörter und Literale in Java

Nachfolgende Zeichenfolgen sind Schlüsselwörter (bzw. Literale im Fall von true, false und null) und sind in Java daher nicht als Bezeichnernamen möglich:<sup>4</sup>

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. Diese Schlüsselwörter nennen wir *reservierte Schlüsselwörter*, weil sie für eine zukünftige Nutzung reserviert sind. Allerdings ist nicht abzusehen, dass goto jemals verwendet werden wird.

### 2.1.6 Zusammenfassung der lexikalischen Analyse

Übersetzt der Compiler Java-Programme, so beginnt er mit der lexikalischen Untersuchung des Quellcodes. Wir haben die zentralen Elemente schon kennengelernt – sie 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";
        System.out.println( text );
        System.out.println( 1 + 2.65 );
```

<sup>4</sup> Auch nutzt Java ab Version 10 zur Deklaration von Variablen var, doch ist das kein reserviertes Schlüsselwort, sondern lediglich ein Bezeichner mit besonderer Bedeutung. Auch der Unterstrich \_ zählt als reserviertes Schlüsselwort, ist aber in der Tabelle nicht aufgeführt. Die JLS definiert das unter <https://docs.oracle.com/javase/specs/jls/se11/html/jls-3.html#jls-3.9>.

```

    }
}

```

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, String, args, text, System, out, println	Namen für Klasse, Variable, Methode ...
Schlüsselwort	class, public, static, void	reservierte Wörter
Literal	"Hallo Welt", 1, 2.65	konstante Werte, wie Strings, Zahlen ...
Operator	=, +	Operator für Zuweisungen, Berechnungen ...
Separator	(, ), {, }, ;	Symbole, die neben dem Trennzeichen die Tokens trennen

Tabelle 2.4 Tokens 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 Programmelemente wie Klassen, Methoden und Variablen. Ein selbsterklärender Klassenname 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 Schrägstrichen<sup>5</sup> // und kommentieren den Rest einer Zeile aus. Der Kommentar gilt von diesen Zeichen an bis zum Ende der Zeile, also bis zum Zeilenumbruchzeichen.

---

<sup>5</sup> In C++ haben die Entwickler übrigens das Zeilenkommentarzeichen // aus der Vor-Vorgängersprache BCPL wieder eingeführt, das in C entfernt wurde.

- ▶ **Blockkommentare:** Sie kommentieren in `/** */` Abschnitte aus. Der Text im Blockkommentar darf selbst kein `*/` enthalten, denn Blockkommentare dürfen nicht verschachtelt sein.
- ▶ **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 */ ) {
        System.out.println( "Ich habe /*richtig*/ viel //Hunger//" );
    }
}
```

Für den Compiler ist ein Kommentar ein Token, weshalb `1/*2*/3` nicht das Token 13 gibt, sondern die beiden Tokens 1 und 3. Doch vereinfacht gesprochen sieht für den Compiler eine Datei mit Kommentaren genauso aus wie ohne, also wie `class DoYouHaveAnyCommentsToMake { public static void main( String[] args ) { System.out.println( "Ich habe /*richtig*/ viel //Hunger//" ); } }`. Die Ausgabe zeigt, dass es innerhalb von String-Literalen keine Kommentare geben kann; die Symbole `/*`, `*/` und `//` gehören zum String.

Im Bytecode steht exakt das Gleiche – alle Kommentare werden vom Compiler verworfen, kein Kommentar kommt in den Bytecode.

### Kommentare mit Stil

Alle Kommentare und Bemerkungen sollten in Englisch verfasst werden, um Projektmitgliedern aus anderen Ländern das Lesen zu erleichtern. Die Javadoc-Kommentare dokumentieren im Allgemeinen das »Was« und die Blockkommentare das »Wie«.

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+/` kommentiert eine Zeile bzw. einen Block in IntelliJ ein und aus. Achtung, der / muss über die Zehnertastatur ausgewählt werden.



## 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*) vorgibt. 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

Die imperative Befehlsform ist für Programmiersprachen gar nicht selbstverständlich, und es gibt andere Programmierparadigmen. Eine entgegengesetzte Philosophie verfolgenden deklarative Programmiersprachen, bei denen die Logik im Vordergrund steht und kein Programmablauf formuliert wird. Bekannte Vertreter sind SQL, reguläre Ausdrücke und allgemein funktionale sowie logische Programmiersprachen. Ein Vertreter der letzten Gattung ist Prolog, bei der das System zu einer Problembeschreibung selbstständig eine Lösung findet. Die Herausforderung besteht darin, die Aufgabe so präzise zu beschreiben, dass das System eine Lösung

finden kann. Bei der Datenbanksprache SQL müssen wir beschreiben, wie unser Ergebnis aussehen soll, und dann kann das Datenbankmanagement-System anfangen zu arbeiten, doch die internen Abläufe kontrollieren wir nicht.

## 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** src/main/java/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 Zeilenkommentar. 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.

Java ist eine objektorientierte Programmiersprache, die Programmanweisungen 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 Klassename ist ein Bezeichner und darf grundsätzlich beliebig sein, doch besteht die Einschränkung, dass in einer mit `public` deklarierten Klasse der Klassename so lauten muss wie der Dateiname.

### Namenskonvention

Alle Schlüsselwörter in Java beginnen mit Kleinbuchstaben, und Klassennamen beginnen üblicherweise mit Großbuchstaben. Methoden (wie `main`) sind kleingeschrieben, anders als in C#, wo sie großgeschrieben werden.

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(String[])`

Eine besondere Methode ist `public static void main(String[] args) { }`. Die Schlüsselwörter davor und die Angabe in dem Paar runder Klammern hinter dem Namen müssen wir einhalten. Die Methode `main(String[])` ist für die Laufzeitumgebung etwas ganz Besonderes, denn beim Aufruf des Java-Interpreters mit einem Klassennamen wird unsere Methode als Erstes ausgeführt.<sup>6</sup> 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(String[])` befindet sich ein Parameter mit dem Namen `args`. Der Name ist willkürlich gewählt, wir werden allerdings immer `args` verwenden.

Dass Fehler unterkringelt 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. Den nächsten Fehler springt die Tastenkombination `Strg+.` (Punkt) an.

`F2` springt bei IntelliJ zum Fehler, `↑+F2` geht rückwärts zum vorangehenden Fehler.



#### 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, Webservices aufzurufen und vieles mehr. Am untersten Ende der Klassenbibliothek stehen Methoden, die eine gewünschte Operation ausführen.

---

<sup>6</sup> 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.

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.

#### Bemerkung: Warum heißt es nicht einfach `print(...)` für eine Konsolenausgabe?

Anders als viele Skriptsprachen hat Java keine eingebauten Methoden, die einfach so »da« sind. Jede Methode »gehört« immer zu einem Typ, in unserem Fall gehört `println(...)` zu `out`. Auch `out` gehört jemandem, und zwar der Klasse `System`. Erst durch die vollständige Schreibweise ist dem Java-Compiler und der Laufzeitumgebung klar, wer die Ausgabe übernimmt.

Implementieren<sup>7</sup> wir damit eine vollständige Java-Klasse mit einem Methodenaufruf, die über `println(...)` etwas auf dem Bildschirm ausgibt:

**Listing 2.2** src/main/java/Application2.java

```
class Application2 {

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

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

        // Ende des Programms
    }
}
```

#### 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 Variablendeclarationen (die später vorgestellt werden)

<sup>7</sup> »Implementieren« stammt vom lateinischen Wort »implere« ab, das für »erfüllen« und »ergänzen« steht.

nennen sich *atomare* (auch *elementare*) Anweisungen. Diese unteilbaren Anweisungen werden zu *Anweisungssequenzen* zusammengesetzt, die Programme bilden.

### Beispiel

[zB]

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 »gedruckt« wird, also auf dem Bildschirm geschrieben wird. Die Endung `ln` (kurz für *line*) bedeutet, dass noch ein Zeilenvorschubzeichen ausgegeben wird. Umgangssprachlich heißt das: Eine neue Ausgabe beginnt in der nächsten Zeile. Neben `println(...)` existiert die Bibliotheksmethode `print(...)`, die keinen Zeilenvorschub anhängt.

Die `printXXX(...)`-Methoden<sup>8</sup> 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.

---

<sup>8</sup> Abkürzung für Methoden, die mit `print` beginnen, also `print(...)` und `println(...)`

## 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** src/main/java/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
```

In der letzten Zeile ist gut zu sehen, dass es Probleme mit der Genauigkeit gibt – dieses Phänomen werden wir uns noch genauer in [Kapitel 22](#) anschauen.



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

## Variable Argumentlisten

Java unterstützt 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** src/main/java/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" );
    }
}
```

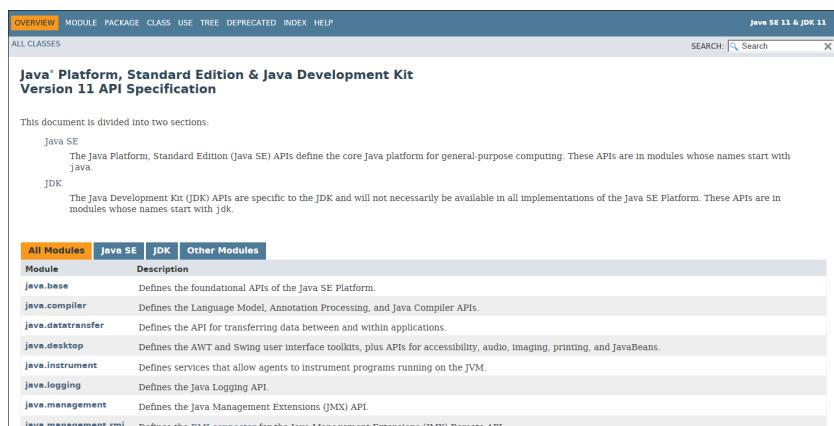
Die Ausgabe der Anweisung ist:

Was sagst du?  
220 Kanäle und überall nur Katzen.

Die Formatierungsanweisung `%n` setzt einen Zeilenumbruch, `%d` ist ein Platzhalter für eine ganze Zahl (`%f` wäre für eine Fließkommazahl) und `%s` ein Platzhalter für eine Zeichenkette oder etwas, was in einen String konvertiert werden soll. Weitere Platzhalter werden in [Abschnitt 5.4, »Die Klasse String und ihre Methoden«](#), 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 des JRE noch des JDK – dafür ist die Hilfe zu groß. Wer über eine 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  $\text{Strg} + \text{F2}$  in einem eingebetteten Browser-Fenster die API-Dokumentation an, wobei die Javadoc von den Oracle-Seiten kommt. Mithilfe der  $\text{F2}$ -Taste bekommen wir ein kleines gelbes Vorschaufenster, das ebenfalls die API-Dokumentation zeigt.

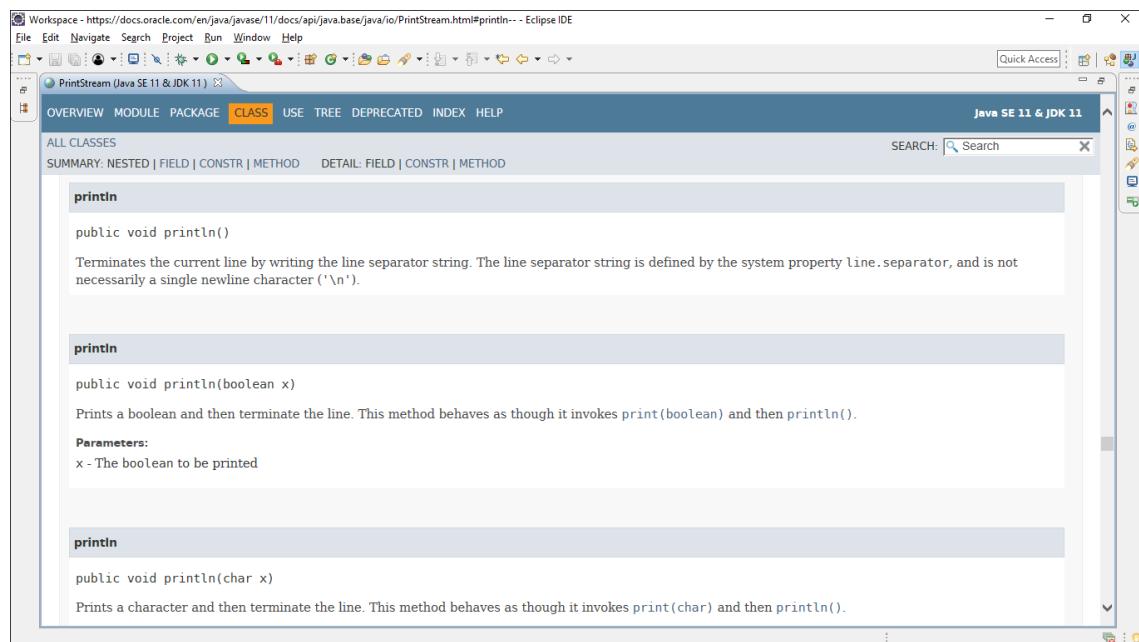


Abbildung 2.2 API-Dokumentation in Eclipse



IntelliJ zeigt mithilfe der Tasten  $\text{Strg} + \text{Q}$  die API-Dokumentation in einem kleinen Fenster an.

### 2.2.8 Ausdrücke

Ein *Ausdruck* (engl. *expression*) ergibt bei der Auswertung ein Ergebnis. Im Beispiel *OverloadedPrintln.java* ([Listing 2.3](#)) 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. In 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 Objekterzeugung).

## 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, kann also 0 werden) und 1 (exklusiv, erreicht 1 also nie wirklich). 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.<sup>9</sup>

### Hinweis

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



<sup>9</sup> 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 einen Zustand, zum Beispiel die Lautstärke oder das Baujahr. Wichtig ist in objektorientierten Sprachen, dass die Operationen und Zustände immer (und da gibt es keine Ausnahmen) an Objekte bzw. Klassen gebunden sind (mehr zu dieser Unterscheidung folgt in [Kapitel 3](#), »Klassen und Objekte«, und in [Kapitel 6](#), »Eigene Klassen schreiben«). Der Aufruf einer Methode auf einem Objekt richtet die Anfrage genau an dieses bestimmte 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`.

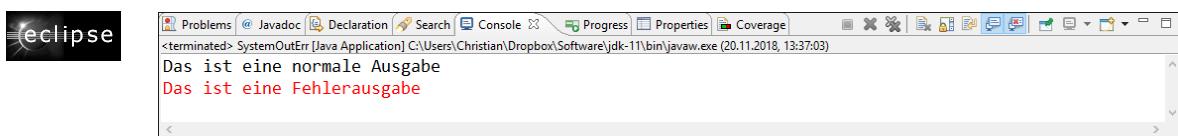
#### `System.out` und `System.err`

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:

**Listing 2.5** src/main/java/SystemOutErr.java, main()

```
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 `printXXX(...)` 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 2.3** 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

Im folgenden Programm kommen drei Modifizierer vor; sie sind fett gesetzt:

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

[zB]

Der Modifizierer `public` ist ein *Sichtbarkeitsmodifizierer*. Er bestimmt, ob die Klasse bzw. 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«, nichtstatische Methoden einführen.

### 2.2.12 Gruppieren von Anweisungen mit Blöcken

Ein *Block* fasst eine Gruppe von Anweisungen zusammen, die hintereinander ausgeführt werden. Ein Block ist eine Anweisung, die in geschweiften Klammern `{ }` eine Folge von Anweisungen zu einer neuen Anweisung zusammenfasst. Anders gesagt: Die Folge von Anweisungen wird dadurch, dass sie in geschweifte Klammern gesetzt und damit zu einem Block zusammengefasst wird, zu einer neuen Anweisung:

```
{
    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.

### Code mit Stil

Die Zeilen, die in geschweiften Klammern stehen, werden in der Regel mit Leerraum eingerückt. Üblicherweise sind es zwei (wie in diesem Buch) oder vier Leerzeichen. Viele Autoren setzen die geschweiften Klammern in eine eigene Zeile. Es gibt eine Fülle von weiteren Ratschlägen zum Code-Style, die das einfache Lesen und das Verständnis für den Menschen in den Mittelpunkt rücken: Zeilen sollen nicht zu lang sein (80 bis 100 Zeichen sind eine gute Größe), tiefe Schachtelungen sind zu vermeiden, Sinneinheiten werden mit Leerzeilen geschaffen. Der »Google Java Style Guide« unter <https://google.github.io/styleguide/javaguide.html> gibt gute Vorschläge.

### 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. In [Abschnitt 2.5](#), »Bedingte Anweisungen oder Fallunterscheidungen«, und in [Abschnitt 2.6](#), »Immer das Gleiche mit den Schleifen«, 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 In-

halt – 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 *stark typisierte* Programmiersprache.<sup>10</sup>

### 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.



### Primitiv- oder Verweistyp

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, Datenstrukturen oder Zwergpinscher verwalten.

Warum sich damals Sun für diese Teilung entschied, lässt sich mit einem einfachen Grund erklären: Java wurde als Programmiersprache entworfen, die kleine, schwache Geräte unterstützen sollte, und auf denen musste die Java-Software, die am Anfang noch interpretiert wurde, so schnell wie möglich laufen. Unterscheidet der Compiler zwischen primitiven Typen und Referenztypen, so kann er relativ leicht Bytecode erzeugen, der ebenfalls zwischen den beiden Typen unterscheidet. Damit kann die Laufzeitumgebung auch den Programmcode viel schneller ausführen, und das mit einem relativ einfachen Compiler. Das war für die Anfangszeit ein wichtiges Kriterium.

---

<sup>10</sup> 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 *stark (stark) typisiert* und *schwach typisiert*.

### Sprachvergleich mit Smalltalk und .NET \*

In Smalltalk ist alles ein Objekt, auch die eingebauten Sprachdatentypen. Für Zahlen gibt es einen Basistyp `Number` und `Integer`, `Float`, `Fraction` als Untertypen. Immer noch gibt es arithmetische Operatoren (`+`, `-`, `*`, `/`, `//`, `\`, um sie alle aufzuzählen), aber das sind nur Methoden der Klasse `Number`.<sup>11</sup> Für Java-Entwickler sind Methodennamen wie `+` oder `-` ungewöhnlich, doch in Smalltalk sind sie das nicht. Syntaktisch unterscheidet sich ein `1 + 2` in Java und Smalltalk nicht, nur in Smalltalk ist die Addition ein Nachrichtenaufruf an das `Integer`-Objekt `1` und an die Methode `+` mit dem Argument `2`, das wiederum ein `Integer`-Objekt ist – die Objekte baut der Compiler selbstständig aus den Literalen auf. Eine Klasse `Integer` für Ganzzahlen besitzt weitere Methoden wie `asCharacter` und `floor`.<sup>12</sup> Es ist wichtig zu verstehen, dass dies nur das semantische Modell auf der Sprachseite ist; das hat nichts damit zu tun, wie später die Laufzeitumgebung diese speziellen Nachrichtenaufrufe optimiert. Moderne Smalltalk-Laufzeitumgebungen mit Just-in-time-Compilation sind bei arithmetischen Operationen auf einem ähnlichen Level wie C oder Java. Durch die Einteilung von Java in primitive Datentypen und Referenztypen haben die Sprachschöpfer einen objektorientierten Bruch in Kauf genommen, um die interpretierte Laufzeit Anfang der 1990er zu optimieren – eine Optimierung, die aus heutiger Sicht unnötig war.

In .NET ist es eher wie in Java. Der Compiler kennt die eingebauten Datentypen und gibt ihnen eine Sonderbehandlung, es sind keine Methodenaufrufe. Auch im Bytecode (Common Intermediate Language, kurz CIL in .NET genannt) finden sich Anweisungen wie Addition und Subtraktion wieder. Doch es gibt noch einen Unterschied zu Java: Der Compiler bildet Datentypen der .NET-Sprachen auf .NET-Klassen ab, und diese Klassen haben Methoden. In C# ist der eingebaute Datentyp `float` mit dem Datentyp `Single` (aus dem .NET-Paket System) identisch, und es ist egal, ob Entwickler `float f` oder `Single f` schreiben. Doch `Single` (respektive `float`) hat im Vergleich zu Smalltalk keine mathematischen Operationen, aber dennoch ein paar wenige Methoden wie `ToString()`.<sup>13</sup> In .NET verhalten sich folglich die eingebauten Datentypen wie Objekte, sie haben Methoden, haben aber die gleiche Wertsemantik, zum Beispiel bei Methodenaufrufen, wie in Java und sehen auch im Bytecode ähnlich aus, was ihnen die gleiche gute Performance verleiht.

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.

<sup>11</sup> Die Dokumentation für das GNU Smalltalk zeigt auf: [http://www.gnu.org/software/smalltalk/manual-base/html\\_node/Number\\_002darithmetic.html#Number\\_002darithmetic](http://www.gnu.org/software/smalltalk/manual-base/html_node/Number_002darithmetic.html#Number_002darithmetic).

<sup>12</sup> [http://www.gnu.org/software/smalltalk/manual-base/html\\_node/Integer.html](http://www.gnu.org/software/smalltalk/manual-base/html_node/Integer.html)

<sup>13</sup> <https://msdn.microsoft.com/en-us/library/system.int32>

### 2.3.1 Primitive Datentypen im Überblick

In Java gibt es zwei Arten eingebauter primitiver Datentypen:

- ▶ *arithmetische Typen* (ganze Zahlen – auch integrale Typen genannt –, Fließkommazahlen, 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)
byte	$-2^7$ bis $2^7 - 1$ ( $-128 \dots 127$ )
short	$-2^{15}$ bis $2^{15} - 1$ ( $-32.768 \dots 32.767$ )
int	$-2^{31}$ bis $2^{31} - 1$ ( $-2.147.483.648 \dots 2.147.483.647$ )
long	$-2^{63}$ bis $2^{63} - 1$ ( $-9.223.372.036.854.775.808 \dots 9.223.372.036.854.775.807$ )
float	1,40239846E-45f ... 3,40282347E+38f
double	4,94065645841246544E-324 ... 1,79769131486231570E+308

Tabelle 2.5 Java-Datentypen und ihre Wertebereiche

Bei den Ganzzahlen fällt auf, dass es eine positive Zahl »weniger« gibt als negative, das liegt an der Kodierung im Zweierkomplement.

Für float und double ist das Vorzeichen nicht angegeben, da die kleinsten und größten darstellbaren Zahlen sowohl positiv als auch negativ sein können. Mit anderen Worten: Die Wertebereiche unterscheiden sich nicht – anders als etwa bei int – in Abhängigkeit vom Vorzeichen.<sup>14</sup>

#### 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 wieder abgebildet.

<sup>14</sup> Es gibt bei Fließkommazahlen noch »Sonderzahlen«, wie plus oder minus unendlich, aber dazu in Kapitel 22, »Bits und Bytes, Mathematisches und Geld« später mehr.



Tabelle 2.6 zeigt eine etwas andere Darstellung.

Typ	Größe	Format
<b>Ganzzahlen</b>		
byte	8 Bit	Zweierkomplement
short	16 Bit	Zweierkomplement
int	32 Bit	Zweierkomplement
long	64 Bit	Zweierkomplement
<b>Fließkommazahlen</b>		
float	32 Bit	IEEE 754
double	64 Bit	IEEE 754
<b>Weitere Datentypen</b>		
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 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
```

Es gibt für jeden primitiven Datentyp eine eigene Klasse mit Hilfsmethoden rund um diesen Datentyp. Mehr zu diesen besonderen Klassen folgt in [Kapitel 10, »Besondere Typen der Java SE«](#).



## 2.3.2 Variablendeklarationen

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Um Variablen zu nutzen, müssen sie deklariert (definiert<sup>15</sup>) werden. Die Schreibweise einer Variablendeklaration ist immer die gleiche: Hinter dem Typnamen folgt

<sup>15</sup> 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*.

der Name der Variablen. Die Deklaration 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 2.6** src/main/java/FirstVariable.java

```
public class FirstVariable {

    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.<sup>16</sup> 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 vorangehenden Listing 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:

---

<sup>16</sup> 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 2.7** src/main/java/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 = false;
    }
}
```

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 = 165 /* cm */;
```

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

### Zinsen berechnen als Beispiel zur Variablen-deklaration, -initialisierung und -ausgabe

Zusammen mit der Konsolenausgabe 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 einem Jahr sind.

**Listing 2.8** src/main/java/InterestRates.java

```
public class InterestRates {
    public static void main( String[] args ) {
        double capital = 20000 /* Euro */;
        double interestRate = 3.6 /* Prozent */;
        double totalInterestRate = capital * interestRate / 100; // Jahr 1
        System.out.print( "Zinsen: " );
        System.out.println( totalInterestRate ); // 720.0
    }
}
```

**Tipp**

Strings können mit einem Plus aneinandergehängt werden; ist ein Segment kein String, so wird es in einen String konvertiert und dann angehängt.

```
System.out.println( "Zinsen: " + totalInterestRate ); // Zinsen: 720.0
```

Mehr Beispiele dazu in [Abschnitt 2.4.11, »Überladenes Plus für Strings«](#).

### 2.3.3 Automatisches Feststellen der Typen mit var

Java 10 hat die Erweiterung gebracht, dass der VariablenTyp bei gewissen Deklarationen entfallen kann und wir einfach stattdessen var nutzen können:

**Listing 2.9** src/main/java/VarObama.java, Ausschnitt

```
var name = "Barack Hussein Obama II";
var age = 48;
var income = 400000;
var gender = 'm';
var isPresident = false;
```

Wir sehen, dass im Gegensatz zu unserem vorherigen Beispiel nicht mehr die VariablenTypen wie String oder int bei der VariablenDeklaration explizit im Code stehen, sondern nur noch var. Das heißt allerdings nicht, dass der Compiler die Typen offenlässt! Der Compiler braucht zwingend die rechte Seite neben dem Gleichheitszeichen, um den Typ feststellen zu können, das nennt sich *Local-Variable Type Inference*. Daher gibt es in unserem Programm auch eine Unstimmigkeit, nämlich bei var income = 400000, die gut ein Problem mit var aufzeigt: Die Variable ist kein double mehr wie vorher, sondern 400000 ist ein Ganzzahl-Literal, weshalb der Java-Compiler der Variablen income den Typ int gibt.

Die Nutzung von var soll Entwicklern helfen, Code kürzer zu schreiben, insbesondere wenn der Variablenname schon eindeutig auf den Typ hinweist. Finden wir eine Variable text vor, ist der Typ String naheliegend, genauso wie age ein int ist oder ein Präfix wie is oder has auf eine boolean-Variable hinweist. Aber wenn var auf die Kosten der Verständlichkeit geht, darf die Abkürzung nicht eingesetzt werden. Auch der Java-Compiler gibt Schranken vor:

- ▶ var ist nur dann möglich, wenn eine Initialisierung einen Typ vorgibt. Eine Deklaration der Art var age; ohne Initialisierung ist nicht möglich und führt zu einem Compilerfehler.
- ▶ var kann nur bei lokalen Variablen eingesetzt werden, wo der Bereich überschaubar ist. Es gibt aber noch viele weitere Stellen, wo in Java Variablen deklariert werden – dort ist var nicht möglich.

### Sprachvergleich

Java ist mit var relativ spät dran.<sup>17</sup> Andere statisch getypte Sprachen bieten die Möglichkeit schon länger, etwa C++ mit auto oder C# auch mit var. Auch JavaScript nutzt var, allerdings in einem völlig anderen Kontext: In JavaScript sind Variablen erst zur Laufzeit getypt, und alle Operationen werden erst zur Ausführungszeit geprüft, während Java die Typsicherheit mit var nicht aufgibt.

#### 2.3.4 Konsoleneingaben

Bisher haben wir Methoden zur Ausgabe kennengelernt und random(). Die println(...)-Methoden »hängen« am System.out- bzw. System.err-Objekt, und random() »hängt« am Math-Objekt.

Der Gegenpol zu printXXX(...) ist eine Konsoleneingabe. Hier gibt es unterschiedliche Varianten. Die einfachste ist die mit der Klasse java.util.Scanner. In [Abschnitt 5.9.2](#), »Yes we can, yes we scan – die Klasse Scanner«, 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	String s = new java.util.Scanner(System.in).nextLine();
int	int i = new java.util.Scanner(System.in).nextInt();
double	double d = new java.util.Scanner(System.in).nextDouble();

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

Verbinden wir die drei Möglichkeiten zu einem Beispiel. Zunächst soll der Name eingelesen werden, dann das Alter und anschließend eine Fließkommazahl.

**Listing 2.10** src/main/java/SmallConversation.java

```
public class SmallConversation {

    public static void main( String[] args ) {
        System.out.println( "Moin! Wie heißt 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 %.%n",
                           age, age * 2 );
    }
}
```

<sup>17</sup> <http://openjdk.java.net/jeps/286>

```

        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 des Platzhalters `%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. Zinseszinsen berücksichtigt das Programm nicht.

**Listing 2.11** src/main/java/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 totalInterestRate = capital * interestRate * month / (12*100);
    }
}

```

```

        System.out.printf( "Zinsen: %g%n", totalInterestRate );
    }
}

```

Die vorher fest verdrahteten Werte sind nun alle dynamisch:

```

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

```

Um den Zinseszins berücksichtigen zu können, muss eine Potenz mit in die Formel gebracht werden. Die nötige Methode dazu ist `Math.pow(a, b)`, was  $a$  hoch  $b$  berechnet. Finanzmathematikern ist das als Übung überlassen.

### Dialogeingabe



Soll die Eingabe nicht von der Konsole kommen, sondern von einem eigenen Dialog, hilft eine Klasse aus dem Swing-Paket:

```
String input = javax.swing.JOptionPane.showInputDialog( "Eingabe" );
```

### 2.3.5 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;
```

Nur den Punkt allein zu nutzen ist natürlich Unsinn, wobei `.0` schon erlaubt ist.

### Hinweis



Der Datentyp `float` ist mit 4 Byte, also 32 Bit, ein schlechter Scherz. Der Datentyp `double` geht mit 64 Bit ja gerade noch. Die IA32-, x86-64- und Itanium-Prozessoren unterstützen mit 80 Bit einen »double extended«-Modus und damit bessere Präzision.



### Hinweis

Der Compiler meldet keinen Fehler, wenn eine Fließkommazahl nicht präzise dargestellt werden kann. Es ist kein Fehler, zu schreiben: double pi = 3.141592653589793238462643383279502884197169399375105820974944592;

### Der Datentyp float \*

Standardmäßig sind die Fließkommaliterale vom Typ `double`. Ein nachgestelltes `f` (oder `F`) zeigt dem Compiler an, dass es sich um ein `float` handelt.



### Beispiel

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 das Ereignis in `double` konvertiert wird und anschließend `4.0` addiert wird, würde `1D + 2 + 4.0` gleich mit der Fließkommazahl `1` beginnen. So ist auch `1D` gleich `1.` bzw. `1.0`.<sup>18</sup>

### 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`?<sup>19</sup>

### Noch genauere Auflösung bei Fließkommazahlen \*

Einen höher auflösenden bzw. 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 22, »Bits und Bytes, Mathematisches und Geld«](#), näher beschrieben ist. Das ist

<sup>18</sup> Ein Literal wie `1D` macht deutlich, warum Bezeichner nichts mit einer Ziffer anfangen können: Wenn eine Variablenklärung wie `double 1D = 2;` erlaubt wäre, dann wüsste der Compiler bei `println(1D)` ja gar nicht, ob `1D` für das Literal steht oder für die Variable.

<sup>19</sup> Prüfe zum Verständnis die Ausgabe von: `System.out.println( Float.toHexString( 20000000000F ) );
System.out.println( Float.toHexString( 20000000000F + 1F ) );
System.out.println( Double.toHexString( 20000000000D ) );
System.out.println( Double.toHexString( 20000000000D + 1D ) );`

sinnvoll für Daten, die eine sehr gute Genauigkeit aufweisen sollen, wie zum Beispiel Währungen.<sup>20</sup>

### 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.

## 2.3.6 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.<sup>21</sup> 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 Arrays 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 gültigen `int`-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 2 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 );
```

<sup>20</sup> 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.

<sup>21</sup> 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 einer expliziten Typumwandlung lassen sich doch Zahlen wie 200 in einem byte speichern.

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

Der Java-Compiler nimmt dazu einfach die Bitbelegung von 200 ( $0b00000000_00000000_00000000\_11001000$ ), schneidet bei der Typumwandlung die oberen drei Byte ab und interpretiert das oberste dann gesetzte Bit als Vorzeichen-Bit. Bei der Ausgabe fällt das auf:

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

Mehr zur Typumwandlung folgt an anderer Stelle: in [Abschnitt 2.4.10](#), »Die Typumwandlung (das Casting)«.

## 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
```

### 2.3.7 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.

### 2.3.8 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 1.000, 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 86.400.000 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 macht durch Unterstriche Zahlen besser lesbar, denn der Unterstrich gliedert die Zahl in Blöcke. Anstatt ein numerisches Literal als 86.400.000 zu schreiben, ist auch Folgendes erlaubt:

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

Die Unterstriche machen die 1.000er-Blöcke gut sichtbar.<sup>22</sup>

#### Beispiel

Hilfreich ist die Schreibweise auch bei Literalen in Binär- und Hexadezimaldarstellung, da die Unterstriche hier ebenfalls Blöcke absetzen können:



<sup>22</sup> Bei Umrechnungen zwischen Stunden, Minuten usw. hilft auch die Klasse `TimeUnit` mit einigen statischen `toXXX()`-Methoden.

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

Mit 0b beginnt ein Literal in Binärschreibweise, mit 0x in Hexadezimalschreibweise (weitere Details in [Kapitel 22, »Bits und Bytes, Mathematisches und Geld«](#)).

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



### Hinweis

Die Unterstriche in Literalen sind nur eine Hilfe wie Leerzeichen zur Einrückung. Im Bytecode ist davon nichts mehr zu lesen. In der Klassendatei sehen 0b01101001\_01001101\_11100101\_01011110 und 0b0110100101001101110010101011110 identisch aus, insbesondere weil sie sowieso als Ganzzahl 1766712670 abgelegt sind.

## 2.3.9 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. Spracheinsteiger 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

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.

## 2.3.10 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 sonst 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 verpfuschten 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.
- ▶ **O 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.<sup>23</sup> Gültig – aber böse – ist auch: int int, int, int; 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]+[Shift]+[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.

### 2.3.11 Initialisierung von lokalen Variablen

Die Laufzeitumgebung – bzw. 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.<sup>24</sup>

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.

<sup>23</sup> Eine Software wie Mathematica warnt vor Variablen mit fast identischem Namen.

<sup>24</sup> Anders ist das bei Objektvariablen (und statischen Variablen sowie Feldern). Sie sind standardmäßig mit null (Referenzen), 0 (bei Zahlen) oder false belegt.

```

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.

## 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 zum Beispiel eine Variable, ein Literal oder Rückgabe eines Methodenaufrufs. Im Fall einer Variablen wird der Wert aus der Variablen ausgelesen und mit ihm die Berechnung durchgeführt.

### Die Arten von Operatoren

Die Operatoren verknüpfen die Operanden. Je nach Anzahl der Operanden unterscheiden wir folgende Arten von Operatoren:

- ▶ 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 sowie 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*).<sup>25</sup> Der Zuweisungsoperator ist ein binärer Operator, bei dem auf der linken Seite die zu belegende Variable steht und auf der rechten Seite ein Ausdruck.

#### Beispiel

[zB]

Ein Ausdruck mit Zuweisungen:

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

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.<sup>26</sup> 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.

#### Sprachvergleich

Das einfache Gleichheitszeichen = dient in Java nur der Zuweisung. Das ist in fast allen Programmiersprachen so. Selten verwenden Programmiersprachen für die Zuweisung ein anderes Symbol, etwa Pascal := oder F# und R <- . Um Zuweisungen von Vergleichen trennen zu können, definiert Java hier, der C(++)-Tradition folgend, einen binären Vergleichsoperator == mit zwei Gleichheitszeichen. 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
```

<sup>25</sup> 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.

<sup>26</sup> 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.

## 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 `printXXX(...)`:

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

## 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
```

Im Produktivcode sollte so etwas dennoch nicht stehen.

### 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 umfassende-

ren 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.<sup>27</sup> (Auch im Java-Bytecode gibt es keine arithmetischen Operationen auf `byte`, `short` und `char`.) Anschließend wird die Operation ausgeführt, und der Ergebnistyp entspricht dem umfassenderen 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 Ganzahldivision 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 mindestens 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 Ganzahldivision mit dem Divisor `0` durch, so bestraft uns Java mit einer `ArithmeticException`, die, wenn sie nicht behandelt wird, zum Ende des Programmablaufs führt. Bei Fließkommazahlen liefert eine Division durch `0` keine Ausnahme, sondern `+/-infinity` und bei `0.0/0.0` den Sonderwert `NaN` (mehr dazu folgt in [Kapitel 22](#), »Bits und Bytes, Mathematisches und Geld«). `NaN` steht für *Not a Number* (auch manchmal »Unzahl« genannt) und wird vom Prozessor erzeugt, falls er eine mathematische Operation wie die Division durch null nicht durchführen kann.

#### 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.

<sup>27</sup> <https://docs.oracle.com/javase/specs/jls/se11/html/jls-5.html#jls-5.6.1>

### 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.<sup>28</sup>

```
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:  $(\text{int})(a/b) \times b + (a \% b) = a$ .



#### Beispiel

Die Gleichung ist erfüllt, wenn wir etwa  $a = 10$  und  $b = 3$  wählen. Es gilt:  $(\text{int})(10/3) = 3$  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; das Ergebnis 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 `ArithmeticException` auslöst und bei Fließkommazahlen zum Ergebnis NaN führt.

**Listing 2.12** src/main/java/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
```

---

<sup>28</sup> 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.

Gewöhnungsbedürftig ist die Tatsache, dass der erste Operand (*Dividend*) das Vorzeichen des Restes definiert und niemals der zweite (*Divisor*). In [Kapitel 22](#), »Bits und Bytes, Mathematisches und Geld«, werden wir eine Methode `floorMod(...)` kennenlernen, die etwas anders arbeitet.

### 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()`<sup>\*</sup>

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 rundenenden 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 die statische Bibliotheksmethode `Math.IEEEremainder(...)`<sup>29</sup> 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<sup>\*</sup>

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:

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

<sup>29</sup> 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.

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 nahm Intel die CPUs zurück, verlor über 400 Millionen US-Dollar und zog spät den Kopf gerade noch aus der Schlinge.

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.



#### Beispiel

Drehe das Vorzeichen einer Zahl um:  $a = -a;$

Eine Alternative ist:  $a = -1 * a;$

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



#### 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  $-5$ . Es ist leichter, den Compiler zu verstehen, wenn wir die Operatorrangfolge einbeziehen und gedanklich Klammern setzen:  $-(-(-2)) + (-(+3))$ . Einen Ausdruck wie  $---2+-+3$  erkennt der Compiler dagegen nicht an, da die zusammenhängenden Minuszeichen als Dekrement interpretiert werden und nicht als unärer Operator. Das Trennzeichen, Leerzeichen in diesem Fall, ist also bedeutend.

#### 2.4.4 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 somit zwei Aufgaben: Neben der Wertrückgabe gibt es eine Veränderung der Variablen.

##### 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, beschrieb einmal Java als C<sup>++</sup>. Er meinte damit C++ ohne die schwer zu pflegenden Eigenschaften.



##### 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- bzw. heruntergesetzt und dann der Wert geliefert.

##### Beispiel



Präfix/Postfix in einer Ausgabeanweisung:

Präfix-Inkrement und -Dekrement	Postfix-Inkrement und -Dekrement
<pre>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</pre>	<pre>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</pre>

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 2.8 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:

```
int i = 1;
i = ++i;
System.out.println( i ); // 2
int j = 1;
j = j++;
System.out.println( j ); // 1
```

Der erste Fall überrascht nicht, denn `i = ++i` erhöht den Wert 1 um 1, und anschließend wird 2 der Variablen `i` zugewiesen. Bei `j` ist es raffinierter: Der Wert von `j` ist 1, und dieser Wert wird intern vermerkt. Anschließend erhöht `j++` die Variable um 1. Doch die Zuweisung setzt `j` auf den gemerkten Wert, der 1 war. Also ist `j = 1`.

### Sprachvergleich: Sequenzpunkte in C(++) \*

Je mehr Freiheiten ein Compiler hat, desto ungenauer kann er optimieren. Besonders Schreibzugriffe interessieren einen Compiler, denn kann er diese einsparen, läuft das Programm später ein bisschen schneller. Damit das Resultat eines Compilers jedoch beherrschbar bleibt, definiert der C(++)-Standard *Sequenzpunkte* (engl. *sequence points*), an denen alle Schreibzugriffe klar zugewiesen wurden (dass der Compiler später Optimierungen vornimmt, ist eine

andere Geschichte; die Sequenzpunkte gehören zum semantischen Modell, Optimierungen verändern das nicht). Das Semikolon als Abschluss von Anweisungen bildet zum Beispiel einen Sequenzpunkt. In einem Ausdruck wie `i = i + 1; j = i;` muss der Schreibzugriff auf `i` aufgelöst sein, bevor ein Lesezugriff für die Zuweisung zu `j` erfolgt. Problematisch ist, dass es gar nicht so viele Sequenzpunkte gibt und es passieren kann, dass zwischen zwei Sequenzpunkten zwei mehrdeutige Schreibzugriffe auf die gleiche Variable stattfinden. Da das jedoch in C(++) undefiniert ist, kann sich der Compiler so verhalten, wie er will – er muss sich ja nur an den Sequenzpunkten so verhalten, wie gefordert. Problemfälle sind: `(i=j) + i` oder, weil ein Inkrement/Dekrement ein Lese-/Schreibzugriff ist, auch `i = i++`, was ja nichts anderes als `i = (i = i + 1)` ist. Bedauerlicherweise bildet die Zuweisung keinen Sequenzpunkt. Bei Zuweisungen der Art `i = ++i + --i` kann alles Mögliche später in `i` stehen, je nachdem, was der Compiler zu welchem Zeitpunkt ausführt. In Java sind diese Dinge von der Spezifikation klar geregelt, in C(++) ist nur geregelt, dass das Verhalten zwischen den Sequenzpunkten klar sein muss. Doch moderne Compiler erkennen konkurrierende Schreibzugriffe zwischen zwei Sequenzpunkten und mahnen sie (bei entsprechender Warnstufe) an.<sup>30</sup>

#### 2.4.5 Zuweisung mit Operation (Verbundoperator)

In Java lassen sich Zuweisungen mit numerischen Operatoren kombinieren. Für einen binären Operator (symbolisch # genannt) im Ausdruck `a = a # (b)` kürzt der *Verbundoperator* (engl. *compound assignment operator*) den Ausdruck zu `a #= b` ab.<sup>31</sup>

Dazu einige Beispiele:

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

Tabelle 2.9 Ausgeschriebene Variante und kurze Schreibweise mit dem Verbundoperator

Während das Präfix-/Postfix-Inkrement/-Dekrement nur um eins vergrößert/vermindert, erlauben die Verbundoperationen in einer relativ kompakten Schreibweise auch größere Inkremente/Dekremente, wie eben `a+=2` oder `a-=10;`.

<sup>30</sup> Beim GCC-Compiler ist der Schalter `-Wsequence-point` (der auch bei `-Wall` mitgenommen wird), siehe dazu <http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>.

<sup>31</sup> Die Abkürzung existiert nur im Programmcode, im Bytecode gibt es keinen Verbundoperator.



### Beispiel

Eine Zuweisung ist auch immer ein Ausdruck:

```
int a = 0;
System.out.println( a );           // 0
System.out.println( a = 2 );       // 2
System.out.println( a += 1 );      // 3
System.out.println( a );           // 3
```

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  $a = a * 3 + 5$ .

### Einmalige Auswertung bei Array-Zugriffen \*

Falls die linke Seite beim Verbundoperator ein Array-Zugriff ist (siehe [Abschnitt 4.1, »Arrays«](#)), wird die Indexberechnung nur einmal vorgenommen. Dies ist wichtig beim Einsatz vom Präfix-/Postfix-Operator oder von Methodenaufrufen, die Nebenwirkungen besitzen, also etwa Zustände wie einen Zähler verändern.<sup>32</sup>



### Beispiel

Wir profitieren bei Array-Zugriffen vom Verbundoperator, da erstens die Schreibweise kurz ist und zweitens die Auswertung des Index nur einmal stattfindet:

```
int[] array1 = { 10, 90, 0 };
int i = 0;
array1[++i] = array1[++i] + 10;
System.out.println( Arrays.toString( array1 ) ); // [10, 10, 0]
int[] array2 = { 0, 90, 0 };
int j = 0;
array2[++j] += 10;
System.out.println( Arrays.toString( array2 ) ); // [0, 100, 0]
```

## 2.4.6 Die relationalen Operatoren und die Gleichheitsoperatoren

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

---

<sup>32</sup> Details in der JLS unter <https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.26.2>

- ▶ größer (>)
- ▶ kleiner (<)
- ▶ größer/gleich ( $\geq$ )
- ▶ kleiner/gleich ( $\leq$ )

Außerdem gibt es einen Spezialoperator `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

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.

[zB]

### (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 == 58000000000L )
```

und

```
if ( 58000000000L == worldExpoShanghaiCostInUSD )
```

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 so genannten *Yoda-Stil*<sup>33</sup> ist die Vermeidung von Fehlern. Fehlt in C ein Gleichheitszeichen, so ist `if(worldExpoShanghaiCostInUSD = 58000000000L)` als Zuweisung compilierbar (wenn auch mittlerweile mit einer Warnung), `if(58000000000L = 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.

---

<sup>33</sup> 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 worldExpoShanghai-CostInUSD 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 ...



### Hinweis

Mehr als diese drei logischen Operatoren sind auch nicht nötig, um alle möglichen logischen Verknüpfungen zu realisieren. In der Mathematik wird das *boolesche Algebra* genannt.

Mit logischen Operatoren werden Wahrheitswerte nach definierten Mustern verknüpft. Logische Operatoren operieren nur auf boolean-Typen, andere Typen führen zu Compilerfehlern. Java bietet die Operatoren *Nicht* (!), *Und* (&&), *Oder* (||) und *XOR* (^) an. XOR ist eine Operation, die nur dann wahr liefert, wenn genau einer der beiden Operanden true ist. Sind beide Operanden gleich (also entweder true oder false), so ist das Ergebnis false. XOR heißt auch *exklusives* oder *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?«  $a \wedge b$  ist eine Abkürzung für  $(a \&\& !b) \mid\mid (!a \&\& b)$ .

boolean a	boolean b	$\neg a$	$a \&\& b$	$a \mid\mid b$	$a \wedge b$
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 22.1.1, »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.

## 2.4.8 Kurzschluss-Operatoren

Ein logischer Ausdruck muss nur dann weiter ausgewertet werden, wenn sich das Endergebnis noch ändern kann. Steht das Ergebnis schon vor der Auswertung aller Teile unumstöß-

lich fest, kürzt der Compiler den Programmfluss ab. Die beiden Operatoren `&&` (Und) bzw. `||` (Oder) 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.

Nehmen wir zum Beispiel `true || Math.random() > 0.5`; hier wird es nicht zum Aufruf der Methode kommen, denn die beiden Operatoren `&&` und `||` sind *Kurzschluss-Operatoren* (engl. *short-circuit operators*).<sup>34</sup> Kennzeichnung der Kurzschluss-Operatoren ist also eine Abkürzung, wenn das Ergebnis des Ausdrucks feststeht; die restlichen Ausdrücke werden nicht ausgewertet.

### 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. Das Ergebnis der Auswertung ist das Gleiche wie vorher.

Die Arbeitsweise dokumentiert das folgende Programm, bei dem gut abzulesen ist, dass eine Variable nur dann erhöht wird, wenn der Nicht-Kurzschluss-Operator auswertet:

**Listing 2.13** src/main/java/CircuitNotCircuitOperator.java, main()

```
int a = 0, b = 0, c = 0, d = 0;
System.out.println( true || a++ == 0 ); // true, a wird nicht erhöht
System.out.println( a ); // 0
System.out.println( true | b++ == 0 ); // true, b wird erhöht
System.out.println( b ); // 1
System.out.println( false && c++ == 0 ); // false, c wird nicht erhöht
System.out.println( c ); // 0
System.out.println( false & d++ == 0 ); // false, d wird erhöht
System.out.println( d ); // 1
```

Für XOR kann es keinen Kurzschluss-Operator geben, da immer beide Operanden ausgewertet werden müssen, bevor das Ergebnis feststeht.

---

<sup>34</sup> Den Begriff verwendet die Java-Sprachdefinition nicht! Siehe dazu auch <https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.23>.

**Hinweis**

Unter gewissen Voraussetzungen kann der Verzicht auf den Kurzschlussoperator performanter sein, weil der Compiler Verzweigungen und Sprünge einsparen kann.



### 2.4.9 Der Rang der Operatoren in der Auswertungsreihenfolge

Aus der Schule ist der Spruch »Punktrechnung geht vor Strichrechnung« bekannt, sodass sich der Ausdruck  $1 + 2 \times 3$  zu 7 und nicht zu 9 auswertet.<sup>35</sup>

**Beispiel**

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.

Neben Plus und Mal gibt es eine Vielzahl von Operatoren, die alle ihre eigenen Vorrangregeln besitzen.<sup>36</sup> Der Multiplikationsoperator besitzt zum Beispiel eine höhere Priorität und damit eine andere Auswertungsreihenfolge als der Plus-Operator.

Die *Rangordnung* der Operatoren (engl. *operator precedence*) ist in [Tabelle 2.11](#) aufgeführt, wobei auf den Lambda-Pfeil `->` verzichtet wird. Der arithmetische Typ steht für Ganz- und Fließkommazahlen, der integrale Typ für `char` und Ganzzahlen und der Eintrag »primitiv« für jegliche primitiven Datentypen (also auch `boolean`):

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

**Tabelle 2.11** Operatoren mit Rangordnung in Java (Operatorrangfolge)

<sup>35</sup> Dass von diesen Rechnungen eine gewisse Spannung ausgeht, zeigen diverse Fernsehkanäle, die damit ihr Abendprogramm füllen.

<sup>36</sup> Es gibt (sehr alte) Programmiersprachen wie APL, die keine Vorrangregeln kennen. Sie werten die Ausdrücke streng von rechts nach links oder umgekehrt aus.

Operator	Rang	Typ	Beschreibung
+, -	3	arithmetisch	Addition und Subtraktion
+	3	String	String-Konkatenation
<<	4	integral	Verschiebung links
>>	4	integral	Rechtsverschiebung mit Vorzeichen-erweiterung
>>>	4	integral	Rechtsverschiebung ohne Vorzeichen-erweiterung
<, <=, >, >=	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	arithmetisch	Zuweisung mit Operation
+=	14	String	Zuweisung mit String-Konkatenation

Tabelle 2.11 Operatoren mit Rangordnung in Java (Operatorrangfolge) (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.

### Beispiel

[zB]

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 das Und stärker als das Oder bindet, also der Ausdruck als  $(A \&\& B) || C$  und *nicht* als  $A \&\& (B || C)$  gelesen wird und somit 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 in ihrem Fall 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 oder Nicht-darstellbarkeit 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  $-16.777.217 + 16.777.216$  den Wert  $-1$ , und  $-1$  plus  $+1$  ist  $0$ . Im zweiten Fall liefert  $-16.777.217 + (16.777.216 + 1) = -16.777.217 + 16.777.217 = 0$ . Doch Java wertet  $a + b$  durch die Beschränkung von float zu 0 aus, sodass bei 0 plus c die Ausgabe 1 statt 0 erscheint.

### 2.4.10 Die Typumwandlung (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 Typumwandlung

In der Praxis kommt es also vor, dass Datentypen konvertiert werden müssen. Dies nennt sich *Typumwandlung* (engl. *typecast*, kurz *cast*). Java unterscheidet zwei Arten der Typumwandlung:

- ▶ *Automatische (implizite) Typumwandlung*: 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 Typumwandlung*: Ein größerer Typ kann einem kleineren Typ mit möglichem Verlust von Informationen zugewiesen werden.

Typumwandlungen gibt es bei primitiven Datentypen und bei Referenztypen. Während die folgenden Absätze die Anpassungen bei einfachen Datentypen beschreiben, kümmert sich Kapitel 6, »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
<code>byte</code>	<code>short, int, long, float, double</code>
<code>short</code>	<code>int, long, float, double</code>
<code>char</code>	<code>int, long, float, double</code>
<code>int</code>	<code>long, float, double</code>
<code>long</code>	<code>float, double</code>
<code>float</code>	<code>double</code>

Tabelle 2.12 Implizite Typumwandlungen

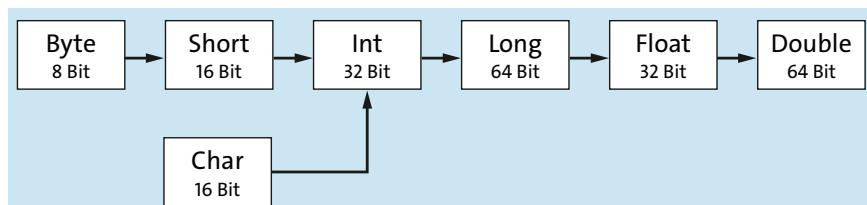


Abbildung 2.4 Grafische Darstellung der automatischen Typumwandlung

Die Anpassung wird im Englischen *widening conversion* genannt, weil sie den Wertebereich automatisch erweitert. Der Typ `boolean` taucht nicht auf, er lässt sich in keinen anderen primären Typ konvertieren.

#### Hinweis

Dass ein `long` auf ein `double` gebracht werden kann – das Gleiche gilt für `int` auf `float` –, ist wohl im Nachhinein als Fehler in der Sprache Java zu sehen, denn es gehen Informationen verloren. Ein `double` kann die 64 Bit für Ganzzahlen nicht so »effizient« nutzen wie ein `long`. Ein Beispiel:

```
System.out.println( Long.MAX_VALUE ); // 9223372036854775807
double d = Long.MAX_VALUE;
```



```
System.out.printf( "%.0f%n", d );      // 9223372036854776000
System.out.println( (long) d );        // 9223372036854775807
```

Die implizierte Typumwandlung sollte aber verlustfrei sein.



### 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 Typumwandlung

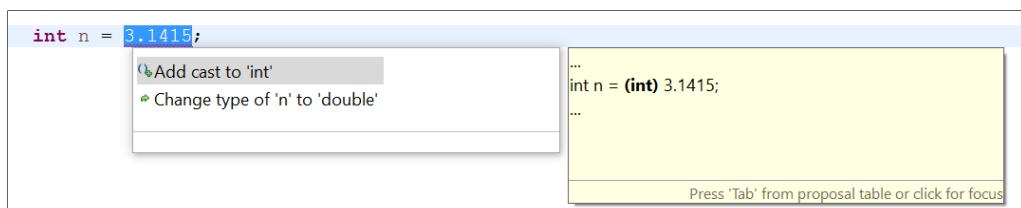
Die explizite Anpassung engt einen Typ ein, sodass diese Operation im Englischen *narrowing conversion* genannt wird. Der gewünschte Typ für eine Typumwandlung wird vor den umzuandelnden Datentyp in Klammern gesetzt. Uns muss bewusst sein, dass bei jeder expliziten Typumwandlung Information verloren gehen können.



### Beispiel

Umwandlung einer Fließkommazahl in eine Ganzzahl, der gesamte Nachkommaanteil verschwindet:

```
int n = (int) 3.1415;           // n = 3
```



**Abbildung 2.5** Passt der Typ eines Ausdrucks nicht, lässt er sich mit den Tasten **Strg**+**1** korrigieren.

Eine Typumwandlung hat eine sehr hohe Priorität. Daher muss der Ausdruck gegebenenfalls geklammert werden.

### Beispiel

[zB]

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 Ganzahltyp kann es selbstverständlich zum Verlust von Genauigkeit kommen sowie zur Einschränkung des Wertebereichs. Bei der Konvertierung von Fließkommazahlen verwendet Java eine Rundung gegen null, schneidet also schlicht den Nachkommaanteil ab.

### Beispiel

[zB]

Explizite Typumwandlung einer Fließkommazahl in ein int:

```
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
int r = (int)(Math.random() * 5);        // 0 <= r <= 4
```

### Automatische Typumwandlung bei Berechnungen mit byte und short in int \*

Eine Operation vom Typ int mit int liefert den Ergebnistyp int, und long mit long liefert ein long.

**Listing 2.14** src/main/java/AutoConvert.java, main()

```
int    int1  = 1, int2 = 2;
int    int3  = int1 + int2;
long   long1 = 1, long2 = 2;
long   long3 = long1 + long2;
```

Diese Zeilen übersetzt der Compiler wie erwartet. Und so erscheint es intuitiv, dass das Gleiche auch für die Datentypen short und byte gilt. Während

```
short short1 = 1, short2 = 2;
byte  byte1  = 1, byte2  = 2;
```

noch funktioniert, führt

```
short short3 = short1 + short2; // ☠ Type mismatch: cannot convert from int to short
byte byte3 = byte1 + byte2; // ☠ Type mismatch: cannot convert from int to byte
```

zu einem Compilerfehler.

Es ist nicht möglich, ohne explizite Typumwandlung zwei short- oder byte-Zahlen zu addieren. Richtig ist:

```
short short3 = (short)( short1 + short2 );
byte byte3 = (byte)( byte1 + byte2 );
```

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.



### Tipp

Kleine Typen wie short und byte führen oft zu Problemen. Wenn sie nicht absichtlich in großen Arrays verwendet werden und Speicherplatz nicht ein absolutes Kriterium ist, erweist sich int als die beste Wahl – auch weil Java nicht durch besonders intuitive Typkonvertierungen 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.<sup>37</sup>

---

<sup>37</sup> <https://docs.oracle.com/javase/specs/jls/se11/html/jls-5.html#jls-5.6.1>

## Keine Typumwandlung zwischen einfachen Typen und Referenztypen

Allgemeine Umwandlungen zwischen einfachen Typen und Referenztypen gibt es nicht. Falsch sind zum Beispiel:

**Listing 2.15** src/main/java/TypecastPrimRef.java, Ausschnitt main()

```
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 Typumwandlung aus, ist aber in Wirklichkeit eine Technik, die sich *Autoboxing* nennt ([Abschnitt 10.5](#), »Wrapper-Klassen und Autoboxing«, geht näher darauf ein):

**Listing 2.16** src/main/java/TypecastPrimRef.java, Ausschnitt main()

```
Long lōng = (Long) 2L;      // Alternativ: Long lōng = 2L;
System.out.println( (Boolean) true );
((Integer)2).toString();
```

## Methoden zur Typumwandlung

Bei der explizierten Typumwandlung von Ganzzahlen schneidet Java die höherwertigen Bytes ab. Bei der Konvertierung von long (8 Byte) in ein int (4 Byte) fallen die oberen 4 Byte heraus. Sind diese vier oberen Byte 0x00 – wir betrachten nur die positiven Ganzzahlen –, dann gibt es keinen Verlust an Informationen. Wollen wir von long in int umwandeln, aber ein Verlust von Informationen soll gemeldet werden, so kann die statische Math-Methode `int toIntExact( long value )` verwendet werden – sie löst eine Ausnahme aus, wenn die Umwandlung einen Datenverlust bedeutet.

Bei der Umwandlung von Fließkommazahlen in Ganzzahlen kommt es immer zum Abschneiden der Nachkommastellen. Die Klasse Math hat Methoden, die auch runden können, dazu zählen `Math.round(float)` und `long round(double)`; [Kapitel 22](#), »Bits und Bytes, Mathematisches und Geld«, gibt detaillierte Auskunft über die Mathe-Klasse.

## Typumwandlung 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  $E1 = (\text{Typ von } E1)((E1) \# (E2))$ .



### Beispiel

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 Typumwandlung vor, sodass die Bedeutung bei  $i = (\text{int})(i + 30.2)$  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

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
```

Die String-Konkatenation ist strikt von links nach rechts und natürlich nicht kommutativ wie die numerische Addition. 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 2.17** src/main/java/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  
System.out.println( 1 + 2 + "3" + (4 + 5) + 6 ); // 3396
```



### Hinweis

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.

### char-Zeichen in der Konkatenation

Nur eine Zeichenkette in doppelten Anführungszeichen ist ein String, und der Plus-Operator entfaltet seine besondere Wirkung. Ein einzelnes Zeichen in einfachen Hochkommata konvertiert Java nach den Regeln der Typumwandlung bei Berechnungen in ein int, und Additionen sind Ganzzahl-Additionen.

```
System.out.println( '0' + 2 );      // 50, denn der ASCII-Wert von '0' ist 48
System.out.println( 'A' + 'a' );    // 162, denn 'A'=65, 'a'=97
```

### 2.4.12 Operator vermisst \*

Einige Programmiersprachen haben einen Potenz-Operator (etwa \*\*), den es in Java nicht gibt. 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 bzw. die primitiven Datentypen immer eine feste Länge haben. Skriptsprachen 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 Skriptsprachen das Prüfen von Zeichenketten gegen reguläre Ausdrücke, etwa Perl mit den Operatoren =~ und !~. Beim Testen von Referenzen auf Identität gibt es in Java den Operator ==. Einige Programmiersprachen bieten zusätzlich einen Operator ===, sodass mit dem einen Operator ein Test auf Gleichheit und mit dem anderen ein Test auf Identität möglich ist. Wir werden uns in Kapitel 3, »Klassen und Objekte«, mit der Identität und dem ==-Operator näher beschäftigen.

## 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 Verzweigung mit der 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.

Ein Beispiel: Mit der if-Anweisung wollen wir testen, ob der Anwender eine Zufallszahl richtig geraten hat:

**Listing 2.18** src/main/java/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 Ausgabeanweisungen 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 mit "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



In Java muss der Testausdruck für die Bedingung der `if`-Anweisung ohne Ausnahme vom Typ `boolean` sein – für Schleifenbedingungen gilt das Gleiche. Andere Programmiersprachen wie C(++) oder PHP bewerten 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. In PHP wäre auch ein leerer String ein leerer Array `FALSE`.

## if-Abfragen und Blöcke

Hinter dem `if` und der Bedingung erwartet der Compiler eine Anweisung. Ein `{}`-Block ist eine besondere Anweisung, und die nutzt das Beispiel. Ist nur eine Anweisung in Abhängigkeit von der Bedingung auszuführen, kann die Anweisung direkt ohne Block gesetzt werden. Folgende Varianten sind also identisch:

```
if ( number == guess ) {
    System.out.println( "Super getippt!" );
}
```

und

```
if ( number == guess )
    System.out.println( "Super getippt!" );
```

Ein Programmfehler entsteht, wenn der `{}`-Block fehlt, aber mehrere Anweisungen in Abhängigkeit von der Bedingung ausgewertet werden sollen. Dazu ein Beispiel: Eine `if`-Anweisung soll testen, ob die geratene Zahl gleich der Zufallszahl war, und dann, wenn das der Fall war, die Variable `number` mit einer neuen Zufallszahl belegen und eine Ausgabe liefern. Zunächst die semantisch falsche Variante:

```
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 ordnet nur die nächstfolgende Anweisung der Fallunterscheidung zu, auch wenn die Optik etwas anderes suggeriert.<sup>38</sup> Dies ist eine große Gefahr für Programmierer, die optisch Zusammenhänge schaffen wollen, die in Wirklichkeit nicht existieren. Der

---

<sup>38</sup> In der Programmiersprache Python bestimmt die Einrückung die Zugehörigkeit.

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!" );
}
```

Grundsätzlich Anweisungen in Blöcke zu setzen – auch wenn nur eine Anweisung im Block steht – ist also nicht verkehrt.



### 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. Es sollte zudem immer nur eine Anweisung in einer Zeile stehen.

## 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

Ist die Bedingung erfüllt, wird der if-Zweig durchlaufen. Doch wie lässt es sich programmieren, dass im umgekehrten Fall – die Bedingung ist nicht wahr – auch Programmcode ausgeführt wird?

Eine Lösung ist, eine zweite if-Anweisung mit negierter Bedingung zu schreiben:

```
if ( number == guess )
    System.out.println( "Super getippt!" );
if ( number != guess )
    System.out.printf( "Tja, stimmt nicht, habe mir %s gedacht!", number );
```

Allerdings gibt es eine bessere Lösung: 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 2.19** src/main/java/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?

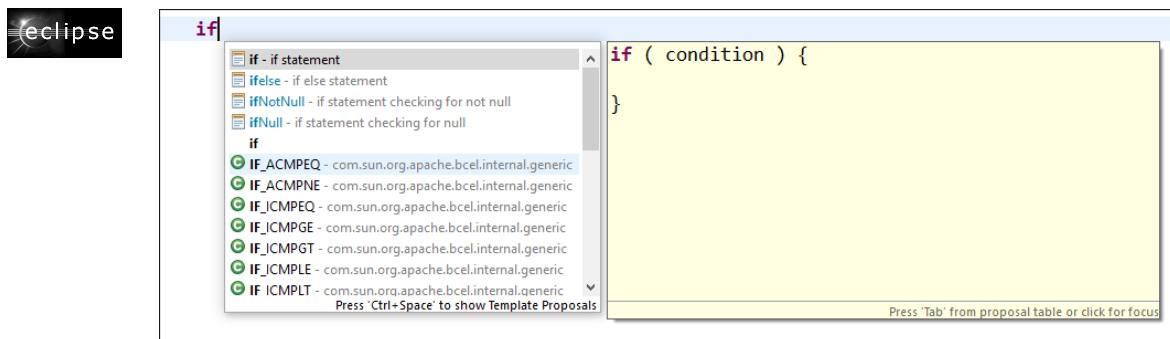
```
if ( Ausdruck1 )
    if ( Ausdruck2 )
        Anweisung1;
else
    Anweisung2;
```

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** Die Tastenkombination [Strg] + Leertaste hinter dem `if` 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 zu 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 bzw. 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 usw.

### Beispiel

Kaskadierte if-Anweisungen sollen uns helfen, die Variable days passend zu dem Monat (vorbelegte Variable month) und der Information, ob das Jahr ein Schaltjahr ist (vorbelegte boolean-Variable isLeapYear), zu belegen:

```
int month = 2; boolean isLeapYear = false; int days;
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;

```

In dem kleinen Programm ist semantisch eingerückt, eigentlich würden die Anweisungen bei jedem `else` immer weiter nach rechts wandern.

Die eingerückten Verzweigungen nennen sich auch *angehäufte if-Anweisungen* oder *if-Kaskade*, da jede `else`-Anweisung ihrerseits weitere `if`-Anweisungen enthält, bis alle Abfragen gestellt 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 2.20** src/main/java/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 // number < guess
            System.out.println( "Nee, meine Zahl ist kleiner als deine!" );
    }
}

```

### 2.5.3 Der Bedingungsoperator

In Java gibt es einen einzigen Operator, der drei Operanden benutzt: der *Bedingungsoperator*, auch genannt *Konditionaloperator*. 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 ? und : voneinander getrennt. Die allgemeine Syntax ist:

*Bedingung ? Ausdruck, wenn Bedingung wahr : Ausdruck, wenn Bedingung falsch*

#### Beispiel

Die Bestimmung des Maximums ist eine schöne Anwendung des Bedingungsoperators:

```
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:

```
if ( a > b ) max = a; else max = b;
```

[zB]

Drei Ausdrücke kommen im Bedingungsoperator vor, was ihn zu einem ternären/trinären Operator – vom lateinischen *ternarius* (»aus drei bestehend«) – macht.<sup>39</sup> 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 sind. Der Aufruf von Methoden, die void zurückgeben, ist nicht gestattet, es ist also keine kompakte Syntax, einfach zwei beliebige Methoden abhängig von einer Bedingung ohne if aufzurufen, denn es gibt beim Bedingungsoperator immer ein Ergebnis. Mit diesem können wir alles Mögliche machen, es etwa direkt ausgeben.

#### Beispiel

Gib das Maximum von a und b direkt aus:

```
System.out.println( ( a > b ) ? a : b );
```

Das wäre mit if-else nur mit temporären Variablen möglich oder eben mit zwei println(...)-Anweisungen.

[zB]

<sup>39</sup> Der Bedingungsoperator ist zwar ein ternärer/trinärer Operator, doch könnte es im Prinzip mehrere Operatoren mit drei Operanden geben, weshalb der Umkehrschluss nicht gilt, automatisch beim ternären/trinären Operator an den Bedingungsoperator zu denken. Da allerdings in Java (und so ziemlich allen anderen Sprachen auch) bisher wirklich nur ein Operator existiert, ist es in Ordnung, statt vom Bedingungsoperator vom ternären/trinären Operator zu sprechen.

## Beispiele

Der Bedingungsoperator findet sich häufig in kleinen Methoden:

- ▶ Das Maximum oder Minimum zweier Zahlen liefern die Ausdrücke  $a > b ? a : b$  bzw.  $a < b ? a : b$ .
- ▶ Den Absolutwert einer Zahl liefert  $x >= 0 ? x : -x$ .<sup>40</sup>
- ▶ Ein Ausdruck soll eine Zahl  $n$ , die zwischen 0 und 15 liegt, in eine Hexadezimalzahl konvertieren:  $(\text{char})((n < 10) ? ('0' + n) : ('a' - 10 + n))$ .

## Geschachtelte Anwendung des Bedingungsoperators\*

Die Anwendung des Bedingungsoperators 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 einen Ausdruck schreiben, der für eine Zahl  $n$  abhängig vom Vorzeichen  $-1$ ,  $0$  oder  $1$  liefert, lösen wir das Problem mit einem geschachtelten Bedingungsoperator:

```
int sign = (n < 0) ? -1 : (n > 0) ? 1 : 0;
```

Ein Umbruch hinter dem Doppelpunkt bietet sich an und macht die geschachtelten Bedingungsausdrücke deutlicher:

```
int sign = (n < 0) ? -1 :
            (n > 0) ? 1 :
            0;
```

---

<sup>40</sup> Wegen der Zweierkomplement-Arithmetik ist das nicht ganz unproblematisch, lässt sich aber nicht verhindern. [Kapitel 22, »Bits und Bytes, Mathematisches und Geld«](#), arbeitet das bei `Math.abs(...)` noch einmal auf.

### Der Bedingungsoperator ist kein L-Value \*

Der Bedingungsoperator liefert als Ergebnis einen Ausdruck zurück, der auf der rechten Seite einer Zuweisung verwendet werden kann. Da er rechts vorkommt, nennt er sich auch *R-Value*. Er lässt sich nicht derart auf der linken Seite einer Zuweisung einsetzen (L-Value), dass er eine Variable auswählt, der ein Wert zugewiesen wird.<sup>41</sup>

#### Beispiel

Die folgende Anwendung des Bedingungsoperators ist in Java nicht möglich:

```
boolean up = false, down = false;
int direction = 12;
((direction >= 0) ? up : down) = true; // ☠ Compilerfehler
```

Der Bedingungsoperator kann jedoch eine Referenz auswählen, und dann ist ein Methodenaufruf gültig.



### 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 10](#), »Besondere Typen der Java SE«)
- ▶ Aufzählungen (enum)
- ▶ Strings

#### Interna

Im Bytecode gibt es nur eine switch-Variante für Ganzzahlen. Bei Strings, Aufzählungen und Wrapper-Objekten wendet der Compiler Tricks an, um sie auf Ganzahl-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)`, mit der wir von der String-Eingabe auf das erste Zeichen zugreifen, um ein `char` zu bekommen):

---

<sup>41</sup> In C(++) kann dies durch `*((Bedingung) ? &a : &b) = Ausdruck;` über Pointer gelöst werden.

**Listing 2.21** src/main/java/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 das `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 im Abschnitt »`switch` hat ohne Unterbrechung Durchfall«.) 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 der Datentypen `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 10.7](#), »Die Spezial-Oberklasse `Enum`«. 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 hinter case aufgeführten 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, 61–99 zu definieren. Als Lösung bleiben nur angehäufte 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`-Konstante 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.



### Hinweis

Der Compiler kann natürlich nicht wissen, ob alle von uns eingesetzten Werte mit einem `case`-Block abgedeckt sind. Ein `default` kann helfen, Fehler schneller ausfindig zu machen, wenn bei `switch` eine Zahl ankommt, für die das Programm keine Operationen hinterlegt. Falls wir kein `default` einsetzen, bleibt ein unbehandelter Wert sonst ohne Folgen.

Bei Aufzählungen sieht das schon wieder anders aus, denn hier ist die Menge abzählbar. Eine Codeanalyse kann herausfinden, ob es genauso viele `case`-Blöcke wie Konstanten gibt. Eclipse kann das zum Beispiel melden.

### **switch hat ohne Unterbrechung 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 hinein, 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, das einen kleinen Parser für einfache Datumswerte realisiert. Der Parser soll mit drei unterschiedlichen Datumsangaben umgehen können, je ein Beispiel:

- ▶ "18 12": Jahr in Kurzform, Monat
- ▶ "2018 12": Jahr, Monat
- ▶ "12": Nur Monat, es soll das aktuelle Jahr implizit gelten.

**Listing 2.22** src/main/java/SimpleYearMonthParser.java

```
public class SimpleYearMonthParser {

    @SuppressWarnings( "resource" )
    public static void main( String[] args ) {
```

```

String date = "17 12";

int month = 0, year = 0;
java.util.Scanner scanner = new java.util.Scanner( date );

switch ( date.length() ) {
    case 5: // YY MM
        year = 2000;
        // Fall-through
    case 7: // YYYY MM
        year += scanner.nextInt();
        // Fall-through
    case 2: // MM
        month = scanner.nextInt();
        if ( year == 0 )
            year = java.time.Year.now().getValue();
        break;
    default :
        System.err.println( "Falsches Format" );
}
System.out.println( "Monat=" + month + ", Jahr=" + year );
}
}

```

In dem Beispiel bestimmt eine `case`-Anweisung über die Länge, wie der Aufbau ist. Ist die Länge 5, so ist die Angabe des Jahres verkürzt, und wir initialisieren das Jahr mit 2000, um im folgenden Schritt mithilfe vom Scanner das Jahr einzulesen. Zu diesem Schritt wären wir auch direkt gekommen, wenn die Länge der Eingabe 7 gewesen wäre, also das Jahr vierstellig gewesen wäre. Damit ist der Jahresanteil geklärt, es bleibt, die Monate zu parsen. Kommen wir direkt über einen String der Länge 2, ist vorher kein Jahr gesetzt, wir bekommen über `java.time.Year.now().getValue()` das aktuelle Jahr, andernfalls überschreiben wir die Variable nicht.

Was sollte der Leser von diesem Beispiel mitnehmen? Eigentlich nur Kopfschütteln für eine schwer zu verstehende Lösung. Das Durchfallen ist eigentlich nur zur Zusammenfassung mehrerer `case`-Blöcke sinnvoll zu verwenden.

### Sprachvergleich \*

Obwohl ein fehlendes `break` zu lästigen Programmierfehlern führt, haben die Entwickler von Java dieses Verhalten vom syntaktischen Vorgänger C übernommen. Eine interessante andere Lösung wäre gewesen, das Verhalten genau umzudrehen und das Durchfallen explizit einzulegen.

fordern, zum Beispiel mit einem Schlüsselwort. Dazu gibt es eine interessante Entwicklung: Java übernimmt diese Eigenschaft von C(++), sie wiederum erbt den Durchfall 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. Das Gleiche gilt für die neue Programmiersprache Swift; auch hier gibt es die Anweisung fallthrough. Selbst in C++ gibt es seit dem Standard C++17 das Standardattribut `[[fall-through]]`<sup>42</sup> – Attribute sind mit Java-Annotationen vergleichbar; es steuert den Compiler, bei einem Durchfallen keine Warnung anzuzeigen.

### Stack-Case-Labels

Stehen mehrere case-Blöcke untereinander, um damit Bereiche abzubilden, nennt sich das auch *Stack-Case-Labels*. Nehmen wir an, eine Variable `hour` steht für eine Stunde am Tag, und wir wollen herausfinden, ob Mittagsruhe, Nachtruhe oder Arbeitszeit ist:

**Listing 2.23** src/main/java/RestOrWork.java, Ausschnitt

```
int hour = 12;

switch ( hour ) {
    // Nachtruhe von 22 Uhr bis 6 Uhr
    case 22:
    case 23:
    case 24: case 0:
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        System.out.println( "Nachtruhe" );
        break;

    // Mittagsruhe von 13 bis 15 Uhr
    case 13:
    case 14:
        System.out.println( "Mittagsruhe" );
        break;
```

---

<sup>42</sup> <http://en.cppreference.com/w/cpp/language/attributes>

```

default :
    System.out.println( "Arbeiten" );
    break;
}

```

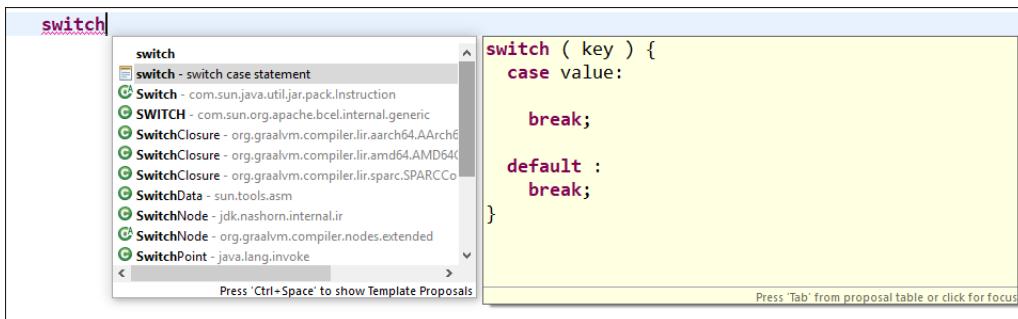


Abbildung 2.7 Die Tastenkombination [Strg] + Leertaste nach dem switch bietet an, ein Grundgerüst für eine switch-Fallunterscheidung anzulegen.

### switch auf Strings

Neben der Möglichkeit, ein switch bei Ganzzahlen zu verwenden, ist eine switch-Anweisung auf String-Objekten möglich:

Listing 2.24 src/main/java/SweetsLover.java, main()

```

String input = javax.swing.JOptionPane.showInputDialog( "Eingabe" );

switch ( input.toLowerCase() ) {
    case "kekse":
        System.out.println( "Ich mag Keeeeeke" );
        break;
    case "kuchen":
        System.out.println( "Ich mag Kuchen" );
        break;
    case "schokolade":           // Fällt durch
    case "lakritze":
        System.out.println( "Hm. Lecker" );
        break;
    default:
        System.out.printf( "Kann man %s essen?%n", input );
}

```

Obwohl direkte Zeichenkettenvergleiche möglich sind, fallen Überprüfungen auf reguläre Ausdrücke leider heraus, die insbesondere Skriptsprachen wie Ruby oder Perl 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 Immer das Gleiche mit den 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. Abhängig von der Schleifenbedingung kann der Rumpf mehrmals ausgeführt werden. Dazu wird bei jedem Schleifendurchgang die Schleifenbedingung geprüft. Das Ergebnis entscheidet, ob der Rumpf ein weiteres Mal durchlaufen (`true`) oder die Schleife beendet wird (`false`). Java bietet vier Typen von Schleifen:

Schleifentyp	Syntax
while-Schleife	<code>while ( Bedingung ) Anweisung</code>
do-while-Schleife	<code>do Anweisung while ( Bedingung );</code>
einfache for-Schleife	<code>for ( Initialisierung; Bedingung; Fortschaltung ) Anweisung</code>
erweiterte for-Schleife (auch <i>for-each-Loop</i> genannt)	<code>for ( Variablentyp variable : Sammlung ) Anweisung</code>

Tabelle 2.13 Die vier Schleifentypen in Java

Die ersten drei Schleifentypen erklären die folgenden Abschnitte, während die erweiterte `for`-Schleife nur bei Sammlungen nötig ist und daher später bei Arrays (siehe [Kapitel 4](#), »Arrays und ihre Anwendungen«) und dynamischen Datenstrukturen (siehe [Kapitel 17](#), »Einführung in 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 `while`-Schleifen der Typ der Bedingung `boolean` sein.<sup>43</sup>

---

<sup>43</sup> Wir hatten das Thema bei `if` schon angesprochen: In C(++) ließe sich `while ( i )` schreiben, was in Java `while ( i != 0 )` wäre.



## Beispiel

Zähle von 100 bis 40 in Zehnerschritten herunter:

**Listing 2.25** src/main/java/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 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 ( Files.notExists( Paths.get( "dump.bin" ) ) )
    ;
```

Existiert die Datei nicht, liefert notExists(...) die Rückgabe true, die Schleife läuft weiter, und es folgt sofort ein neuer Existenztest. Existiert die Datei, ist die Rückgabe false, und dies lässt das Ende der Schleife ein. Tipp an dieser Stelle: Anstatt direkt zum nächsten Dateitest überzugehen, sollte eine kurze Verzögerung eingebaut werden.

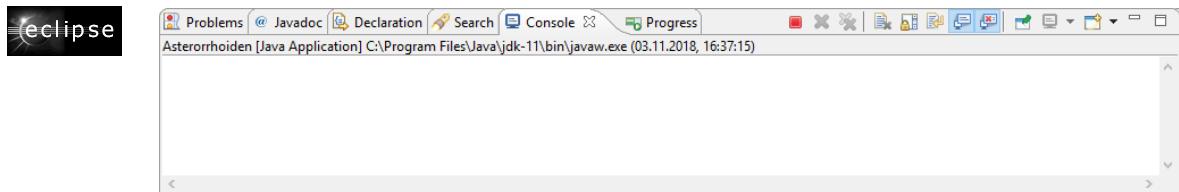
## Endlosschleifen

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 2.26** src/main/java/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, »Schleifenabbruch mit break und zurück zum Test mit continue«](#), genauer an. Genau genommen beenden aber auch nicht abgefangene Exceptions oder auch `System.exit(int)` die Programme.



**Abbildung 2.8** Die Ansicht »Console« mit der Schaltfläche zum Beenden von Programmen

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.

### 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 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 2.27** src/main/java/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 // 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.<sup>44</sup> 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`-Blocks 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.

Die äquivalente `do-while`-Schleife sieht wie folgt aus:

```

if ( Ausdruck )
    do
        Anweisung
    while ( Ausdruck ) ;

```

---

<sup>44</sup> 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.

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 erst dann ausgeführt, wenn die Bedingung wahr ist.



#### Beispiel

Gib die Zahlen von 1 bis 10 auf dem Bildschirm aus:

**Listing 2.28** src/main/java/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. Der Ausdruck initialisiert die Variable `i`, das Ergebnis wird dann aber 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.<sup>45</sup> Es darf keine lokale Variable mit dem gleichen Namen geben. Var kann

---

<sup>45</sup> 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 sie 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.

- ▶ *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, bzw. kein weiteres Mal, 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 Fortschaltausdruck:* 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-Errors* 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 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 Erdenkliche testen, nur das Ergebnis muss hier ein `boolean` sein.

### Eine `for`-Schleife muss keine Zählschleife sein

Die `for`-Schleife zeigt kompakt im Kopf alle wesentlichen Informationen, ist aber nicht auf das Hochzählen von Werten beschränkt. Sie ist vielmehr dann eine gute Option, wenn es eine Variable gibt, deren Zustand in jeder Iteration verändert wird und der Abbruch irgendwie abhängig von der Variablen ist.

### 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 in der `for`-Schleife 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:

**Listing 2.29** src/main/java/Superstar.java, main()

```
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 zählt die äußere Schleife mit *i* die Zeilen, und die innere Schleife gibt die Sternchen in eine Zeile aus, ist also für die Spalte verantwortlich.

Da Schleifen beliebig tief verschachtelt werden können, muss besonderes Augenmerk auf die Laufzeit gelegt werden. Die inneren Schleifen werden mit ihren Durchläufen immer so oft ausgeführt, wie die äußere Schleife durchlaufen wird.

### for-Schleifen und mit Komma Ausdrucksanweisungen hintereinandersetzen \*

Im ersten und letzten Teil einer for-Schleife lässt sich ein Komma einsetzen, um mehrere Ausdrucksanweisungen hintereinanderzusetzen. Damit lassen sich entweder mehrere Variablen gleichen Typs deklarieren – wie wir es schon kennen – oder mehrere Ausdrücke nebeneinanderschreiben, aber keine beliebigen Anweisungen oder sogar andere Schleifen.

Mit den Variablen *i* und *j* können wir auf diese Weise 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 werden dadurch lesbarer, dass 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 steht an dieser Stelle ein absoluter Vergleich mit ==, der aus zwei Gründen problematisch werden kann.

### Frage

Das Programm zählt bis 10, oder?

```
int input = new java.util.Scanner( System.in ).nextInt();
for ( int i = input; i != 11; i++ )
    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. Kommt 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 war sicherlich eine andere. Die Schleife sollte nur so lange zählen, wie i kleiner 11 ist, und nicht einfach nur ungleich 11. Daher passt Folgendes besser:

```
int input = new java.util.Scanner( System.in ).nextInt();
for ( int i = input; i < 11; i++ )           // Zählt immer nur bis 10 oder gar nicht
    System.out.println( i );
```

Jetzt rennt der Interpreter bei Zahlen größer 11 nicht endlos weiter, sondern stoppt die Schleife sofort ohne Durchlauf.

### Rechenun genauigkeiten sind nicht des Programmierers Freund

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`. Das gilt für Fließkommazahlen wie für Ganzzahlen. 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 linkere untere Grenze ist nie größer als die rechte 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
for ( i = 1; i <= 10; i++ )          // Auch nicht so gut
```

### 2.6.5 Schleifenabbruch mit break und zurück zum Test mit continue

Eine `break`-Anweisung innerhalb einer `for`-, `while`- oder `do-while`-Schleife beendet den Schleifendurchlauf, und die Abarbeitung wird bei der ersten Anweisung nach der Schleife fortgeführt.

Dass eine Endlosschleife 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 2.30** src/main/java/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.

### Neudurchlauf mit 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 zurückgeht. 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 Blocks zu kommen, und der beginnt mit einer neuen Eingabeaufforderung.

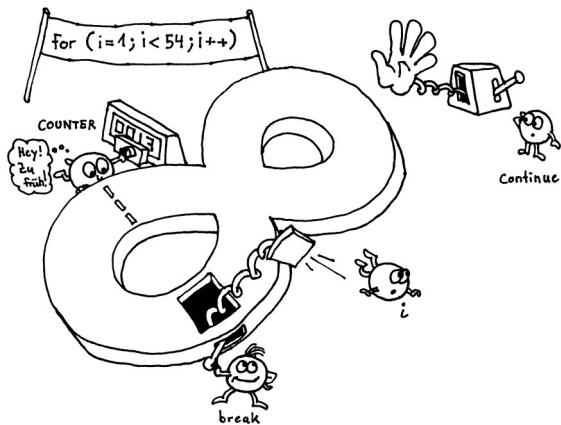
**Listing 2.31** src/main/java/GuessRight.java

```
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 fortgesetzt 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, so dass `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 2.32** src/main/java/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 auskommentiert werden. Das `break hell` in der inneren Schleife wirkt wie ein einfaches `break` ohne Marke, und das ablaufende Programm führt laufend zu Bildschirmausgaben von »hell«.

### 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:

```

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

Warum übersetzt der Compiler Folgendes ohne Murren?

**Listing 2.33** src/main/java/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. Den Zugriff auf ein Zeichen im String realisiert die String-Objektmethode `charAt(int)`; 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 `a`, `g`, `c`, `t` um 1 hoch. Falls ein falsches Zeichen im String vorkommt, wird die Schleife beendet. Und genau hier bekommt die Markierung ihren Auftritt:

**Listing 2.34** src/main/java/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 );
    }
}
```

### 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 mehrere Methoden kennengelernt und am häufigsten `println(...)` in unseren Beispielen eingesetzt. 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 *Methodenname* und einer optionalen *Parameterliste*.

Nehmen wir die bekannte statische `main(String[])`-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.<sup>46</sup>

---

#### Die Signatur einer Methode

Der Methodenname und die Parameterliste bestimmen die *Signatur* einer Methode; der Rückgabetyp und Ausnahmen gehören nicht dazu.<sup>47</sup> 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).

---

<sup>46</sup> 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 Methodenname wählte. Unzählige Zeilen Programmcode mussten daraufhin von `assert()` nach `assertTrue()` konvertiert werden.

<sup>47</sup> Um es ganz präzise zu machen: Typparameter gehören auch zur Signatur – sie sind Bestandteil von Kapitel 10, »Besondere Typen der Java SE«.

## Duck-Typing

Insbesondere Skriptsprachen, wie Python, Ruby oder Groovy, erlauben Methodendeklarationen ohne Parametertyp, sodass die Methoden mit unterschiedlichen Argumenttypen aufgerufen werden können:

```
printSum( a, b ) print 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. Programmiersprachen, die erst zur Laufzeit das Vorhandensein von Methoden oder Operatoren auf den Typ prüfen, nutzen das so genannte *Duck-Typing*. Der Begriff 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.

### 2.7.2 Signatur-Beschreibung in der Java-API

In der Java-Dokumentation sind alle Methoden mit ihren Rückgaben und Parametern inklusive möglicher Ausnahmen genau definiert. Betrachten wir die Dokumentation der statischen Methode `max(int, int)` der Klasse Math:

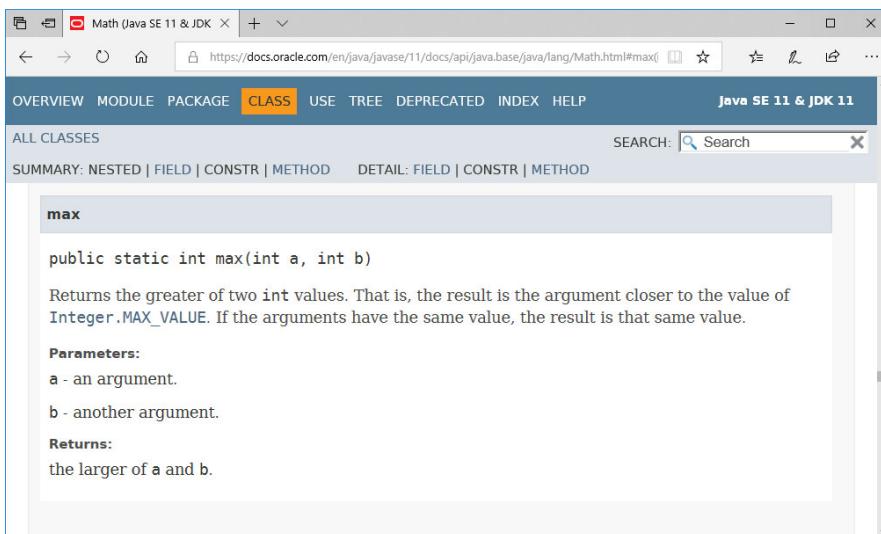


Abbildung 2.9 Die Online-API-Dokumentation für Math.max()

Die Hilfe gibt Informationen über die komplette Signatur der Methode. Der Rückgabetyp ist ein `int`, die statische Methode heißt `max`, und sie erwartet genau zwei `int`-Zahlen. Verschwie-

gen haben wir die Schlüsselwörter `public` und `static`, die so genannten *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.

### Weitere Modifizierer und Ausnahmen \*

Es gibt Methoden, die noch andere Modifizierer und eine erweiterte Signatur besitzen. Ein weiteres Beispiel aus der API zeigt 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 Ausnahmzustand widmen wir Kapitel 8, »Ausnahmen müssen sein«. 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.

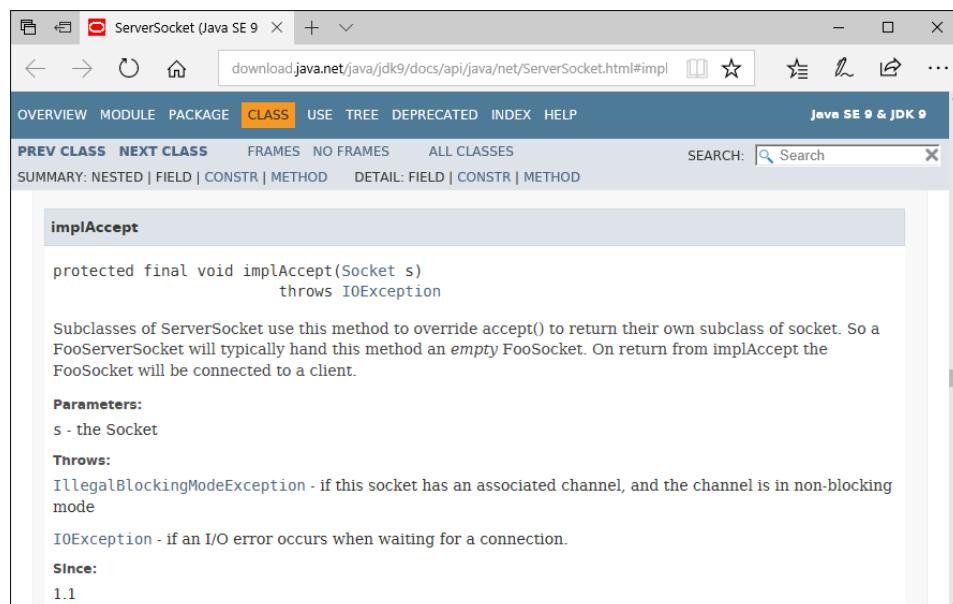


Abbildung 2.10 Ausschnitt aus der API-Dokumentation für die Klasse `java.net.ServerSocket`

### 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 `System.out.println()` gibt nichts als Ergebnis einer »Berechnung« zurück. Anders die statischen Methoden `Math.max(...)` und `Math.random()`, sie liefern ein Ergebnis. Damit ergeben sich vier unterschiedliche Typen von Methoden:

Methode	Ohne Rückgabewert	Mit Rückgabewert
Ohne Parameter	<code>System.out.println()</code>	<code>Math.random()</code>
Mit Parameter	<code>System.out.println(4)</code>	<code>Math.max(12, 33)</code>

Tabelle 2.14 Methoden mit Rückgabewerten

Die folgenden Abschnitte gehen diese vier Fälle der Reihe nach durch:

- ▶ Methode ohne Parameter und ohne Rückgabe
- ▶ Methode mit Parameter und ohne Rückgabe
- ▶ Methode ohne Parameter und mit Rückgabe
- ▶ Methode mit Parameter und mit Rückgabe

### 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 an Stelle von `void` geschrieben.

Schreiben wir eine statische Methode ohne Rückgabe und Parameter, die etwas auf dem Bildschirm ausgibt:

**Listing 2.35** src/main/java/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()` vergeben 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 auf `copyFile(...)` fallen. Methodennamen sollten immer mit dem Tätigkeitswort beginnen, gefolgt vom Was, dem Objekt.



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 haben wir nur mit statischen Methoden (auch *Klassenmethoden* genannt) gearbeitet. 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 [Kapitel 3, »Klassen und Objekte«](#), auf. Die Fehlermeldung sollte uns 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()` in `FriendlyGreeter` –, so ist der Aufruf allein mit dem Namen der Methode eindeutig. Befinden sich jedoch Methodendeklaration und Methodenauftrag in unterschiedlichen Klassen, so muss der Aufrufer den Namen der Klasse nennen; wir haben so etwas schon einmal bei Aufrufen wie `Math.max(...)` gesehen.

<pre>class FriendlyGreeter {     static void greet() {         System.out.println( "Moin!" );     } }</pre>	<pre>class FriendlyGreeterCaller {     public static void main( String[] args ) {         FriendlyGreeter.greet();     } }</pre>
---	--

## 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(double, double)`, 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(double a, double b)` 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, sonst gibt es keine Einschränkungen.

### 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 Methodenblocks bedeutet automatisch das Ende für die Parametervariablen. Der Aufrufer weiß auch nicht, wie die internen Parametervariablen heißen. Eine Typumwandlung von `int` in `double` nimmt der Compiler in unserem Fall automatisch vor. Die Argumente müssen vom Typ her natürlich passen, und es gelten die für die Typumwandlung bekannten Regeln.

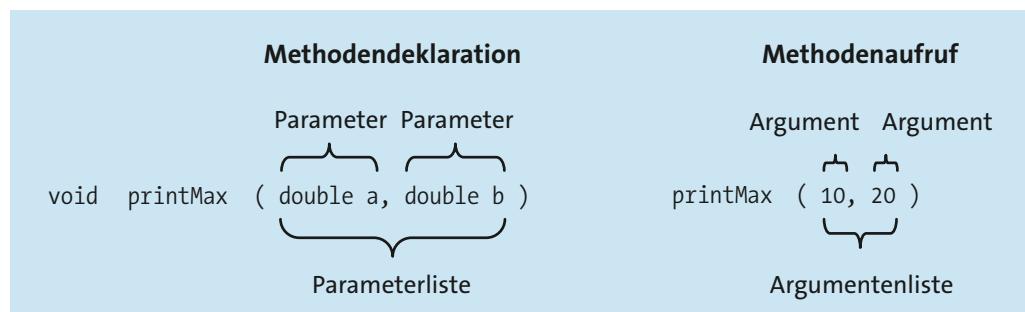


Abbildung 2.11 Die Begriffe »Parameter« und »Argument«

### 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 *Parameterübergabe-Mechanismus*. Viele Programmiersprachen verfügen 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 Methodenauftrag 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(...)` intern seine Variable `b`, so ändert dies nur die lokale Variable `b` (überschreibt also `i`), 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.<sup>48</sup>

### 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.

---

#### Frage

Was ist die Ausgabe folgender Zeilen?

```
int i = 1;
System.out.println( Math.max( i++, --i ) );
```

### 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 Methodenrumpfs sollte nicht geschrieben werden.

---

#### Beispiel

Eine statische Methode `printSqrt(double)` 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:




---

48 ... wie dies Referenzen in C++ tun.

**Listing 2.36** src/main/java/PrintSqrt.java, printSqrt()

```
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 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 Rückgaben

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 liefert Zufallszahlen von 0 bis echt kleiner 100 zurück:

```
static double largerRandom() {
    return Math.random * 100;
}
```

Fehlt der Ausdruck und ist es nur ein einfaches `return`, meldet der Compiler einen Programmfehler.

### Tipp

[+]

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.

### Was andere Programmiersprachen 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 so genanntes 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.

## Rückgaben abhängig von Eingaben

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.



### Beispiel

Eine statische Methode bildet den Mittelwert und gibt diesen zurück:

```
static double avg( double x, double y ) {
    return (x + y) / 2;
}
```

### Rückgaben nutzen \*

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. Gerne passiert das bei den String-Methoden, die String-Objekte nicht modifizieren, sondern neue String-Objekte zurückgeben.



### Beispiel

Bei einer String-Methode die Rückgabe zu ignorieren ist ein semantischer Programmfehler, aber kein syntaktischer Compilerfehler.

```
String s = "Ja";
System.out.println( s.concat( "va" ) ); // Java
s.concat( "va" );                    // Rückgabe ignoriert, Fehler!
System.out.println( s );             // Ja
```

### 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(int)`-Methode wäre:

```
static int sign( int value ) {
    if ( value < 0 )
        return -1;
    if ( value > 0 )
        return +1;
    return 0;
}
```

### Kompakte Rümpfe bei boolean-Rückgaben

Ist die Rückgabe ein Wahrheitswert, so kann oftmals die Implementierung gekürzt werden, wenn ein Ausdruck vorher ausgewertet wird, der den Rückgabewert direkt bestimmt. Gewisse Konstruktionen lassen sich kompakter schreiben:

Konstrukt mit Fallunterscheidung	Verkürzung
<code>if ( Bedingung )     return true; else     return false;</code>	<code>return Bedingung;</code>
<code>if ( Bedingung )     return false; else     return true;</code>	<code>return ! Bedingung;</code>
<code>return Bedingung ? true : false;</code>	<code>return Bedingung;</code>
<code>return Bedingung ? false : true;</code>	<code>return ! Bedingung;</code>

Tabelle 2.15 Codeoptimierungen

### 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 Programmfpad gibt, der nicht zu einem `return`-Abschluss führt.



### Beispiel

Die statische Methode `isLastBitSet(int)` 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 Wochentagsmethode 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.



### Beispiel

Die statische Methode `posOrNeg(double)` 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 ein `NaN` (*Not a Number*) als Quadratwurzel aus einer negativen Zahl

sein. Diesen speziellen Wert überprüft `posOrNeg(double)` 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 bzw. 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(a, x, 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 Gleichung. Wir möchten gerne `a <= x <= b` schreiben, doch ist dies in Java nicht erlaubt.<sup>49</sup> So müssen wir einen Und-Vergleich anstellen, der etwa so lautet: Ist `a <= x && x <= b`, dann liefere `true` zurück. In Form einer kompletten Methode:

```
static boolean between( int a, int x, int b ) {
    return a <= x && x <= b;
}
```

Die zweite Implementierung zeigt, dass sich das Problem auch ohne Und-Vergleich durch das Ausschlussprinzip lösen lässt:

```
static boolean between( int a, int x, 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 a, int x, int b ) {
    if ( a <= x )
        if ( x <= b )
```

---

<sup>49</sup> ... im Gegensatz zur Programmiersprache Python.

```

    return true;

    return false;
}

```

Ob Programmierer die Variante mit dem Und-Operator oder dem verschachtelten `if` nutzen, ist oft Geschmacksache, aber die am besten lesbare Lösung sollte gewinnen, und das dürfte in dem Fall die mit dem Und sein.

### 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 Methode nennt sich *überladen*, wenn sie unter dem gleichen Namen mehrfach auftaucht und unterschiedliche Parameterlisten hat. Das ist auf zwei Arten möglich:

- ▶ Eine Methode heißt gleich, unterscheidet sich aber in der *Anzahl der Parameter*.
- ▶ 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(...)`.



#### Beispiel

Die statische Methode `avg(...)` könnten wir für zwei und drei Parameter deklarieren:

```

static double avg( int x, int y ) {
    return (x + y) / 2;
}

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;
}

static double avg( int x, int y, int z ) {
    return (x + y + z) / 3;
}

```

### print(...) und println(...) sind überladen \*

Die bekannten print(...) und println(...) sind überladene Methoden, die etwa wie folgt deklariert sind:

```
class PrintStream {
    ...
    void print( int    arg ) { ... }
    void print( String arg ) { ... }
    void print( double arg ) { ... }
    ...
}
```

Wird nun die Methode print(...) mit irgendeinem Objekttyp aufgerufen, dann wird die am besten passende Methode herausgesucht. Das funktioniert selbst bei beliebigen Objekten, wie es Abschnitt 7.5.2, »Implementierung von System.out.println(Object)«, zeigt.

### 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 Grafikbibliothek 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(...) usw. 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 die Typumwandlung von Java die Daten automatisch anpasst 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 je-

doch nicht. In Sprachen ohne überladene Methoden wird der Methode/Funktion ein Array 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 Gültigkeitsbereich

Variablen können in jedem Block und in jeder Klasse<sup>50</sup> deklariert werden. Jede Variable hat einen *Gültigkeitsbereich* (engl. *scope*), auch *Geltungsbereich* genannt. Nur in ihrem Gültigkeitsbereich kann das Java-Programm 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 Variablen. 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 2.37** src/main/java/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 );
        }

        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

    }
}
```

<sup>50</sup> Das sind die so genannten Objektvariablen oder Klassenvariablen, doch dazu später mehr in [Kapitel 6](#), »Eigene Klassen schreiben«.

```

    int baz;           // 💀 Fehler: Duplicate local variable baz
}
}
}
}

```

Zu jeder Zeit können Blöcke aufgebaut werden. Außerhalb des Blocks sind deklarierte Variablen nicht gültig. 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 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 (C++ sei genannt) 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.

### Hinweis

Der Gültigkeitsbereich ist noch etwas komplexer, denn es kommen ja noch zum Beispiel geerbte Eigenschaften aus den Oberklassen mit hinzu.



## 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 Sprachen C# und C++ definieren in der Sprachgrammatik eine Möglichkeit für optionale Argumente, die wir in Java nicht haben).



### 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 / 100.0;
}
static double tax( double cost ) {
    return tax( cost, 19.0 ); // statt cost * 19.0 / 100;
}

```

### 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 = i + 1; // j = 3
    i = 3;             // 💀 Compilerfehler
    j = 4;             // 💀 Compilerfehler
    a = 2;             // 💀 Compilerfehler
}

```

Im Fall von Eclipse meldet der Compiler: »The final local variable ... cannot be assigned. It must be blank and not using a compound assignment.«

### 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 in [Kapitel 6, »Eigene Klassen schreiben«](#), 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 schwups, so war es geschehen, und alle lebten glücklich bis ...

Einige Leser werden vielleicht die Hand vor den Kopf schlagen und sagen: »Quatsch! Selbstgießende 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

Eine kleine Wunsch-Methode:

```
static void fairy() {
    wish();
    wish();
    fairy();
}
```

[zB]

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 Fantasieprogramme (deren Laufzeit und Speicherbedarf auch sehr schwer zu berechnen sind) gegen Java-Methoden austauschen.

#### Beispiel

Eine Endlosrekursion:

**Listing 2.38** src/main/java/EndlessRecursion.java, `down()`

```
static void down( int n ) {
    System.out.print( n + " , " );
```

[zB]

```
    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 2.39** src/main/java/Recursion.java, `down1()`

```
static void down1( int n ) {
    if ( n <= 0 )                  // Rekursionsende
        return;

    System.out.print( n + ", " );
    down1( n - 1 );
}
```

Die statische `down1(int)`-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 2.40** src/main/java/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(int)` zuerst die Zahl `n` ausgibt und anschließend rekursiv `down1(int)` aufruft. Die Methode `down2(int)` steigt jedoch erst immer tiefer ab, und die Rekursion muss beendet sein, bis es zum ersten `print(...)` kommt. Daher gibt die statische Methode `down2(int)` im Gegensatz zu `down1(int)` 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(int)`, 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(int)` wieder beendet (ein unsichtbares, nicht direkt geschriebenes `return`), und sie springt zum Aufrufer zurück. Das ist wieder die Methode `down2(int)`, 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 auruft. In den Aufrufen von `down2(int)` tauchen also wiederum wegen des rekursiven Aufrufs `down2(int)` sowie `main(...)` auf.



### Rekursion und der Stack sowie die Gefahr eines StackOverflowErrors \*

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, die 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.

### Stack-Größen ändern \*

Die Oracle-JVM nutzt für unterschiedliche Betriebssysteme unterschiedliche Stack-Größen. Sie lässt sich über einen JVM-Schalter<sup>51</sup> ändern, der `-Xss:n` heißt (oder etwas länger in der Schreibweise `-XX:ThreadStackSize=n`). Um die Stack-Größe auf 2 MiB (2.048 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.

### Keine Optimierung von Endrekursion \*

Eine *Endrekursion* (engl. *tail call recursion*) ist eine Besonderheit bei Methoden, da sie mit einem rekursiven Aufruf enden. Zum Einstieg: Ist die bekannte rekursive Implementierung der Fakultät endrekursiv?

```
int factorial( int n ) {
    if ( n <= 0 ) return 1;
    return n * factorial( n - 1 );
}
```

Zwar sieht es optisch so aus, als ob `factorial(int)` mit einem Methodenaufruf an `factorial(...)` endet, doch findet hier keine Endrekursion statt, da nach dem Methodenaufruf noch eine Multiplikation folgt – etwas umgeschrieben ist es besser zu erkennen:

---

<sup>51</sup> <https://docs.oracle.com/javase/9/tools/java.htm>

```

int factorial( int n ) {
    if ( n <= 0 ) return 1;
    int fac = factorial( n - 1 );
    return n * fac;
}

```

Die Berechnung der Fakultät lässt sich umschreiben, sodass tatsächlich eine Endrekursion stattfindet, und zwar durch Einführung eines Containers für Zwischenergebnisse, genannt *Akkumulator*:

```

int factorial( int n ) {
    return factorialTailrec( n, 1 );
}
private int factorialTailrec( int n, int accumulator ) {
    if ( n <= 0 ) return accumulator;
    return factorialTailrec( n - 1, n * accumulator );
}

```

Die umgeschriebene Version büßt gegenüber der ursprünglichen Version an Schönheit ein. Doch endrekursive Aufrufe sind attraktiv, da eine schlaue Übersetzungseinheit sie so optimieren kann, dass der rekursive Methodenaufruf durch einen Sprung ersetzt wird. In der Java-Sprache haben wir keine direkten Sprünge, doch im Bytecode schon, sodass die Basis der Optimierung im Grunde so aussehen kann:

```

private int factorialTailrec( int n, int accumulator ) {
start:
    if ( n <= 0 ) return accumulator;
    accumulator *= n;
    n--;
    goto start;
}

```

Die Rekursion ist durch eine ordinäre Schleife ersetzt, was Stack einspart und eine sehr gute Performance ergibt.

In funktionalen Programmen ergeben sich eher Situationen, in denen Endrekursion vorkommt, sodass es attraktiv ist, diese zu optimieren. Die Standard-JVM kann das bisher nicht, weil Java traditionell keine funktionale Programmiersprache ist und Endrekursion eher selten vorkommt. Zwar wird die Optimierung von Endrekursion (engl. *tail call optimization*, kurz TCO) immer wieder diskutiert und auch schon in Prototypen ausprobiert, aber nie von der Oracle-JVM implementiert. Für Entwickler heißt das, rekursive Aufrufe nicht in endrekursive Varianten umzuschreiben, da sie sowieso nicht optimiert werden und nur unleserlicher würden, sondern bei großen Datenvolumen, sprich Stack-Tiefe, auf die übliche nicht-

rekursive iterative Version umzustellen. Im Fall von factorialTailrec(...) kann dies nämlich auch vom Entwickler gemacht werden und sieht so aus:

```
private int factorialTailrec( int n, int accumulator ) {
    while ( n > 0 ) {
        accumulator *= n;
        n--;
    }
    return accumulator;
}
```

### 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.<sup>52</sup> 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 Porphy<sup>53</sup>. 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 Algo-

---

<sup>52</sup> 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).

<sup>53</sup> Gestein vulkanischen Ursprungs. Besondere Eigenschaften von Porphy sind: hohe Bruchfestigkeit, hohe Beständigkeit gegen physikalisch-chemische Wirkstoffe und hohe Wälz- und Gleitreibung.

rithmus 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 2.41** src/main/java/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 );
        }
        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", "Goldsäule", "Silbersäule" );
    }
}
```

Starten wir das Programm mit vier Scheiben, so bekommen wir folgende Ausgabe:

```
Bewege Scheibe 1 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 2 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 1 von der Silbersäule zur Goldsäule.
Bewege Scheibe 3 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 1 von der Goldsäule zur Kupfersäule.
Bewege Scheibe 2 von der Goldsäule zur Silbersäule.
Bewege Scheibe 1 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 4 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 1 von der Silbersäule zur Goldsäule.
Bewege Scheibe 2 von der Silbersäule zur Kupfersäule.
Bewege Scheibe 1 von der Goldsäule zur Kupfersäule.
Bewege Scheibe 3 von der Silbersäule zur Goldsäule.
Bewege Scheibe 1 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 2 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 1 von der Silbersäule zur Goldsäule.
```

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 Superquelle ist die *Java Language Specification*, die online unter <http://docs.oracle.com/javase/specs> 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 [https://de.wikipedia.org/wiki/M.\\_C.\\_Escher](https://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>. Amüsant ist dazu auch <http://tutego.de/go/unmain> zu lesen.

# Kapitel 3

## Klassen und Objekte

»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 (Variablen, Operatoren Fallunterscheidung, Schleifen, einfache statische Methoden) in einer neuen Grammatik für Java,
- ▶ dann die Objektorientierung (Objekte, Klassen, Vererbung, Schnittstellen), erweiterte Möglichkeiten der Java-Sprache (Ausnahmen, Generics, Closures) und zum Schluss
- ▶ die Bibliotheken (String-Verarbeitung, Ein-/Ausgabe ...).

Dieses Kapitel stellt das Paradigma der Objektorientierung in den Mittelpunkt und zeigt die Syntax, wie etwa in Java Klassen realisiert werden und Attribute und Methoden eingesetzt werden.

---

#### Hinweis

Java ist natürlich nicht die erste objektorientierte Sprache (OO-Sprache), auch C++ war nicht die erste. Klassischerweise gelten Smalltalk und insbesondere Simula-67 aus dem Jahr 1967 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*.<sup>1</sup>



#### 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

---

<sup>1</sup> Keine Sorge, alle vier Grundsäulen werden in den nächsten Kapiteln ausführlich beschrieben!

in eine Programmiersprache schwer, und es entsteht ein Bruch. Im Laufe der Zeit entwickeln sich Dokumentation und Implementierung auseinander; die Software ist dann schwer zu warten und zu erweitern. Besser ist es, objektorientiert zu denken und dann eine objektorientierte Programmiersprache zur Abbildung zu haben.



### Hinweis

Bad code can be written in any language.

### Identität, Zustand, Verhalten

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 dieselbe 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 Klas-

sen 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, ist das kein Garant für tolles Design und optimale Wiederverwendbarkeit. 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.«<sup>2</sup>

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 Abstraktion 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*<sup>3</sup> 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.

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*<sup>4</sup> genannt werden).

---

2 ... oder wie es Bertrand Meyer sagt: »Do not replace legacy software by lega-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«.

4 Den Begriff *Feld* benutze ich im Folgenden nicht. Er bleibt für Arrays reserviert.

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 Klassenarbeit mit 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:

Begriff	Erklärung
Klassenname	Ich bin ein <b>Punkt</b> .
Attribute	Ich habe eine <b>x- und y-Koordinate</b> .
Operationen	Ich kann mich <b>verschieben</b> und meine <b>Position festlegen</b> .

Tabelle 3.1 OOP-Begriffe und was sie bedeuten

Zu unserem Punkt können wir in der API-Dokumentation (<https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/java.awt/Point.html>) von Oracle nachlesen, dass er die Attribute `x` und `y` definiert, unter anderem eine Methode `setLocation(...)` besitzt und einen Konstruktor anbietet, der zwei Ganzzahlen annimmt.

## 3.3 Natürlich modellieren mit der 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.

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 Typangabe ist gegenüber Java umgekehrt: Zuerst kommt der Name der Variablen, dann der Typ bzw. bei Methoden der Typ des Rückgabewerts.

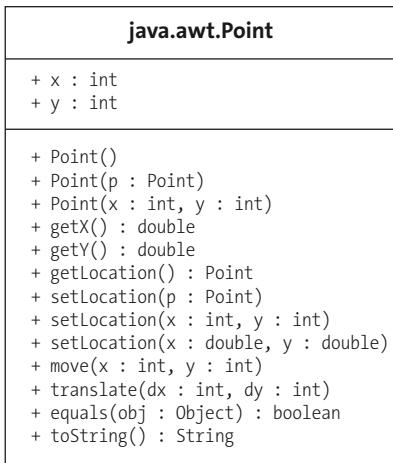


Abbildung 3.1 Die Klasse `java.awt.Point` in der UML-Darstellung

### 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 Softwaretechniken, und so bildet die UML 2.0 Konzepte wie *Model-Driven Architecture* (MDA) und *Geschäftsprozessmodellierung* (engl. *Business Process Modeling*, kurz BPM) ab und unterstützt *Echtzeitmodellierung* durch spezielle Diagrammtypen. Eine aktuelle Version des Standards lässt sich unter <http://tutego.de/go/uml> einsehen.

### 3.3.2 Wichtige Diagrammtypen der UML \*

Die 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 Programmierwurfs 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 darin, dass ein *Objektdiagramm* die Belegung der Attribute, also den Objektzustand, visualisiert. Dazu werden so genannte *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 nur ein Rechteck im Diagramm. Hat diese Person zur Laufzeit Freunde (gibt es 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 Paketdiagrammen, Komponentendiagrammen, Kompositionssstrukturdiagrammen und Verteilungsdiagrammen). Ein Anwendungsfalldiagramm und ein Sequenzdiagramm zeigen eher das dynamische Verhalten und werden *Verhaltensdiagramme* genannt. Weitere Verhaltensdiagramme sind das Zustandsdiagramm, das Aktivitätsdiagramm, das Interaktionsübersichtsdiagramm, das Kommunikationsdiagramm und das Zeitverlaufsdiagramm. 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, den Designprozess immer im Blick. 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.

Hier eine Auswahl von Produkten:

- ▶ *Enterprise Architect* (<http://www.sparxsystems.de>) ist ein Produkt von Sparx Systems; es unterstützt UML 2.5 und bietet umfangreiche Modellierungsmöglichkeiten. Für die *Business & Software Engineering Edition Standard License* sind 599 US\$ fällig. Eine 30-tägige Testversion ist frei. Das Tool ist an sich eine eigenständige Software, die Integration in Eclipse (und MS Visual Studio) ist möglich.
- ▶ *MyEclipse* (<https://www.genuitec.com/products/myeclipse>) von Genuitec besteht aus einer großen Sammlung von Eclipse-Plugins, unter anderem mit einem UML-Werkzeug. Einblick in das kommerzielle Werkzeug bekommen Sie hier: <https://www.genuitec.com/products/myeclipse/learning-center/uml/myeclipse-uml2-development-overview>.
- ▶ *ObjectAid UML Explorer for Eclipse* (<http://www.objectaid.com>) ist ein kleines und kompaktes Werkzeug, das Klassen aus Eclipse einfach visualisiert. Es entwickelt sich langsam

zu einem größeren kommerziellen Produkt. Die UML-Diagramme aus dem Buch sind in der Mehrzahl mit ObjectAid generiert.

- ▶ *Together* (<https://www.microfocus.com/de-de/products/requirements-management/together>) ist ein alter Hase unter den UML-Tools – mittlerweile ist der Hersteller Borland bei Micro Focus gelandet. Es gibt eine 30-tägige Demoversion. Die letzte Version 12.9 unterstützt Java 8, basiert auf Eclipse 4.6.1, ist also schon angestaubt.
- ▶ *Rational Rose* (<http://www-03.ibm.com/software/products/de/enterprise>) ist das professionelle UML-Werkzeug von IBM. Es zeichnet sich durch seinen Preis aus, aber auch durch die Integration einer ganzen Reihe weiterer Werkzeuge, etwa für Anforderungsdokumente und Tests.
- ▶ *UMLet* (<http://www.umlet.com>) ist ein UML-Zeichenwerkzeug und geht auf ein Projekt der Vienna University of Technology zurück. Es kann alleinstehend eingesetzt oder in Eclipse eingebettet werden. Auf Google Code liegt der offene Quellcode: <https://github.com/umlet/umlet>. Die neue Version <http://www.umletino.com> funktioniert auch im Browser.
- ▶ Der quelloffene *UML Designer* (<https://obeonetwork.github.io/UML-Designer>) greift auf viele Eclipse-Projekte zurück.
- ▶ *UML Lab* von yatta (<http://www.uml-lab.com/de/uml-lab>). Eclipse-basiertes Werkzeug mit Round-Trip-Engineering. Ab 199 €, es gibt eine Demo-Version zum Testen.

Viele Werkzeuge kamen und gingen, unter ihnen:

- ▶ *ArgoUML* (<http://argouml.tigris.org>) ist ein freies UML-Werkzeug mit UML-1.4-Notation auf der Basis von NetBeans. Es ist eigenständig und nicht in Eclipse integriert – Ende 2011 stoppte die Entwicklung, die letzte Version ist 0.34.
- ▶ *TOPCASED/PolarSys* (<http://www.topcased.org>) war ein umfangreicher UML-Editor für Eclipse. Nachfolger ist *Papyrus*.

### 3.4 Neue Objekte erzeugen

Eine Klasse beschreibt also, wie ein Objekt aussehen soll. In einer Mengen- bzw. Elementbeziehung 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 Schlüsselwort new anlegen

Objekte müssen in Java immer ausdrücklich erzeugt werden. Dazu definiert die Sprache das Schlüsselwort `new`.

#### Beispiel

Anlegen eines Punkt-Objekts:

```
new java.awt.Point();
```

[zB]

Im Grunde ist `new` so etwas wie ein unärer Operator. Hinter dem Schlüsselwort `new` 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 in [Abschnitt 3.6.3](#), »Volle Qualifizierung und import-Deklaration«, sehen, dass Entwickler diese Schreibweise auch abkürzen können). Hinter dem Klassennamen folgt ein Paar runder Klammern für den *Konstruktorauftrag*. 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.

### 3.4.2 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`-Werten, also sind mindestens 2 mal 4 Byte nötig. Den Speicherplatz nimmt die Laufzeitumgebung vom *Heap*. Der Heap wächst von einer Startgröße bis hin zu einer erlaubten Maximalgröße, damit ein Java-Programm nicht beliebig viel Speicher vom Betriebssystem abgreifen kann, was die Maschine möglicherweise in den Ruin treibt. In der HotSpot JVM ist der Heap zum Start  $\frac{1}{64}$  des Hauptspeichers groß und geht dann bis zur maximalen Größe von  $\frac{1}{4}$  des Hauptspeichers.<sup>5</sup>

#### Hinweis

Es gibt in Java nur wenige Sonderfälle, wann neue Objekte nicht über `new` angelegt werden. So erzeugt die auf nativem Code basierende Methode `newInstance()` vom `Constructor`-Objekt ein neues Objekt. Auch `clone()` kann ein neues Objekt als Kopie eines anderen Objekts erzeu-

[<>]

<sup>5</sup> <https://docs.oracle.com/en/java/javase/11/gctuning/ergonomics.html>

gen. Bei der String-Konkatenation mit `+` ist für uns zwar kein `new` zu sehen, doch der Compiler wird Anweisungen bauen, um das neue String-Objekt anzulegen.

Ist das System nicht in der Lage, genügend Speicher für ein neues Objekt bereitzustellen, versucht die automatische Speicherbereinigung in einer letzten Rettungsaktion, alles Ungebrauchte wegzuräumen. Ist dann immer noch nicht ausreichend Speicher frei, generiert die Laufzeitumgebung einen `OutOfMemoryError` und beendet das gesamte Programm.<sup>6</sup>

### Heap und Stack

Die JVM-Spezifikation sieht für Daten fünf verschiedene Speicherbereiche (engl. *runtime data areas*) vor.<sup>7</sup> 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, wobei andere Threads unbeeindruckt weiterlaufen.

### Automatische Speicherbereinigung/Garbage-Collector (GC) – es ist dann mal weg

Wird das Objekt nicht mehr vom Programm referenziert, so bemerkt dies die automatische Speicherbereinigung/der Garbage-Collector (GC) und gibt den reservierten Speicher wieder frei.<sup>8</sup> Die automatische Speicherbereinigung testet dazu regelmäßig, ob die Objekte auf dem Heap noch benötigt werden. Werden sie nicht benötigt, löscht der Objektjäger sie. 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. Es gibt verschiedene GC-Algorithmen, und jeder Hersteller einer JVM hat eigene Verfahren.

#### 3.4.3 Deklarieren von Referenzvariablen

Das Ergebnis eines `new` 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.

---

<sup>6</sup> Diese besondere Ausnahme kann aber auch abgefangen werden. Das ist für den Serverbetrieb wichtig, denn wenn ein Puffer zum Beispiel nicht erzeugt werden kann, soll nicht gleich die ganze JVM stoppen.

<sup>7</sup> § 2.5 der JVM-Spezifikation, <https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-2.html#jvms-2.5>

<sup>8</sup> Mit dem gesetzten java-Schalter `-verbose:gc` gibt es immer Konsolenausgaben, wenn der GC nicht mehr referenzierte Objekte erkennt und wegräumt.



### Beispiel

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 wie eine lokale Variable primitiven Typs zu Beginn uninitialisiert):

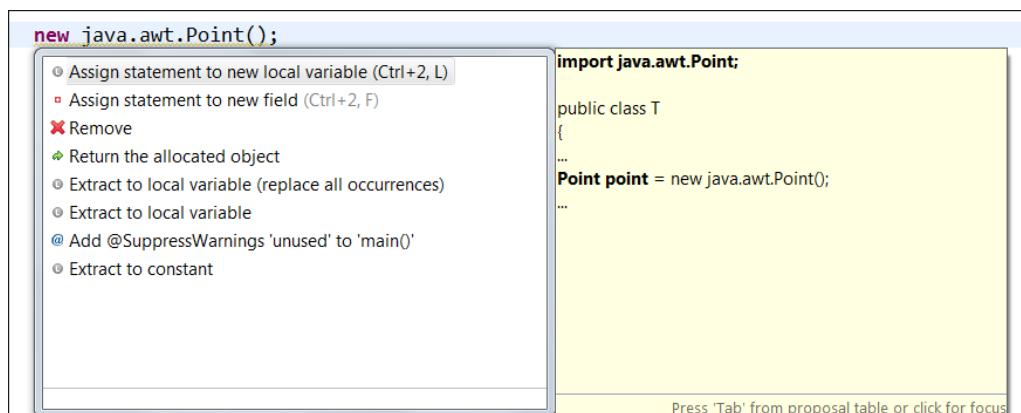
```
java.awt.Point p = new java.awt.Point();
```

Die Typen müssen natürlich kompatibel sein, und ein Punkt-Objekt geht nicht als String durch. Der Versuch, ein Punkt-Objekt 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
```

Damit speichert eine Variable entweder einen einfachen Wert (Variable vom Typ int, boolean, double ...) oder einen Verweis auf ein Objekt. Der Verweis ist letztendlich intern ein Pointer auf einen Speicherbereich, doch der ist für Java-Entwickler so nicht sichtbar.

Referenztypen gibt es in drei Ausführungen: *Klassentypen*, *Schnittstellentypen* (auch *Interface-Typen* genannt) und *Array-Typen* (auch *Feldtypen* genannt). In unserem Fall haben wir ein Beispiel für einen Klassentyp.



**Abbildung 3.2** Die Tastenkombination [Strg]+[1] ermöglicht es, entweder eine neue lokale Variable oder eine Objektvariable für den Ausdruck anzulegen.

### 3.4.4 Jetzt mach mal 'nen Punkt: Zugriff auf Objektattribute und -methoden

Die in einer Klasse deklarierten Variablen heißen *Objektvariablen* bzw. *Exemplar-, Instanz- oder Ausprägungsvariablen*. Jedes erzeugte Objekt hat seinen eigenen Satz von Objektvariablen.<sup>9</sup> Sie bilden den Zustand des Objekts.

Der Punkt-Operator . erlaubt auf Objekten den Zugriff auf die Zustände oder den Aufruf von Methoden. Der Punkt steht zwischen einem Ausdruck, der eine Referenz liefert, und der Objekteigenschaft. Welche Eigenschaften eine Klasse genau bietet, zeigt die API-Dokumentation – wenn ein Objekt eine Eigenschaft nicht hat, wird der Compiler eine Nutzung verbieten.



#### 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;
```

Ein Methodenaufruf gestaltet sich genauso einfach wie ein Attributzugriff. Hinter dem Ausdruck mit der Referenz folgt nach dem Punkt der Methodename.

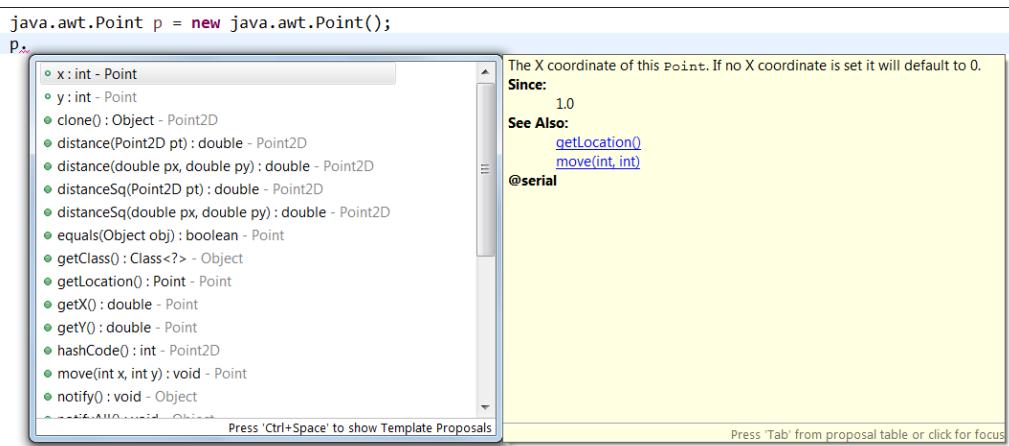


Abbildung 3.3 Die Tastenkombination **Strg** + Leertaste zeigt an, welche Eigenschaften eine Referenz ermöglicht. Eine Auswahl mit der **←**-Taste wählt die Eigenschaft aus und setzt insbesondere bei Methoden den Cursor zwischen das Klammerpaar.

<sup>9</sup> Es gibt auch den Fall, dass sich mehrere Objekte eine Variable teilen, so genannte statische Variablen. Diesen Fall werden wir später in Kapitel 6, »Eigene Klassen schreiben«, genauer betrachten.

## 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  $x$  und  $y$  immer 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 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 außerdem 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 Spiels 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 Abhängigkeit zwischen einer Klasse und dem `java.awt.Point` 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 Punktes preisgibt. Sie ist insofern besonders, als 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 für 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 Fragen des String-Objekts und seiner Länge können wir zu einer Anweisung verbinden; wir sprechen von *kaskadierten Aufrufen*.

```
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
```

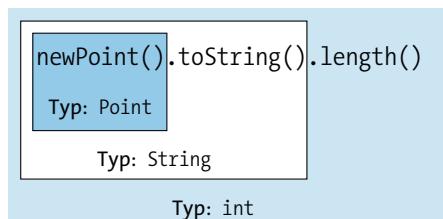


Abbildung 3.5 Jede Schachtelung ergibt einen neuen Typ.

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 die automatische Speicherbereinigung wieder Freiwild ist.



### Beispiel

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- und die 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 ein sehr ausführliches [Kapitel 7, »Objektorientierte Beziehungsfragen«](#), aber es ist jetzt schon wichtig zu verstehen, dass die Oberklasse Attribute und Methoden an Unterklassen weitergibt. 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 die Methoden aus Point2D und Object.

Nehmen wir uns einige Methoden der Oberklasse vor. Die Klassendeklaration von Point trägt ein extends Point2D, was explizit klarmacht, dass es eine Oberklasse gibt.<sup>10</sup>

<sup>10</sup> Damit ist die Klassendeklaration noch nicht vollständig, da ein implements Serializable fehlt, doch das soll uns jetzt erst einmal egal sein.

```
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.

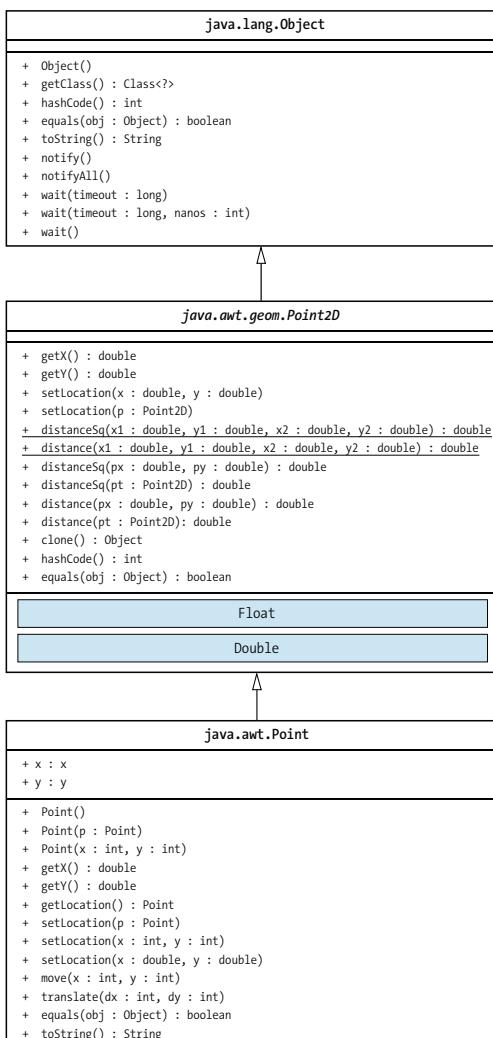


Abbildung 3.6 Vererbungshierarchie bei Point2D

### 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 PointEqualsDemo {

    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

        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
System.out.println( player.distance( snake.x, snake.y ) ); // 10.0
}
}

```

Spieler, Tür und Schlange sind wieder als Point-Objekte repräsentiert und mit Positionen vorbelegt. Beim player rufen wir die Methode distance(...) auf und übergeben den Verweis auf die Tür und Schlange. 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 new angelegt, so wird ein Konstruktor aufgerufen. Ein Konstruktor hat die Aufgabe, ein Objekt in einen Startzustand zu setzen, zum Beispiel die Objektvariablen zu initialisieren. Ein Konstruktor ist dazu ein guter Weg, denn er wird immer als Erstes aufgerufen, noch bevor eine andere Methode aufgerufen wird. Die Initialisierung im Konstruktor stellt sicher, 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. Wir nennen so einen Konstruktor auch *Copy-Konstruktor*.

Ein Konstruktor ohne Argumente ist der *parameterlose Konstruktor*, selten auch *No-Arg-Konstruktor* genannt. Jede Klasse kann höchstens einen parameterlosen Konstruktor besitzen, es kann aber auch sein, dass eine Klasse keinen parameterlosen Konstruktor deklariert, sondern nur Konstruktoren mit Parametern, also parametisierte Konstrukturen.

#### Beispiel

Die drei folgenden Varianten legen ein Point-Objekt mit denselben Koordinaten (1, 2) an; java.awt.Point ist mit Point abgekürzt:



```

1. Point p = new Point(); p.setLocation( 1, 2 );
2. Point q = new Point( 1, 2 );
3. Point r = new Point( q );

```

Als Erstes steht der parameterlose Konstruktor, im zweiten und dritten Fall handelt es sich um parametrisierte 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.
- ▶ Wir testen drei Bedingungen für den Spielstatus: 1. Hat der Spieler das Gold eingesammelt und steht auf der Tür? (Das Spiel ist zu Ende.) 2. Beißt die Schlange den Spieler? (Das Spiel ist verloren.) 3. 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.

Im Quellcode sieht das so aus:

**Listing 3.5** ZZZZZnake.java

```

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++ ) {
            java.awt.Point p = new java.awt.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!" );
        return;
    }
    if ( playerPosition.equals( snakePosition ) ) {
        System.out.println( "ZZZZZZZ. Die Schlange hat dich!" );
        return;
    }
    if ( playerPosition.equals( goldPosition ) ) {
        rich = true;
        goldPosition.setLocation( -1, -1 );
    }

    // Konsoleneingabe und Spielerposition verändern

    switch ( new java.util.Scanner( System.in ).next() ) {
        // Spielfeld ist im Bereich 0/0 .. 39/9
        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:

- ▶ Objektzustände x, y: Der Spieler und die Schlange werden bewegt, und die Koordinaten müssen neu gesetzt werden.
- ▶ Methode setLocation(...): Ist das Gold aufgesammelt, setzen wir die Koordinaten so, dass die Koordinate vom Gold nicht mehr auf unserem Raster liegt.
- ▶ Methode equals(...): Testet, ob ein Punkt auf einem anderen Punkt steht.

### Erweiterung

Wer Lust hat, an der Aufgabe noch ein wenig weiter zu programmieren, der kann Folgendes tun:

- ▶ Spieler, Schlange, Gold und Tür sollen auf Zufallskoordinaten gesetzt werden.
- ▶ Statt nur eines Stücks 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 am Anfang 5 Züge machen können, ohne dass die Schlangen sich bewegen.
- ▶ Für Vorarbeiter: Das Programm, das bisher nur eine Methode ist, soll in verschiedene Untermethoden aufgespalten werden.

## 3.6 Pakete schnüren, Importe und Komplilationseinheiten

Die Klassenbibliothek von Java ist mit Tausenden Typen sehr umfangreich und deckt alles ab, was Entwickler von plattformunabhängigen Programmen als Basis benötigen. Dazu gehören Datenstrukturen, Klassen zur Datums-/Zeitberechnung, Dateiverarbeitung usw. Die meisten Typen sind in Java selbst implementiert (und der Quellcode in der Regel aus der Entwicklungsumgebung direkt verfügbar), einige Teile sind nativ implementiert, etwa wenn es darum geht, aus einer Datei zu lesen.

Wenn wir eigene Klassen programmieren, ergänzen sie sozusagen die Standardbibliothek; im Endeffekt wächst damit die Anzahl der möglichen Typen, die ein Programm nutzen kann.

### 3.6.1 Java-Pakete

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 die beiden Klassen `BigInteger` und `BigDecimal` dem Paket `java.math` angehören, denn die Arbeit mit beliebig großen Ganz- und Fließkommazahlen gehört eben zum Mathematischen. 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`.

### 3.6.2 Pakete der Standardbibliothek

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 an dem 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 nachvollziehbar ist, ob das Paket Bestandteil jeder Distribution ist. Klassen, die in einem Paket liegen, das mit `javax` beginnt, müssen nicht zwingend zur Java SE gehören, aber dazu folgt mehr in [Abschnitt 15.1.2, »Übersicht über die Pakete der Standardbibliothek«](#).

### 3.6.3 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«. Alle Typen, die bei `import` genannt werden, merkt sich der Compiler für später. 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 z. B. ein `import java.lang.String;` nicht nötig ist.

### 3.6.4 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 ausmachen, was passiert, wenn aus dem Paket für grafische Oberflächen zusätzlich Fenster, Beschriftungen, Schaltflächen, Schieberegler usw. eingebunden werden. Die Lösung in diesem Fall ist ein `*`, das das letzte Glied in einer `import`-Deklaration sein darf:

```
import java.awt.*;
import java.math.*;
```

Mit dieser Syntax kennt der Compiler alle Typen im Paket `java.awt` und `java.math`, sodass die Klassen `Point` und `Polygon` genau bekannt sind, wie auch die Klasse `BigInteger`.

#### Hinweis

Das `*` ist nur auf der letzten Hierarchieebene 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 Typdeklarationen, sondern nur Unterpakete.

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 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.

### 3.6.5 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 zum Beispiel die Pakete awt und util, und unter javax liegen swing und sonstige Unterpakete.

Die zu einem Paket gehörenden Klassen befinden sich normalerweise<sup>11</sup> im gleichen Verzeichnis. Der Name des Pakets ist 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/insel/printer/DatePrinter.class`

Hier ist der Paketname `com.tutego.insel` und somit der Verzeichnisname `com/tutego/insel`. 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 Domänenamen. 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.

#### 3.6.6 Die package-Deklaration

Um die Klasse `DatePrinter` in ein Paket `com.tutego.insel.printer` zu setzen, müssen zwei Dinge gelten:

- ▶ Sie muss sich physikalisch in einem Verzeichnis befinden, also in `com/tutego/insel/printer`.
- ▶ 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** `src/maun/java/com/tutego/printer/DatePrinter.java`

```
package com.tutego.insel.printer;

import java.time.LocalDate;
import java.time.format.*;

public class DatePrinter {
    public static void printCurrentDate() { DateTimeFormatter formatter =

```

---

<sup>11</sup> 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.

```

        DateTimeFormatter.ofLocalizedDate( FormatStyle.MEDIUM );
        System.out.println( LocalDate.now().format( formatter ) );
    }
}

```

Hinter die package-Deklaration kommen wie gewohnt import-Anweisungen und die Typ-deklarationen.

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** src/main/java/DatePrinterUser1.java

```

public class DatePrinterUser1 {
    public static void main( String[] args ) {
        com.tutego.insel.printer.DatePrinter.printCurrentDate();      // 20.09.2017
    }
}

```

Und die Variante mit der import-Deklaration:

**Listing 3.10** src/main/java/DatePrinterUser2.java

```

import com.tutego.insel.printer.DatePrinter;

public class DatePrinterUser2 {
    public static void main( String[] args ) {
        DatePrinter.printCurrentDate();                                // 20.09.2017
    }
}

```

### 3.6.7 Unbenanntes Paket (default package)

Eine Klasse ohne Paketangabe befindet sich im *unbenannten Paket* (engl. *unnamed package*) bzw. *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.

Eine im Paket befindliche Klasse kann jede andere sichtbare Klasse aus anderen Paketen importieren, aber keine Klassen aus dem unbenannten Paket. Nehmen wir Sugar im unbenannten Paket und Chocolate im Paket com.tutego an:

*Sugar.class*  
*com/tutego/insel/Chocolate.class*

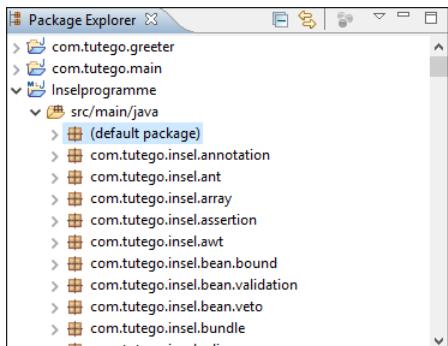


Abbildung 3.7 Das Verzeichnis »default package« steht in Eclipse für das unbenannte Paket.

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 – das auch ein Oberpaket sein kann! –, so wäre das wiederum möglich, und Chocolate könnte Sugar importieren.

`com/Sugar.class`

`com/tutego/insel/Chocolate.class`

### 3.6.8 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.

### 3.6.9 Kompilationseinheit (Compilation Unit)

Eine `.java`-Datei ist eine *Kompilationseinheit (Compilation Unit)*, die aus drei (optionalen) Segmenten besteht – in dieser Reihenfolge:

1. package-Deklaration
2. import-Deklaration(en)
3. Typdeklaration(en)

So besteht eine Kompilationseinheit aus höchstens einer Paketdeklaration (nicht nötig, wenn der Typ im Default-Paket stehen soll), beliebig vielen `import`-Deklarationen und beliebig vielen Typdeklarationen. Der Compiler übersetzt jeden Typ einer Kompilationseinheit in eine eigene `.class`-Datei. Ein Paket ist letztendlich eine Sammlung aus Kompilationseinheiten. In der Regel ist die Kompilationseinheit eine Quellcode datei, die Codezeilen könnten grundsätzlich auch aus einer Datenbank kommen oder zur Laufzeit generiert werden.

### 3.6.10 Statischer Import \*

Die `import`-Deklaration informiert den Compiler über die Pakete, sodass ein Typ nicht mehr voll qualifiziert werden muss, wenn er im `import`-Teil explizit aufgeführt wird oder wenn das Paket des Typs über \* genannt ist.

Falls eine Klasse statische Methoden oder Konstanten vorschreibt, werden ihre Eigenschaften immer über den Typnamen angesprochen. Java bietet mit dem *statischen Import* die Möglichkeit, die statischen Methoden oder Variablen ohne vorangestellten Typnamen sofort zu nutzen. Während also das normale `import` dem Compiler Typen benennt, macht ein statisches `import` dem Compiler Klasseneigenschaften bekannt, geht also eine Ebene tiefer.

#### Beispiel

[zB]

Binden für die Bildschirmausgabe die statische Variable `out` aus `System` statisch ein:

```
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(...)`.

Bilden wir in einem Beispiel mehrere statische Eigenschaften mit einem statischem `import` ein:

**Listing 3.11** src/main/java/com/tutego/insel/oop/StaticImport.java

```
package com.tutego.insel.oop;

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.*;
```

#### Best Practice

Auch wenn Java diese Möglichkeit bietet, sollte der Einsatz maßvoll erfolgen. Die Möglichkeit der statischen Importe ist nützlich, wenn Klassen Konstanten nutzen wollen, allerdings besteht auch die Gefahr, dass durch den fehlenden Typnamen nicht mehr sichtbar ist, woher die Eigenschaft eigentlich kommt und welche Abhängigkeit sich damit aufbaut. Auch gibt es Probleme mit gleichlautenden Methoden: Eine Methode aus der eigenen Klasse überdeckt statisch importierte Methoden. Wenn also später in der eigenen Klasse – oder Oberklasse – eine Methode aufgenommen wird, die die gleiche Signatur hat wie eine statisch importierte Methode, wird das zu keinem Compilerfehler führen, sondern sich die Semantik ändern, weil jetzt die neue eigene Methode verwendet wird und nicht mehr die statisch importierte.

## 3.7 Mit Referenzen arbeiten, Identität und Gleichheit (Gleichwertigkeit)

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 null-Referenz und die Frage der Philosophie

In Java gibt es drei spezielle Referenzen: `null`, `this` und `super` (wir verschieben `this` und `super` auf Kapitel 6, »Eigene Klassen schreiben«). Das spezielle Literal `null` lässt sich zur Initialisie-

rung von Referenzvariablen verwenden. Die `null`-Referenz ist typenlos, kann also jeder Referenzvariablen zugewiesen und jeder Methode übergeben werden, die ein Objekt erwartet.<sup>12</sup>

### Beispiel



Deklaration und Initialisierung zweier Objektvariablen mit `null`:

```
Point p = null;
String s = null;
System.out.println( p ); // null
```

Die Konsolenausgabe über die letzte Zeile liefert kurz »null«. Wir haben hier die String-Representation vom `null`-Typ vor uns.

Da `null` typenlos ist und es nur ein `null` gibt, kann `null` zu jedem Typ typangepasst werden, und so ergibt zum Beispiel `((String) null == null && (Point) null == null` das Ergebnis `true`. Das Literal `null` ist ausschließlich für Referenzen vorgesehen und kann in keinen primitiven Typ wie die Ganzzahl 0 umgewandelt werden.<sup>13</sup>

Mit `null` lässt sich eine ganze Menge machen. Der Haupteinsatzzweck sieht vor, damit uninitialisierte 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 oder bei einem grafischen Dialog, dass der Benutzer den Dialog abgebrochen hat; `null` ist dann ein gültiger Indikator und kein Fehlerfall.

### Hinweis



Bei einer mit `null` initialisierten lokalen Variablen funktioniert die Abkürzung `mit var` nicht; es gibt einen Compilerfehler:

```
var text = null; // ☠ Cannot infer type for local variable initialized to 'null'
```

### Auf `null` geht nix, nur die `NullPointerException`

Da sich hinter `null` kein Objekt verbirgt, ist es auch nicht möglich, eine Methode aufzurufen oder von `null` ein Attribut zu erfragen. Der Compiler kennt zwar den Typ jedes Ausdrucks, aber erst die Laufzeitumgebung (JVM) weiß, was referenziert wird. Bei dem Versuch, über die

<sup>12</sup> `null` verhält sich also so, als ob es ein Untertyp jedes anderen Typs wäre.

<sup>13</sup> Hier unterscheiden sich C(++) und Java.

null-Referenz auf eine Eigenschaft eines Objekts zuzugreifen, löst eine JVM eine NullPointerException<sup>14</sup> aus:

**Listing 3.12** src/main/java/com/tutego/insel/oop/NullPointerException.java

```
package com.tutego.insel.oop; // 1
public class NullPointerException { // 2
    public static void main( String[] args ) { // 3
        java.awt.Point p = null; // 4
        String s = null; // 5
        p.setLocation( 1, 2 ); // 6
        s.length(); // 7
    } // 8
}
```

Wir beobachten eine NullPointerException zur Laufzeit, denn das Programm bricht bei `p.setLocation(...)` mit folgender Ausgabe ab:

```
Exception in thread "main" java.lang.NullPointerException
at com.tutego.insel.oop.NullPointerException.main(NullPointerException.java:6)
```

Die Laufzeitumgebung teilt uns in der Fehlermeldung mit, dass sich der Fehler, die NullPointerException, in Zeile 5 befindet. Um den Fehler zu korrigieren, müssen wir entweder die Variablen initialisieren, das heißt, ein Objekt zuweisen wie in

```
p = new java.awt.Point();
s = "";
```

oder vor dem Zugriff auf die Eigenschaften einen Test durchführen, ob Objektvariablen auf etwas zeigen oder null sind, und in Abhängigkeit vom Ausgang des Tests den Zugriff auf die Eigenschaft zulassen oder nicht.

### »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, womit Referenzvariablen initialisiert werden können, aber was kein Objekt repräsentiert und keine Methoden besitzt. Und das kann bei der Dereferenzierung eine Null-

---

<sup>14</sup> 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.

PointerException 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, das es im System nur einmal gibt. `nil` hat auch ein paar öffentliche Methoden wie `to_s` (wie Javas `toString()`), das 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. In Objective-C, der (bisherigen) Standardsprache für iOS-Programme, gibt es das Null-Objekt `nil`. Üblicherweise passiert nichts, wenn eine Nachricht an das `nil`-Objekt gesendet wird; die Nachricht wird einfach ignoriert.<sup>15</sup>

### 3.7.2 Alles auf 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 kurzschießenden 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** src/main/java/NullCheck.java, main

```
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" );
}
```

---

<sup>15</sup> 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.

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.isEmpty()` eine `NullPointerException`, wenn jeder Teilausdruck ausgewertet würde und `s` gleich `null` wäre.

### Das Glück der anderen: null coalescing operator \*

Da `null` viel zu oft vorkommt, `null`-Referenzierungen aber vermieden werden müssen, gibt es viel Code der Art: `o != null ? o : non_null_o`.

Diverse Programmiersprachen bieten für dieses Konstrukt eine Abkürzung über den so genannten *null coalescing operator (coalescing, zu Deutsch »verschmelzend«)*, der geschrieben wird mal als `??` oder als `?:`, für unser Beispiel: `o ?? non_null_o`. Besonders hübsch ist das bei sequenziellen Tests der Art `o ?? p ?? q ?? r`, wo es dann sinngemäß lautet: Liefere die erste Referenz ungleich `null`.

Java-Programmierer kommen nicht zu diesem Glück.

### 3.7.3 Zuweisungen bei Referenzen

Eine Referenz erlaubt den Zugriff auf das referenzierte Objekt, eine Referenzvariable speichert eine Referenz. Es kann durchaus mehrere Referenzvariablen geben, die die gleiche Referenz speichern. Das wäre so, als ob ein Objekt unter verschiedenen Namen angesprochen wird – so wie eine Person von den Mitarbeitern als »Chefin« angesprochen wird, aber von ihrem Mann als »Schnuckiputzi«. Dies nennt sich auch *Alias*.

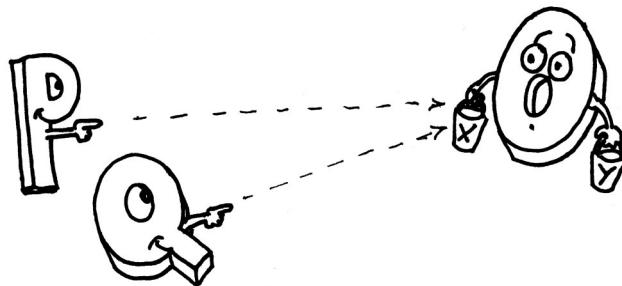


#### Beispiel

Ein Punkt-Objekt wollen wir unter einem alternativen Variablennamen 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 dasselbe Objekt. Zum besseren Verständnis: Wichtig ist, wie oft es `new` gibt, denn das sagt, wie viele Objekte die JVM bildet. Und bei den zwei Zeilen gibt es nur ein `new`, also auch nur einen Punkt.



Verweisen zwei Objektvariablen auf dasselbe Objekt, hat das natürlich zur Konsequenz, dass über zwei Wege Objektzustände ausgelesen und modifiziert werden können. 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 das-selbe Punkt-Objekt, können Änderungen über `p` auch über die Variable `q` beobachtet werden:

**Listing 3.14 ItsTheSame.java, main**

```
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 Referenztypen als Parametern

Dass sich dasselbe 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 ihr 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« – falls Spieler und Schläge zufälligerweise zusammenkommen, gibt es eine doppelte Ausgabe.

**Listing 3.15** src/main/java/com/tutego/insel/oop/DrawPlayerAndSnake.java

```
package com.tutego.insel.oop;
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 );
    }
}
```

Die Ausgabe kann so aussehen:

In dem Moment, in dem `main(...)` die statische Methode `initializeToken(Point)` aufruft, gibt es sozusagen zwei Namen für das `Point`-Objekt: `playerPosition` und `p`. Allerdings ist das nur innerhalb der virtuellen Maschine so, denn `initializeToken(Point)` 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. Die `Point`-Methode `distanceSq(int, int)` liefert den quadrierten Abstand vom aktuellen Punkt zu den übergebenen Koordinaten.

## Hinweis

Der Name einer Parametervariablen darf durchaus mit dem Namen der Argumentvariablen 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 (*Call by Value*). Das Gleiche gilt für Referenzen, die ja als eine Art Zeiger zu verstehen sind, und das sind im Prinzip nur Ganzzahlen. Daher hat auch die folgende statische Methode keine Nebenwirkungen:

### **Listing 3.16** Java's Always Call By Value.java

```
package com.tutego.insel.oop;  
import java.awt.Point;
```

```

public class JavaIsAlwaysCallByValue {

    static void clear( Point p ) {
        System.out.println( p ); // java.awt.Point[x=10,y=20]
        p = new Point();
        System.out.println( p ); // java.awt.Point[x=0,y=0]
    }

    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 `p = new Point()` in der `clear(Point)`-Methode referenziert die Parametervariable `p` ein anderes Punkt-Objekt, und der der Methode übergebene Verweis geht damit verloren. Diese Änderung wird nach außen hin natürlich nicht sichtbar, denn die Parametervariable `p` von `clear(...)` ist ja nur ein temporärer alternativer Name für das `p` aus `main`; eine Neuzuweisung an das `clear-p` ändert nicht den Verweis vom `main-p`. Das bedeutet, dass der Aufrufer von `clear(...)` – und das ist `main(...)` – kein neues Objekt unter sich hat. Wer den Punkt mit null initialisieren möchte, muss auf die Zustände des übergebenen Objekts direkt zugreifen, etwa so:

```

static void clear( Point p ) {
    p.x = p.y = 0;
}

```

### 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; }
```

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 Referenzparameter und das fehlende const

Wir haben gesehen, dass finale Variablen dem Programmierer vorgeben, dass er Variablen nicht wiederbeschreiben darf. Final können lokale Variablen, Parametervariablen, Objektvariablen oder Klassenvariablen sein. In jedem Fall sind neue Zuweisungen 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 `p` und auch an `value` verboten:

```
public void clear( final Point p, final int value )
```

Ist die Parametervariable nicht `final` und ein Referenztyp, so würden wir mit einer Zuweisung den Verweis auf das ursprüngliche Objekt verlieren, und das wäre wenig sinnvoll, wie wir im vorangehenden Beispiel gesehen haben. `final` deklarierte Parametervariablen machen im Programmcode deutlich, dass eine Änderung der Referenzvariablen unsinnig ist, und der Compiler verbietet eine Zuweisung. Im Fall unserer `clear(...)`-Methode wäre die Initialisierung direkt als Compilerfehler aufgefallen:

```
static void clear( final Point p ) {
    p = new Point();      // ☠ The final local variable p cannot be assigned.
}
```

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 Objekte verändern, 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.

#### 3.7.5 Identität von Objekten

Die Vergleichsoperatoren `==` und `!=` sind für alle Datentypen so definiert, dass sie die vollständige Übereinstimmung zweier Werte testen. Bei primitiven Datentypen ist das einfach einzusehen und bei Referenztypen im Prinzip genauso (zur Erinnerung: Referenzen lassen

sich als Pointer verstehen, was Ganzzahlen sind). Der Operator `==` testet bei Referenzen, ob sie übereinstimmen, also auf dasselbe Objekt verweisen, `!=` das Gegenteil, ob sie nicht übereinstimmen, also die Referenzen ungleich sind. 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 `==` und `!=` keine Rolle.



### 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 (Gleichwertigkeit) und die Methode equals(...)

Die allgemeingültige Lösung besteht darin, die Klasse festlegen zu lassen, wann Objekte gleich(wertig) sind. Dazu kann jede Klasse eine Methode `equals(...)` implementieren, die Exemplare dieser Klasse mit beliebigen anderen Objekten vergleicht. Die Klassen entscheiden immer nach Anwendungsfall, welche Attribute sie für einen Gleichheitstest heranziehen, und `equals(...)` liefert `true`, wenn die gewünschten Zustände (Objektvariablen) übereinstimmen.



### Beispiel

Zwei nichtidentische, 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
<b>Identität der Referenzen</b>	<code>==</code> bzw. <code>!=</code>	Nichts zu tun
<b>Gleichheit der Zustände</b>	<code>equals(...)</code> bzw. <code>! equals(...)</code>	Abhängig von der Klasse

Tabelle 3.4 Identität und Gleichheit von Objekten

### equals(...)-Implementierung von Point \*

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 ) {
        ...
        Point pt = (Point) obj;
        return (x == pt.x) && (y == pt.y); // (*)
        ...
    }
}
```

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 Punkt-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 univer-

selle Oberklasse `Object` sie vererbt. Wir greifen hier auf [Kapitel 7](#), »Objektorientierte Beziehungsfragen«, 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`. Diese Klasse sieht wie folgt aus:

**Listing 3.18** `java/lang/Object.java`, Ausschnitt

```
public class Object {  
    public boolean equals( Object obj ) {  
        return ( this == obj );  
    }  
    ...  
}
```

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.

### Sprachvergleich

Es gibt Programmiersprachen, die für den Identitätsvergleich und Gleichheitstest eigene Operatoren anbieten. Was bei Java `==` und `equals(...)` ist, ist bei Python `is` und `==`, bei Swift `===` und `==`.

## 3.8 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

## Arrays und ihre Anwendungen

»Die aus der Reihe fallen, bilden den Anfang einer neuen Reihe.«  
– Erhard Horst Bellermann (\*1937)<sup>1</sup>

### 4.1 Arrays

Ein *Array* (auch deutsch *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 Schlümpfe) nimmt immer Werte des gleichen Typs auf (nur Schlümpfe und keine Pokémon). 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 usw.
- ▶ Referenztypen
- ▶ Referenztypen anderer Arrays, um mehrdimensionale Arrays zu realisieren

#### 4.1.1 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, den lesenden Zugriff ebenso wie den schreibenden

#### Beispiel

1. Deklariere eine Variable `randoms`, die ein Array referenziert:

```
double[] randoms;
```

[zB]

<sup>1</sup> <https://ebellermann.wordpress.com>

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;
```

Wir sehen, dass eckige Klammern an verschiedenen Stellen zum Einsatz kommen: einmal zur Deklaration vom Typ, dann zum Aufbau des Arrays, dann zum Schreiben in Arrays und zum Lesen aus Arrays.

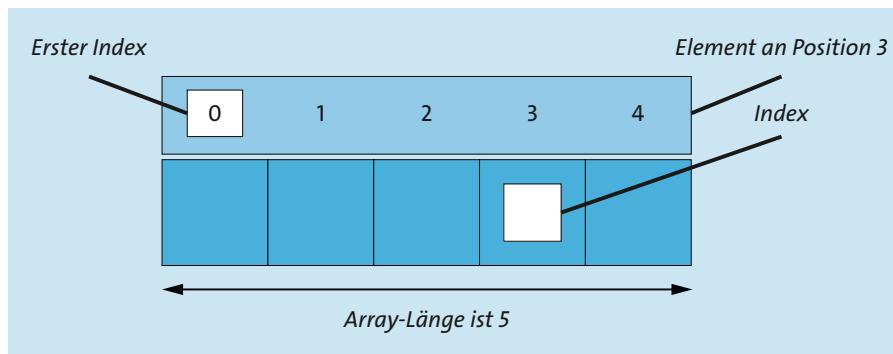


Abbildung 4.1 Begriffe eines Arrays

Die drei Punkte schauen wir uns nun detaillierter an.

### 4.1.2 Deklaration von Array-Variablen

Eine Array-Variablen-Deklaration ähnelt einer gewöhnlichen Deklaration, nur dass nach dem Datentyp die Zeichen [ und ] gesetzt werden.



#### Beispiel

Deklariere zwei Array-Variablen:

```
int[] primes;
Point[] points;
```

Eine Variable wie `primes` hat jetzt den Typ »ist Array« 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[][][];
```

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 Typebezeichner, so hat es der Java-Schöpfer James Gosling gewollt.

### 4.1.3 Array-Objekte mit new erzeugen

Das Anlegen der Array-Referenzvariablen allein erzeugt noch kein Array mit einer bestimmten Länge. In Java ist das Anlegen des Arrays genauso dynamisch wie die Objekterzeugung. Ein Array muss mit dem Schlüsselwort `new` erzeugt werden, da Arrays Objekte sind.<sup>2</sup> Die Länge des Arrays wird in eckigen Klammern angegeben. Hier kann ein beliebiger Integer-Wert stehen, auch eine Variable. Selbst 0 ist möglich. Später kann die Größe nicht mehr verändert werden.



## Beispiel

Erzeuge ein Array für zehn Elemente:

```
int[] values;  
values = new int[ 10 ];
```

Die Array-Deklaration ist auch zusammen mit der Initialisierung möglich:

```
double[] values = new double[ 10 ];
```

Die JVM initialisiert die Arrays standardmäßig: bei primitiven Werten mit 0, 0.0 oder `false` und bei Verweisen mit `null`.

---

<sup>2</sup> Programmiersprachen wie C(++) bieten bei der Felderzeugung Abkürzungen wie `int array[100]`. Das führt in Java zu einem Compilerfehler.

### Arrays sind ganz normale Objekte

Dass Arrays Objekte sind, zeigen einige Indizien:

- ▶ Eine spezielle Form der new-Schreibweise 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 Objektbedeutung: Sie vergleichen lediglich, ob zwei Variablen auf das identische 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.

#### 4.1.4 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

Wertebelegung eines Arrays bei der Initialisierung:

```
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 Array mit passender Größe angelegt, und die Elemente, die in der Aufzählung genannt sind, werden in das Array kopiert. Innerhalb der Aufzählung kann abschließend ein Komma stehen, wie die Aufzählung bei `strings` demonstriert.

Es ist möglich, dass vor der schließenden geschweiften Klammer noch ein Komma folgt, so dass es etwa `int[] primes = { 2, 3, };` heißt. Das vereinfacht das Hinzufügen, ein leeres Element produziert es nicht. Selbst Folgendes ist in Java möglich: `int[] primes = { , };`.

**Hinweis**

Die Deklaration einer Array-Variablen mit Initialisierung funktioniert mit var nicht:

```
var primes = { 2, 3 }; // ☠ Array initializer needs an explicit target-type
```

#### 4.1.5 Die Länge eines Arrays über das Attribut length auslesen

Die Anzahl der Elemente, die ein Array aufnehmen kann, wird *Größe* oder *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.

**Beispiel**

Ein Array und die Ausgabe der Länge:

```
int[] primes = { 2, 3, 5, 7, 7 + 4 };
System.out.println( primes.length ); // 5
```

**Hinweis**

Die Länge ist immer die Kapazität des Arrays. Es gibt keine Information darüber, wie viele Array-Elemente von uns in der Nutzung sind. Wenn das Array zum Beispiel 10 Elemente groß ist, aber nur die ersten 2 Positionen von uns verwendet werden, so müssen wir uns das selbst merken.

#### Array-Längen sind final

Das Attribut length eines Arrays ist nicht nur öffentlich (public) und vom Typ int, sondern natürlich auch final. Schreibzugriffe sind nicht gestattet, denn eine dynamische Vergrößerung eines Arrays ist nicht möglich; ein Schreibzugriff führt zu einem Übersetzungsfehler.

#### Warum »können« Arrays so wenig?

Arrays haben ein Attribut length und eine Methode clone() sowie die von Object geerbten Methoden. Das ist nicht viel. Für Arrays gibt es keine Klassendeklaration, also auch keine .class-Datei, die Methoden deklariert. Es gibt auch keinen Java-Code wie bei anderen Klassen wie String, Arrays oder System, die mit Javadoc dokumentiert sind und folglich in der API-Dokumentation auftauchen. Da es unendliche viele Array-Typen gibt – hinter jedem beliebigen Typ kann [] gesetzt werden –, würde das auch unendlich viele Klassendeklarationen nach sich ziehen.

Das, was Arrays können, ist in der Java Sprachdefinition (JLS) festgeschrieben. Bei jeder neuen Methode oder Änderung müsste die JLS angepasst werden, was unpraktisch ist. Natürlich wären Methoden wie `sort(...)`, `indexOf(...)` auf Array-Objekten praktisch, aber zu viel Eingebautes in der Sprache ist nicht gut und auch die Dokumentation solcher Methoden will Oracle aus der JLS heraushalten, und so sind die Methoden in eine Extraklasse `Arrays` gewandert.

#### 4.1.6 Zugriff auf die Elemente über den Index

Der Zugriff auf die Elemente eines Arrays 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 Arrays. Das heißt: Bei einem Array `a` der Länge `n` ist der gültige Bereich `a[0]` bis `a[n - 1]`.

Da der Zugriff auf die Variablen über einen Index erfolgt, werden diese Variablen auch *indexierte Variablen* genannt.



#### Beispiel

Greife auf das erste und letzte Zeichen aus dem Array zu:

```
char[] name = { 'C', 'h', 'r', 'i', 's' };
char first = name[ 0 ];                                // C
char last  = name[ name.length - 1 ];                  // s
```



#### Beispiel

Laufe das Array 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 Array einfach nur so abzulaufen und die Werte auszugeben, soll unser nächstes Programm den Mittelwert einer Zahlenfolge berechnen und ausgeben:

**Listing 4.1** src/main/java/com/tutego/insel/array/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 Array 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.

#### Hinweis

Der Index eines Arrays muss vom Typ `int` sein, das schließt Anpassungen von `byte`, `short` und `char` ein. Günstig ist ein Index vom Typ `char`, zum Beispiel als Laufvariable, wenn Arrays 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.



### Strings sind keine Arrays \*

Ein Array von `char`-Zeichen hat einen ganz anderen Typ als ein `String`-Objekt. Während bei Arrays eckige Klammern erlaubt sind, bietet die `String`-Klasse keinen Zugriff auf Zeichen über `[ ]`. Die Klasse `String` bietet jedoch einen Konstruktor an, sodass aus einem Array mit Zeichen ein `String`-Objekt erzeugt werden kann. Alle Zeichen des Arrays werden kopiert, so dass anschließend Array und `String` keine Verbindung mehr besitzen. Dies bedeutet: Wenn sich das Array ändert, ändert sich der `String` nicht automatisch mit. Das kann er auch nicht, da `Strings` unveränderlich sind.

### 4.1.7 Typische Array-Fehler

Beim Zugriff auf ein Array-Element können Fehler auftreten. Zunächst einmal kann das Array-Objekt fehlen, sodass die Referenzierung fehlschlägt.



#### Beispiel

Der Compiler bemerkt den folgenden Fehler nicht, und die Strafe ist eine `NullPointerException` zur Laufzeit:<sup>3</sup>

```
int[] array = null;
array[ 1 ] = 1;    // ☠ NullPointerException
```

Weitere Fehler können im Index begründet sein. Ist der Index negativ<sup>4</sup> oder zu groß, dann gibt es eine `IndexOutOfBoundsException`. Jeder Zugriff auf das Array wird zur Laufzeit getestet, auch wenn der Compiler durchaus einige Fehler finden könnte.



#### Beispiel

Bei folgenden Zugriffen könnte der Compiler theoretisch Alarm schlagen, was aber zumindest 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 Compilezeit
array[ 100 ] = 1;   // ☠ Fehler zur Laufzeit, nicht zur Compilezeit
```

Wird die `IndexOutOfBoundsException` nicht abgefangen, bricht das Laufzeitsystem das Programm mit einer Fehlermeldung ab. Dass die Array-Grenzen 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 Array-Zugriff korrekt ist.

### Spielerei: der Index und das Inkrement \*

Wir haben beim Inkrement schon ein Phänomen wie `i = i++` betrachtet. Ebenso ist die Anweisung bei einem Array-Zugriff zu behandeln:

```
array[ i ] = i++;
```

<sup>3</sup> Obwohl er sich bei nicht initialisierten lokalen Variablen auch beschwert.

<sup>4</sup> Ganz anders verhalten sich da Python oder Perl. Dort wird ein negativer Index dazu verwendet, ein Feld- element relativ zum letzten Array-Eintrag anzusprechen. Und auch bei C ist ein negativer Index durchaus möglich und praktisch.

Bei der Position `array[i]` wird `i` gesichert und anschließend die Zuweisung vorgenommen. Wenn wir eine Schleife herum konstruieren, erweitern wir dies zu einer Initialisierung:

```
int[] array = new int[ 4 ];
int i = 0;
while ( i < array.length )
    array[ i ] = i++;
```

Die Initialisierung ergibt 0, 1, 2 und 3. Von der Anwendung ist wegen mangelnder Übersicht abzuraten.

#### 4.1.8 Arrays als Methodenparameter

Verweise auf Arrays 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 haben bereits den Mittelwert einer Zahlenreihe ermittelt. Die Logik dafür ist perfekt in eine Methode ausgelagert:

**Listing 4.2** src/main/java/com/tutego/insel/array/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 = { 2, 3, 4 };
        System.out.println( avg( numbers ) );           // 3.0
    }
}
```

#### null-Referenzen prüfen

Referenzen bringen immer das Problem mit sich, dass sie `null` sein können. Syntaktisch gültig ist ein Aufruf von `avg(null)`. Daher sollte eine Implementierung auf `null` testen und ein falsches Argument melden, etwa so:

```
if ( array == null || array.length == 0 )
    throw new IllegalArgumentException( "Array null oder leer" );
```

Zu den Details siehe [Kapitel 8](#), »Ausnahmen müssen sein«.

#### 4.1.9 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 Array-Inhalte erst nach der Variablen Deklaration initialisiert werden oder soll das Array auch ohne Variable als Argument genutzt werden, so erlaubt Java dies nicht:

```
primes = { 2, 5, 7, 11, 13 }; // 💀 Compilerfehler
avg( { 1.23, 4.94, 9.33, 3.91, 6.34 } ); // 💀 Compilerfehler
```

Ein Versuch wie dieser schlägt mit der Compilermeldung »Array constants can only be used in initializers« fehl.

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 der `new`-Schreibweise, die 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 Arrays übergeben werden:

```
avg( new double[]{ 1.23, 4.94, 9.33, 3.91, 6.34 } );
```

Da hier ein initialisiertes Array 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 Array nicht mit eigenen Werten initialisiert.

#### 4.1.10 Die erweiterte for-Schleife

for-Schleifen laufen oft Arrays oder Datenstrukturen ab. Bei der Berechnung des Mittelwerts 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 Array ansprechen.

Weil das komplette Durchlaufen von Arrays häufig ist, gibt es eine Abkürzung für solche Iterationen:

```
for ( Typ Bezeichner : Array )
    ...
```

Die erweiterte Form der for-Schleife löst sich vom Index und erfragt jedes Element des Arrays. Das können Sie sich als Durchlauf einer Menge vorstellen, denn der Doppelpunkt liest sich als »in«. Rechts vom Doppelpunkt steht immer ein Array 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 Array-Verweis `null` ist oder das Array keine Elemente enthält:

**Listing 4.3** src/main/java/com/tutego/insel/array/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.

### Anonyme Arrays in der erweiterten for-Schleife nutzen

Rechts vom Doppelpunkt lässt sich auf die Schnelle ein Array 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 );
```

Das ist praktisch, um über eine feste Menge von Werten zu laufen. Das funktioniert auch für Objekte, etwa Strings:

```
for ( String name : new String[]{ "Cherry", "Gracel", "Fe" } )
    System.out.println( name );
```



#### Hinweis

Rechts vom Doppelpunkt kann ein Array oder ein Objekt vom Typ Iterable stehen:

```
for ( String name : Arrays.asList( "Cherry ", "Gracel", "Fe" ) )
    System.out.println( name );
```

Arrays.asList(...) erzeugt kein Array als Rückgabe, sondern baut aus der variablen Argumentliste eine Sammlung auf, die von einem speziellen Typ Iterable ist; alles, was Iterable ist, kann die erweiterte for-Schleife ablaufen. Wir kommen später noch einmal darauf zu sprechen. Unabhängig vom erweiterten for hat die Nutzung von Arrays.asList(...) noch einen anderen Vorteil, etwa bei Ist-Element-von-Anfragen, etwa so:

```
if ( Arrays.asList( 1, 2, 6, 7, 8, 10 ).contains( number ) )
    ...
```

Mehr zu der Methode gleich in [Abschnitt 4.1.19, »Die Klasse Arrays zum Vergleichen, Füllen, Suchen, Sortieren nutzen«.](#)

### 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 standardmäßig immer das ganze Array ab. Ein Anfang- und ein Ende-Index können nicht ausdrücklich gesetzt werden.
- ▶ Die Ordnung ist immer von vorn nach hinten.
- ▶ Die Schrittänge ist immer eins.
- ▶ Der Index ist nicht sichtbar.
- ▶ Die Schleife liefert ein Element, kann aber nicht in das Array schreiben.

Abbrechen lässt sich die Schleife mit einem break. Bestehen andere Anforderungen, kann weiterhin nur eine klassische for-Schleife helfen.

### 4.1.11 Arrays mit nichtprimitiven 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 Arrays, 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.

#### Arrays mit Strings durchsuchen

In unserem ersten Beispiel soll ein nichtprimitives Array Strings referenzieren und später schauen, ob eine Benutzereingabe im Array ist. String-Vergleiche lassen sich mit `equals(..)` realisieren:

**Listing 4.4** src/main/java/com/tutego/insel/array/UserInputInStringArray.java, main()

```
String[] validInputs = { "Banane", "Apfel", "Kirsche" };

boolean found = false;
while ( ! found ) {
    String input = new Scanner( System.in ).nextLine();

    for ( String s : validInputs )
        if ( s.equals( input ) ) {
            found = true;
            break;
        }
}

System.out.println( "Gültiges Früchtchen eingegeben" );
```

Zur Initialisierung des Arrays nutzt das Programm eine kompakte Variante, die drei Dinge vereint: den Aufbau eines Array-Objekts (mit Platz für drei Referenzen), die Initialisierung des Array-Objekts mit den drei Objektreferenzen und schlussendlich die Initialisierung der Variablen `validInputs` mit dem neuen Array – alles in einer Anweisung.

Für die Suche kommt das erweiterte `for` zum Einsatz, das in einer Schleife eingebettet ist, die genau dann endet, wenn das Flag `found` gleich `true` wird. Wenn wir nie einen korrekten String eingeben, wird die äußere Schleife auch nie enden. Wenn wir einen Eintrag finden, kann das Flag gesetzt werden und `break` die Array-Schleife früher verlassen.

### Zufällige Spielerpositionen erzeugen

Im zweiten Beispiel sollen fünf zufällig initialisierte Punkte in einem Array abgelegt werden. Die Punkte sollen Spieler repräsentieren.

Zunächst benötigen wir ein Array:

```
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 Array-Plätze 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 4.5** src/main/java/com/tutego/insel/array/FivePlayers.java, main()

```
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 Array `players` gleich dem Punkt mit den x/y-Koordinaten ist.

Die Ausgabe erzeugt zum Beispiel Folgendes:

```
.....&.....  
&.....  
.....  
.....&.....  
.....&.....  
...&.....&.....  
.....  
.....
```

Während die erweiterte `for`-Schleife gut das Array ablaufen kann, funktioniert das zur Initialisierung nicht, denn das erweiterte `for` ist nur zum Lesen gut. Elementinitialisierungen funktionieren bei Arrays nur mit `players[i]=...`, und dazu ist eben eine klassische `for`-Schleife mit dem Index nötig.

#### 4.1.12 Methode mit variabler Argumentanzahl (Varargs)

Bei vielen Methoden ist klar, wie viele Argumente exakt übergeben werden müssen; einer Methode `Math.random()` darf nichts übergeben werden, und bei `Math.sin(double)` ist genau ein Argument gültig.

##### `System.out.printf(...)` nimmt eine beliebige Anzahl von Argumenten an

Es gibt auch Methoden, bei denen die Anzahl gültiger Argumente prinzipiell unbeschränkt ist. Ein Beispiel ist `printf(...)`, das verpflichtend einen String als Erstes erwartet, dann aber frei ist, was sonst übergeben wird. Gültig sind zum Beispiel:

Aufruf	Variable Argumentliste
<code>System.out.printf("%n")</code>	Ist leer.
<code>System.out.printf("%s", "Eins")</code>	Besteht aus nur einem Element: "Eins".
<code>System.out.printf("%s,%s,%s", "1", "2", "3")</code>	Besteht aus drei Elementen: "1", "2", "3".

Tabelle 4.1 Gültige Aufrufe von `printf(...)`

Um die Anzahl der Parameter beliebig zu gestalten, sieht Java Methoden mit *variabler Argumentanzahl* vor, *Varargs* genannt – in anderen Programmiersprachen heißen sie *variadische Funktion*. Die Methode `printf(formatierungsstring, arg1, args2, arg3, ...)` ist so eine Varargs-Methode.

```
class java.io.PrintStream extends FilterOutputStream
implements Appendable, Closeable
```

- PrintStream printf(String format, Object... args)

Nimmt eine beliebige Liste von Argumenten an und formatiert sie nach dem gegebenen Formatierungs-String format. Der Formatierungs-String bestimmt, wie viele Argumente nötig sind. Der Compiler wertet den String aber nicht aus, kann also die Korrektheit – dass die Anzahl stimmt – auch zur Compilezeit nicht prüfen.

Eine Methode mit variabler Argumentanzahl nutzt die Ellipse (...) zur Verdeutlichung, dass eine beliebige Anzahl Argumente angegeben werden darf, dazu zählt auch die Angabe keines Elements. Der Typ fällt dabei aber nicht unter den Tisch; er wird ebenfalls angegeben.

### Durchschnitt finden von variablen Argumenten

Wir haben vorher eine Methode avg(double[] array) geschrieben, die den arithmetischen Mittelwert von Werten berechnet. Den Parametertyp können wir nun ändern in avg(double... array), sodass die Methode einfach mit variablen Argumenten gefüllt werden kann.

Ausprogrammiert sieht das so aus; es gibt nur eine kleine Änderung von [ ] in ..., sonst ändert sich an der Implementierung nichts:

**Listing 4.6** src/main/java/com/tutego/insel/array/AvgVarArgs.java, Ausschnitt

```
public class AvgVarArgs {

    static double avg( double... array ) { /* Implementierung wie vorher */ }

    public static void main( String[] args ) {
        System.out.println( avg(1, 2, 9, 3) );      // 3.75
    }
}
```



### Hinweis

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. Das bedingt automatisch, dass es nur maximal ein Vararg in der Parameterliste geben kann.

### Zusammenhang Vararg und Array

Eine Methode mit Vararg ist im Kern eine Methode mit einem Parametertyp Array. Im Bytecode steht nicht wirklich avg(double... array), sondern avg(double[] array) mit der Zusatz-

info, dass array ein Vararg ist, damit der Compiler beliebig viele Argumente und nicht ausschließlich ein `double[]`-Array als Argument erlaubt.

Der Nutzer kann eine Varargs-Methode aufrufen, ohne ein Array für die Argumente explizit zu definieren. Er bekommt auch gar nicht mit, dass der Compiler im Hintergrund ein Array mit vier Elementen angelegt hat. So übergibt der Compiler:

```
System.out.println( avg( new double[] { 1, 2, 9, 3 } ) );
```

An der Schreibweise lässt sich gut ablesen, dass wir ein Array auch von Hand übergeben können:

```
double[] values = { 1, 2, 9, 3 };
System.out.println( avg( values ) );
```

### Hinweis

Da Varargs als Arrays umgesetzt werden, sind überladene Varianten wie `avg(int... array)` und `avg(int[] array)`, also einmal mit einem Vararg und einmal mit einem Array, nicht möglich. Besser ist es hier, immer eine Variante mit Varargs zu nehmen, da sie mächtiger ist. Einige Autoren schreiben auch die Einstiegsmethode `main(String[] args)` mit variablen Argumenten, also `main(String... args)`. Das ist gültig, denn im Bytecode steht ja ein Array.



### Varargs-Designtipps \*

- ▶ Hat eine Methode nur einen Array-Parameter und steht er noch am Ende, so kann er relativ einfach durch ein Vararg ersetzt werden. Das gibt dem Aufrufer die komfortable Möglichkeit, eine kompaktere Syntax zu nutzen. Unsere `main(String[] args)`-Methode kann auch als `main(String... args)` deklariert werden, sodass der `main(...)`-Methode bei Tests einfach variable Argumente übergeben werden können.
- ▶ Muss eine Mindestanzahl von Argumenten garantiert werden – bei `max(...)` sollten das mindestens zwei sein –, ist es besser, eine Deklaration wie folgt zu nutzen: `max(int first, int second, int... remaining)`.
- ▶ Aus Performance-Gründen ist es empfehlenswert, Methoden mit häufigen Parameterlistengrößen als feste Methoden anzubieten, etwa `max(double, double)`, `max(double, double, double)` und dann `max(double...)`. Der Compiler wählt automatisch immer die passende Methode aus, für zwei oder drei Parameter sind keine temporären Array-Objekte nötig, und die automatische Speicherbereinigung muss nichts wegräumen.

### 4.1.13 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 Arrays initialisiert, aufgebaut und abgegrast werden.

### Mehrdimensionale Array-Objekte mit new aufbauen

Die folgende Zeile deklariert ein zweidimensionales Array mit Platz für insgesamt 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.



#### Tipp

Zwei alternative Deklarationen (die Position der eckigen Klammern ist verschoben) sind:

```
int A[][] = new int[ 4 ][ 8 ];
int[] A[] = new int[ 4 ][ 8 ];
```

Es ist zu empfehlen, alle eckigen Klammern hinter den Typ zu setzen.

### Anlegen und Initialisieren in einem Schritt

Ebenso wie bei eindimensionalen Arrays lassen sich mehrdimensionale Arrays 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 Array nicht unbedingt rechteckig sein muss. Dazu gleich mehr.

### Zugriff auf Elemente

Einzelne Elemente spricht der Ausdruck `A[i][j]` an.<sup>5</sup> Der Zugriff erfolgt mit so vielen Klammerpaaren, wie die Dimension des Arrays angibt.



#### Beispiel

Der Aufbau von zweidimensionalen Arrays (und der Zugriff auf sie) ist mit einer Matrix bzw. Tabelle vergleichbar. Dann lässt sich der Eintrag im Array `a[x][y]` in folgender Tabelle ablesen:

---

<sup>5</sup> 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.

```
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 Arrays

Nehmen wir eine Buchstabendefinition wie die folgende:

```
char[][] letter = { { ' ', '#', ' ' },
                    { '#', ' ', '#' },
                    { '#', ' ', '#' },
                    { '#', ' ', '#' },
                    { ' ', '#', ' ' } };
```

Dann können wir length auf zwei verschiedene Weisen anwenden:

- ▶ letter.length ergibt 5, denn es gibt fünf Zeilen.
- ▶ letter[0].length ergibt 3 – genauso wie letter[1].length usw. –, weil jedes Unter-Array die Größe 3 hat.

### Zweidimensionale Arrays mit ineinander verschachtelten Schleifen ablaufen

Um den Buchstaben unseres Beispiels auf dem Bildschirm auszugeben, nutzen wir zwei ineinander verschachtelte Schleifen:

```
for ( int line = 0; line < letter.length; line++ ) {
    for ( int column = 0; column < letter[line].length; column++ )
        System.out.print( letter[line][column] );
    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 nebeneinanderzulegen.

**Listing 4.7** src/main/java/com/tutego/insel/array/BinaryBanner.java

```
package com.tutego.insel.array;

import java.util.Scanner;

public class BinaryBanner {
```

```

static void printLetter( char[][] letter ) {
    for ( int column = 0; column < letter[0].length; column++ ) {
        for ( int line = letter.length - 1; line >= 0; line-- )
            System.out.print( letter[line][column] );
        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(char[][])` bekommt als Argument das zweidimensionale Array und läuft anders ab als im ersten Fall, um die Rotation zu realisieren. Mit der Eingabe »2« gibt es folgende Ausgabe:

Banner für 2 (binär 10):

```
#  
#####  
  
###  
# #  
###
```

#### 4.1.14 Nichtrechteckige Arrays \*

Da in Java mehrdimensionale Arrays als Arrays von Arrays implementiert sind, müssen diese Arrays nicht zwingend rechteckig sein. Jede Zeile im Array kann eine eigene Größe haben.

##### Beispiel

[zB]

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 Unter-Arrays

Wenn wir ein mehrdimensionales Array deklarieren, erzeugen versteckte Schleifen automatisch die inneren Arrays. Bei

```
int[][] a = new int[ 3 ][ 4 ];
```

erzeugt die Laufzeitumgebung die passenden Unter-Arrays automatisch. Dies ist bei

```
int[][] a = new int[ 3 ][];
```

nicht so. Hier müssen wir selbst die Unter-Arrays 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

Es gibt verschiedene Möglichkeiten, ein mehrdimensionales Array zu initialisieren:

```
int[][] A3x2 = { {1,2}, {2,3}, {3,4} };
```

oder

```
int[][] A3x2 = new int[][]{ {1,2}, {2,3}, {3,4} };
```

oder

```
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 4.8** 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 Array 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 4.9** src/main/java/com/tutego/insel/array/PascalsTriangle.java, main()

```

class PascalsTriangle {

    public static void main( String[] args ) {
        int[][] triangle = new int[7][];
        for ( int row = 0; row < triangle.length; row++ ) {
            System.out.print( new String( new char[(14 - row * 2)] ).replace( '\0', ' ' ) );
            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 `new String( new char[(14 - row * 2)] ).replace( '\0', ' ' )` produziert Einrückungen und greift eine fortgeschrittene API auf.  $(14 - \text{row} * 2)$  ist die Größe des standardmäßig mit 0 initialisierten Arrays, das dann in den Konstruktor der Klasse `String` übergeben wird, das wiederum ein `String`-Objekt aus dem `char`-Array aufbaut. Die `replace(...)`-Methode auf dem frischen `String`-Objekt führt wieder zu einem neuen `String`-Objekt, in dem alle `'\0'`-Werte durch Leerzeichen ersetzt sind.

### Andere Anwendungen

Mit zweidimensionalen Arrays 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 Adjazenzmatrizen<sup>6</sup> bei ungerichteten Graphen.

#### 4.1.15 Die Wahrheit über die Array-Initialisierung \*

So schön die kompakte Initialisierung der Array-Elemente 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 Primzahl-Array

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

---

<sup>6</sup> 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).

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;
```

Erst nach kurzem Überlegen wird das Ausmaß der Umsetzung sichtbar: Zunächst ist es der Speicherbedarf für die Methoden. Ist das Array `primes` in einer Methode deklariert und mit Werten initialisiert, kostet die Zuweisung Laufzeit, da wir viele Zugriffe haben, die auch alle schön durch die Indexüberprüfung 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 Array (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 Array umwandeln. Das ist eine sehr clevere Methode, Binärdaten einfach unterzubringen.

#### 4.1.16 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 4.10** src/main/java/com/tutego/insel/array/MultipleReturnValues.java, Ausschnitt

```
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 Array-Rückgaben 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. Selbst so etwas wie `int[] transposeMatrix(int[][] m)[]` ist möglich, wohl aber sinnvollerweise als `int[][] transposeMatrix(int[][] matrix)` geschrieben.

### 4.1.17 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()`.<sup>7</sup> Sie klonet – in unserem Fall kopiert – die Elemente des Array-Objekts in ein neues.

**Listing 4.11** src/main/java/com/tutego/insel/array/CloneDemo.java, Ausschnitt, 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-Arrays ist es wichtig, zu verstehen, dass die Kopie flach ist. Die Verweise aus dem ersten Array kopiert `clone()` in das neue Array, es klonet aber die referenzierten Objekte selbst nicht. Bei mehrdimensionalen Arrays wird also nur die erste Dimension kopiert, Unter-Arrays werden somit gemeinsam genutzt:

---

<sup>7</sup> Das ist gültig, da Arrays intern die Schnittstelle `Cloneable` implementieren.  
`System.out.println( new int[0] instanceof Cloneable );` gibt true zurück.

**Listing 4.12** src/main/java/com/tutego/insel/array/CloneDemo.java, Ausschnitt, 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 Arrays dasselbe Point-Objekt referenzieren; die Kopie ist flach, aber nicht tief.

#### 4.1.18 Array-Inhalte kopieren \*

Eine weitere nützliche statische Methode ist `System.arraycopy(...)`. Sie kann auf zwei Arten arbeiten:

- ▶ *Auf zwei schon existierenden Arrays.* Ein Teil eines Arrays wird in ein anderes Array kopiert. `arraycopy(...)` eignet sich dazu, sich vergrößernde Arrays zu implementieren, indem zunächst ein neues größeres Array angelegt wird und anschließend die alten Array-Inhalte in das neue Array kopiert werden.
- ▶ *Auf dem gleichen Array.* So lässt sich die Methode dazu verwenden, Elemente eines Arrays um bestimmte Positionen zu verschieben. Die Bereiche können sich durchaus überlappen.



#### Beispiel

Um zu zeigen, dass `arraycopy(...)` auch innerhalb des eigenen Arrays kopiert, sollen alle Elemente bis auf eines im Array 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` Einträge des Arrays `src` ab der Position `srcPos` in ein Array `dest` ab der Stelle `destPos`. Der Typ des Arrays ist egal, es muss nur in beiden Fällen der gleiche Typ sein. Die Methode arbeitet für große Arrays schneller als eine eigene Kopierschleife.

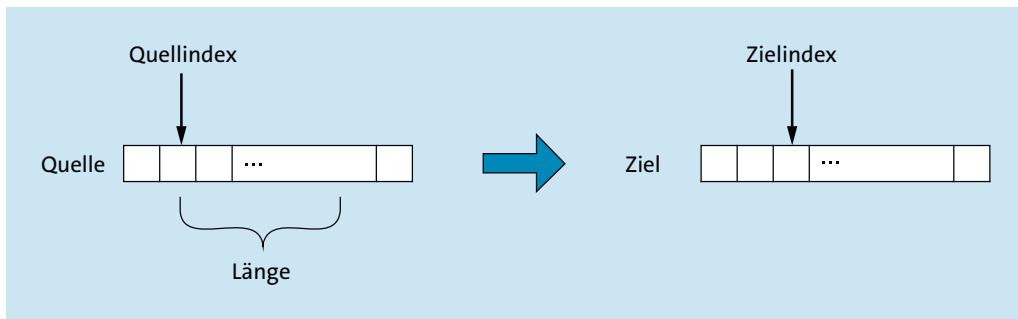


Abbildung 4.2 Kopieren der Elemente von einem Array in ein anderes

#### 4.1.19 Die Klasse Arrays zum Vergleichen, Füllen, Suchen, Sortieren nutzen

Die Klasse `java.util.Arrays` deklariert nützliche statische Methoden für den Umgang mit Arrays. So bietet sie Möglichkeiten zum Vergleichen, Sortieren und Füllen von Arrays sowie zur binären Suche.

#### String-Repräsentation eines Arrays

Nehmen wir an, wir haben es mit einem Array von Hundenamen zu tun, das wir auf dem Bildschirm ausgeben wollen:

##### Listing 4.13 DogArrayToString, main()

```
String[] dogs = {
    "Flocky Fluke", "Frizzi Faro", "Fanny Favorit", "Frosty Filius",
    "Face Flash", "Fame Friscco"
};
```

Soll der Array-Inhalt zum Testen auf den Bildschirm gebracht werden, so kommt eine Ausgabe mit `System.out.println(dogs)` nicht in Frage, 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 Arrays die gewünschte String-Repräsentation des Arrays:

```
System.out.println( Arrays.toString(dogs) ); // [Flocky Fluke, ...]
```

Das spart nicht unbedingt eine `for`-Schleife, die durch das Array läuft und auf jedem Element `printXXX(...)` aufruft, denn die Ausgabe ist immer von einem bestimmten Format, das mit [ beginnt, jedes Element mit Komma und einem Leerzeichen trennt und mit ] abschließt.

Die Klasse `Arrays` deklariert die `toString()`-Methode für unterschiedliche Array-Typen:

```
class java.util.Arrays
```

- `static String toString(XXX[] a)`  
Liefert eine String-Repräsentation des Arrays. Der Typ `XXX` steht stellvertretend für `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`.
- `static String toString(Object[] a)`  
Liefert eine String-Repräsentation des Arrays. Im Fall des Objekttyps ruft die Methode auf jedem Objekt im Array `toString()` auf.
- `static String deepToString(Object[] a)`  
Ruft auch auf jedem Unter-Array `Arrays.toString(...)` auf und nicht nur `toString()` wie bei jedem anderen Objekt.

### Sortieren

Diverse statische `Arrays.sort(...)`/`Arrays.parallelSort(...)`-Methoden ermöglichen das Sortieren von Elementen im Array. Bei primitiven Elementen (kein `boolean`) gibt es keine Probleme, da sie eine natürliche Ordnung haben.

[zB]

### Beispiel

Sortiere zwei Arrays:

```
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 Array 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 10, »Besondere Typen der Java SE«](#), beschreibt diese Möglichkeiten präzise.

```
class java.util.Arrays
```

- `static void sort(XXX[] a)`
- `static void sort(XXX[] a, int fromIndex, int toIndex)`  
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 in-

klusiv und `toIndex` exklusiv. Sind die Grenzen fehlerhaft, löst die Methode eine `IllegalArgument`-Exception (im Fall `fromIndex > toIndex`) oder eine `ArrayIndexOutOfBoundsException` (`fromIndex < 0` oder `toIndex > a.length`) aus.

- `static void sort(Object[] a)`
- `static void sort(Object[] a, int fromIndex, int toIndex)`  
Sortiert ein Array von Objekten. Die Elemente müssen Comparable implementieren.<sup>8</sup> Bei der Methode gibt es keinen generischen Typparameter, der das zur Übersetzungszeit erzwingt!
- `static <T> void sort(T[] a, Comparator<? super T> c)`
- `static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)`  
Sortiert ein Array von Objekten mit gegebenem Comparator.

### Paralleles Sortieren

Spezielle Sortiermethoden sind für sehr große Arrays gedacht. Bei den `parallelSort(...)`-Methoden verwendet die Bibliothek mehrere Threads, um Teile parallel zu sortieren, was die Geschwindigkeit erhöhen kann. Eine Garantie ist das aber nicht, denn ein Performance-Vorteil ergibt sich wirklich nur bei großen Arrays.

```
class java.util.Arrays
```

- `static void parallelSort(byte[] a)`
- `static void parallelSort(byte[] a, int fromIndex, int toIndex)`
- `static void parallelSort(char[] a)`
- `static void parallelSort(char[] a, int fromIndex, int toIndex)`
- `static void parallelSort(short[] a)`
- `static void parallelSort(short[] a, int fromIndex, int toIndex)`
- `static void parallelSort(int[] a)`
- `static void parallelSort(int[] a, int fromIndex, int toIndex)`
- `static void parallelSort(long[] a)`
- `static void parallelSort(long[] a, int fromIndex, int toIndex)`
- `static void parallelSort(float[] a)`
- `static void parallelSort(float[] a, int fromIndex, int toIndex)`
- `static void parallelSort(double[] a)`
- `static void parallelSort(double[] a, int fromIndex, int toIndex)`
- `static <T extends Comparable<? super T>> void parallelSort(T[] a)`

<sup>8</sup> Das stimmt nicht wirklich: Besteht das Array aus keinem oder nur einem Element, ist nichts zu vergleichen, also ist folglich auch nicht relevant, ob der Objekttyp Comparable ist.

- static <T extends Comparable<? super T>> void parallelSort(T[] a, int fromIndex, int toIndex)
- static <T> void parallelSort(T[] a, Comparator<? super T> c)
- static <T> void parallelSort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)

Der verwendete Algorithmus ist einfach zu verstehen: Zunächst wird das Array in Teil-Arrays partitioniert, diese werden parallel sortiert und dann zu einem größeren sortierten Array zusammengelegt. Das Verfahren nennt sich im Englischen auch *parallel sort-merge*.

### Arrays von Primitiven mit `Arrays.equals(...)` und `Arrays.deepEquals(...)` vergleichen \*

Die statischen Methoden `Arrays.equals(...)` vergleichen, ob zwei Arrays die gleichen Inhalte besitzen; dazu ist die überladene Methode für alle wichtigen Typen definiert. Wenn zwei Arrays 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`.



#### Beispiel

Vergleiche drei Arrays:

```
int[] array1 = { 1, 2, 3, 4 };
int[] array2 = { 1, 2, 3, 4 };
int[] array3 = { 9, 9, 2, 3, 9 };
System.out.println( Arrays.equals( array1, array2 ) );           // true
System.out.println( Arrays.equals( array2, 1, 3, array3, 2, 4 ) ); // true
```

Ein Vergleich von Teil-Arrays ist erst in Java 9 hinzugekommen.

Bei unterreferenzierten Arrays (`Arrays` zeigen auf `Arrays`) betrachtet `Arrays.equals(...)` das innere Array als einen Objektverweis und vergleicht es auch mit `equals(...)` – was jedoch bedeutet, dass nicht identische, aber mit gleichen Elementen referenzierte innere Arrays als ungleich betrachtet werden. Die statische Methode `deepEquals(...)` bezieht auch unterreferenzierte Arrays in den Vergleich ein.



#### 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 Arrays enthalten die gleichen Elemente, sind aber zwei unterschiedliche Objekte, also nicht identisch.



### Hinweis

`deepEquals(...)` vergleicht auch eindimensionale Arrays:

```
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 Arrays gleichen Typs und liefert `true`, wenn die Arrays gleich groß und Elemente paarweise gleich sind. `XXX` steht stellvertretend für `boolean, byte, char, int, short, long, double, float`.

- `static boolean equals(XXX[] a, int aFromIndex, int aToIndex, XXX[] b, int bFromIndex, int bToIndex)`

Vergleicht zwei Arrays, bleibt jedoch in den gewählten Ausschnitten. Diverse neue überlade-  
ne Methoden in Java 9.

### Objekt-Arrays mit `Arrays.equals(...)` und `Arrays.deepEquals(...)` vergleichen \*

Die `Arrays.equals(...)`-Methode kann auch Arrays mit beliebigen Objekten vergleichen, doch nutzt sie dann nicht die Identitätsprüfung per `==`, sondern die Gleichheit per `equals(...)`. Eine seit Java 9 hinzugekommene Methode fragt einen `Comparator`.



### Beispiel

Enthalten zwei String-Arrays die gleichen Wörter, wobei Groß-/Kleinschreibung keine Rolle spielt?

```
String[] words1 = { "Zufriedenheit", "übertrifft" , "Reichtum" };
String[] words2 = { "REICHTUM", "übertrifft" , "ZuFRIEDEnheit" };
Arrays.sort( words1, String.CASE_INSENSITIVE_ORDER );
Arrays.sort( words2, String.CASE_INSENSITIVE_ORDER );
System.out.println( Arrays.equals( words1, words2,
                                String.CASE_INSENSITIVE_ORDER ) );
```

```
class java.util.Arrays
```

- static boolean equals(Object[] a, Object[] a2)  
Vergleicht zwei Arrays mit Objektverweisen. Ein Objekt-Array 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 Arrays ebenso wie alle Unter-Arrays – rekursiv im Fall von Unter-Objekt-Arrays – gleich sind.
- static <T> boolean equals(T[] a, T[] a2, Comparator<? super T> cmp)  
Vergleicht zwei Arrays mit einem Comparator, und `cmp.compare(a[i], b2[i])` muss für alle Pärchen 0 sein, damit beide Elemente als gleich gelten. Die Angabe `<? super T>` können wir erst einmal ignorieren.
- static <T> boolean equals(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex, Comparator<? super T> cmp)  
Vergleicht Ausschnitte von Arrays mit einem Comparator.

### Unterschiede suchen mit mismatch (...)\*

Weiterhin gibt es die folgenden Methoden:

- int mismatch(XXX[] a, XXX[] b)
- int mismatch(XXX[] a, int aFromIndex, int aToIndex, XXX[] b, int bFromIndex, int bToIndex)

Sie geben den Index auf das erste Element zurück, das ungleich ist. Sind beide Arrays gleich, ist die Rückgabe -1.

Für Objekt-Array gibt es weiterhin:

- int mismatch(Object[] a, Object[] b)
- int mismatch(Object[] a, int aFromIndex, int aToIndex, Object[] b, int bFromIndex, int bToIndex)
- <T> int mismatch(T[] a, T[] b, Comparator<? super T> cmp)
- <T> int mismatch(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex, Comparator<? super T> cmp)

Die erste und zweite Methode nutzt direkt `equals(...)`, die dritte und vierte einen externen Comparator.

### Füllen von Arrays \*

`Arrays.fill(...)` füllt ein Array mit einem festen Wert. Der Start-End-Bereich lässt sich optional angeben.



### Beispiel

Fülle ein char-Array 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 Array. Mögliche Typen für **XXX** sind boolean, char, byte, short, int, long, double, float oder mit Object beliebige Objekte. Beim Bereich ist fromIndex inklusiv und toIndex exklusiv.

Neben der Möglichkeit, ein Array mit festen Werten zu füllen, gibt es außerdem die Methoden setAll(...)/parallelSetAll(...). Die Methoden durchlaufen ein gegebenes Array und rufen eine bestimmte Methode für jeden Index auf, die zur Initialisierung verwendet wird.



### Beispiel \*

Fülle ein double-Array mit Zufallszahlen:

```
double[] randoms = new double[ 10 ];
Arrays.setAll( randoms, v -> Math.random() );
System.out.println( Arrays.toString( randoms ) );
```

Das Beispiel nutzt Lambda-Ausdrücke, um die Funktion zu beschreiben. Wir kommen in [Kapitel 12, »Lambda-Ausdrücke und funktionale Programmierung«](#), auf diese Syntax noch einmal zurück.

```
class java.util.Arrays
```

- static void setAll(int[] array, IntUnaryOperator generator)
- static void setAll(double[] array, IntToDoubleFunction generator)
- static void setAll(long[] array, IntToLongFunction generator)
- static <T> void setAll(T[] array, IntFunction<? extends T> generator)
- static void parallelSetAll(double[] array, IntToDoubleFunction generator)
- static void parallelSetAll(int[] array, IntUnaryOperator generator)
- static void parallelSetAll(long[] array, IntToLongFunction generator)

- static <T> void parallelSetAll(T[] array, IntFunction<? extends T> generator)
 Läuft ein gegebenes Array komplett ab und übergibt dabei dem generator Schritt für Schritt den Index. Der Generator bildet den Index auf einen Wert ab, der wiederum zur Array-Initialisierung genutzt wird.

### Array-Abschnitte kopieren \*

Die Klasse `Arrays` bietet eine Reihe von `copyOf(...)`- und `copyOfRange(...)`-Methoden, die gegenüber `clone()` den Vorteil haben, dass sie auch Bereichsangaben erlauben und das neue Array größer machen können; im letzten Fall füllen die Methoden das Array 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 Array mit der gewünschten Größe bzw. dem angegebenen Bereich aus einem existierenden Array. 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 Ziel-Array 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 (siehe [Kapitel 11](#)), um den Typ flexibel zu halten. Zum einfacheren Verständnis können wir uns `T` als Typ `Object` vorstellen und das, was in spitzen Klammern steht, löschen.

### Halbierungssuche \*

Ist das Array sortiert, lässt sich mit `Arrays.binarySearch(...)` eine binäre Suche (Halbierungssuche) durchführen. Ist das Array nicht sortiert, ist das Ergebnis unvorhersehbar. Findet `binarySearch(...)` das Element, ist der Rückgabewert der Index der Fundstelle, andernfalls ist die Rückgabe negativ.

### Beispiel

[zB]

Suche ein Element im sortierten Array:

```
int[] numbers = { 1, 10, 100, 1000 };
System.out.println( Arrays.binarySearch( numbers, 100 ) ); // 2
```

Ist das Array nicht aufsteigend sortiert, ist ein falsches Ergebnis die Folge.

`binarySearch(...)` liefert in dem Fall, dass das Element nicht im Array ist, eine kodierte Position zurück, an der das Element eingefügt werden könnte. Damit der Index nicht mit einer

normalen Position einer Fundstelle kollidiert, die immer  $\geq 0$  ist, ist die Rückgabe negativ und als **-Einfügeposition - 1** kodiert.



### Beispiel

Das Element 101 ist nicht im Array:

```
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 Array: 1 (Position 0), 10 (Position 1), 100 (Position 2), 101 (Position 3).



### Tipp

Da das Array 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 Objekt-Array. 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 11, »Generics<T>«](#)) etwas schwieriger. Wir werden in [Kapitel 17, »Einführung in die Generics«](#) mehr darüber erfahren.

rung 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 10](#), »Besondere Typen der Java SE«, genau klären.

### Lexikografische Array-Vergleiche mit `compare(...)` und `compareUnsigned(...)`

Diverse in Java 9 eingeführte `int compareXXX(XXX[] a, XXX[] b)`-Methoden gehen die Arrays ab und testen alle Paare auf ihre Ordnung. Es gibt die von `Comparator` bekannte Rückgabe: Ist jedes `a[i] == b[i]`, ist die Rückgabe »0«. Ist in der Abfragefolge ein `a[i]` kleiner als `b[i]`, dann ist die Rückgabe negativ; ist ein `a[i]` größer als `b[i]`, ist die Rückgabe positiv. Die Methode ist überladen mit einer Variante, die einen Bereich im Array auswählt: `compare(XXX[] a, int aFromIndex, int aToIndex, XXX[] b, int bFromIndex, int bToIndex)`. Für `byte`, `short`, `int` und `long` gibt es weiterhin eine Vergleichsmethode ohne Vorzeichen über den gesamten Wertebereich:

- ▶ `int compareUnsigned(XXX[] a, XXX[] b)`
- ▶ `int compareUnsigned(XXX[] a, int aFromIndex, int aToIndex, XXX[] b, int bFromIndex, int bToIndex)`

Für Objekte gibt es eigene Methoden:

- `static <T extends Comparable<? super T>> int compare(T[] a, T[] b)`  
Vergleiche zwei Objekt-Arrays, wobei der `Comparator` die Gleichheit der Objektpaare feststellt.
- `static <T extends Comparable<? super T>> int compare(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex)`  
Vergleicht Ausschnitte.
- `static <T> int compare(T[] a, T[] b, Comparator<? super T> cmp)`
- `static <T> int compare(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex, Comparator<? super T> cmp)`  
Vergleicht mithilfe eines externen `Comparator`-Objekts.

### Arrays zu Listen mit `Arrays.asList(...)` – praktisch für die Suche und zum Vergleichen \*

Ist das Array 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 Arrays möglich, für die `Arrays` bisher keine Methoden definierte.



### Beispiel

Teste, ob auf der Kommandozeile der Schalter -? gesetzt ist. Die auf der Kommandozeile übergebenen Argumente übergibt die Laufzeitumgebung als String-Array an die main(String[] args)-Methode:

```
if ( Arrays.asList( args ).contains( "-?" ) )
    ...
}
```



### Beispiel

Teste, ob Teile zweier Arrays gleich sind:

```
// Index      0           1           2
String[] a = { "Asus",      "Elitelgroup", "MSI" };
String[] b = { "Elitelgroup", "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 Array vom Typ T.

Die statische Methode asList(...) nimmt über das Vararg entweder ein Array von Objekten (kein primitives Array!) 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(...) übergebenes primitives Array 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]

```

Der Grund ist einfach: `Arrays.asList(...)` erkennt `nums` nicht als Array von Objekten, sondern als genau ein Element einer Aufzählung. So setzt die statische Methode das Array mit Primitiven als ein Element in die Liste, und die Objektmethode `toString()` eines `java.util.List`-Objekts ruft lediglich auf dem Array-Objekt `toString()` auf, was die kryptische Ausgabe zeigt.

### Parallele Berechnung von Präfixen \*

Stehen mehrere Prozessoren oder Kerne zur Verfügung, können bestimmte Berechnungen bei Arrays parallelisiert werden. Ein Algorithmus nennt sich *parallele Präfix-Berechnung* und basiert auf der Idee, dass eine assoziative Funktion – nennen wir sie  $f$  – auf eine bestimmte Art und Weise auf Elemente eines Arrays – nennen wir es  $a$  – angewendet wird, nämlich so:

- ▶  $a[0]$
- ▶  $f(a[0], a[1])$
- ▶  $f(a[0], f(a[1], a[2]))$
- ▶  $f(a[0], f(a[1], f(a[2], a[3])))$
- ▶ ...
- ▶  $f(a[0], f(a[1], \dots f(a[n-2], a[n-1])\dots))$

In der Aufzählung sieht das etwas verwirrend aus, daher soll ein praktisches Beispiel das Verständnis erleichtern. Das Array sei  $[1, 3, 0, 4]$  und die binäre Funktion die Addition.

Index	Funktion	Ergebnis
0	$a[0]$	1
1	$a[0] + a[1]$	$1 + 3 = 4$
2	$a[0] + (a[1] + a[2])$	$1 + (3 + 0) = 4$
3	$a[0] + (a[1] + (a[2] + a[3]))$	$1 + (3 + (0 + 4)) = 8$

Tabelle 4.2 Präfix-Berechnung des Arrays  $[1, 3, 0, 4]$  mit Additionsfunktion

Auf den ersten Blick wirkt das wenig spannend, doch kann der Algorithmus parallelisiert werden und somit im besten Fall in logarithmischer Zeit (mit  $n$  Prozessoren) gelöst werden. Voraussetzung dafür ist allerdings eine assoziative Funktion, wie Summe und Maximum. Ohne ins Detail zu gehen, könnten wir uns vorstellen, dass ein Prozessor/Kern  $0 + 4$  berechnet, ein anderer zeitgleich  $1 + 3$  und dass dann das Ergebnis zusammengezählt wird.



### Beispiel \*

Das Beispiel unserer Präfix-Berechnung mithilfe einer Methode aus Arrays:

```
int[] array = {1, 3, 0, 4};
Arrays.parallelPrefix( array, (a, b) -> a + b );
System.out.println( Arrays.toString( array ) ); // [1, 4, 4, 8]
```

Die Implementierung nutzt die fortgeschrittene Syntax der Lambda-Ausdrücke.

Statt  $(a + b) \rightarrow a + b$  verkürzt es `Integer::sum` sogar noch.

Ein weiteres Beispiel: Finde das Maximum in einer Menge von Fließkommazahlen:

```
double[] array = {4.8, 12.4, -0.7, 3.8 };
Arrays.parallelPrefix( array, Double::max );
System.out.println( array[array.length -1] ); // 12.4
```

Das Beispiel nutzt schon die Methode, die `Arrays` für die parallele Präfix-Berechnung bietet:

#### `class java.util.Arrays`

- `static void parallelPrefix(int[] array, IntBinaryOperator op)`
- `static void parallelPrefix(int[] array, int fromIndex, int toIndex, IntBinaryOperator op)`
- `static void parallelPrefix(long[] array, LongBinaryOperator op)`
- `static void parallelPrefix(long[] array, int fromIndex, int toIndex, LongBinaryOperator op)`
- `static void parallelPrefix(double[] array, DoubleBinaryOperator op)`
- `static void parallelPrefix(double[] array, int fromIndex, int toIndex, DoubleBinaryOperator op)`
- `static <T> void parallelPrefix(T[] array, BinaryOperator<T> op)`
- `static <T> void parallelPrefix(T[] array, int fromIndex, int toIndex, BinaryOperator<T> op)`

#### 4.1.20 Eine lange Schlange

Das neu erworbene Wissen über Arrays 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 Array 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 Schlangenpunkte, ist das Spiel verloren. Ist bei der Bildschirmausgabe eine Koordinate gleich einem der Schlangenpunkte, zeichnen wir 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 Array mit den fünf Punkten immer nur die letzten fünf 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 Arrays 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 4.14** src/main/java/com/tutego/insel/array/LongerZZZZZnake.java

```
package com.tutego.insel.array;

import java.awt.Point;
import java.util.Arrays;

public class LongerZZZZZnake {

    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() ) {
        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
}
}
```

Wer wieder etwas vorarbeiten möchte, kann Folgendes tun:

- ▶ Ersetze das Array durch eine dynamische Datenstruktur vom Typ `ArrayList<Point>`.
- ▶ Nach jedem zweiten Schritt vom Benutzer soll die Länge der Schläge um eins wachsen.
- ▶ Wenn der Spieler ein Goldstück einsammelt, soll die Länge der Schlange um eins schrumpfen.

## 4.2 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.

### 4.2.1 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 diese Methode `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 ein 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 nutzt so genannte 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 syntaktisch gültig, aber eben keiner, den 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.

## 4.2.2 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 4.15** src/main/java/com/tutego/insel/array/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 com.tutego.insel.array.LovesGoldenHamster Raphael Perly Mirjam Paul
```

### Bibliothek

Zum Parsen der Kommandozeilenargumente bieten sich zum Beispiel die Bibliotheken *Commons CLI* (<http://commons.apache.org/proper/commons-cli>) oder *args4j* (<http://args4j.kohsuke.org>) an.

### 4.2.3 Der Rückgabetyp von main(...) und System.exit(int) \*

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 Plattformunabhängigkeit 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(int)` 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 `cmd.exe` von Windows unter `%ERRORLEVEL%`, einer Art dynamischer Umgebungsvariablen.

Dazu ein Beispiel; ein Java-Programm liefert den Statuswert 42:

**Listing 4.16** src/main/java/com/tutego/insel/array/SystemExitDemo.java

```
package com.tutego.insel.array;

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 4.17** showreturn.bat

```
@echo off
cd target\classes
java com.tutego.insel.array.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(int)`, was zu einer Belegung der Variablen `%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 des kleinen Skripts:

```
>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%` zwischenzuspeichern, etwa so:

**Listing 4.18** showreturn2.bat

```
@echo off
cd target\classes
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 0 zeigt einen Fehler an. Also ist der Rückgabewert beim normalen fehlerfreien Verlassen 0. Eine SecurityException wird ausgelöst, falls der aktuelle SecurityManager dem aufrufenden Code nicht erlaubt, die JVM zu beenden. Das gilt insbesondere bei Programmen in einem Container, wie einem Webserver.

### 4.3 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 5

## Der Umgang mit Zeichenketten

»*Ohne Unterschied macht Gleichheit keinen Spaß.*«

– Dieter Hildebrandt (1927–2013)

### 5.1 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.

#### 5.1.1 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*) heißt. Es gibt 94 druckbare Zeichen (Buchstaben, Ziffern, Interpunktionszeichen), 33 nicht druckbare Steuerzeichen (etwa den Tabulator und viele andere Zeichen, die bei Fernschreibern nützlich waren, aber heute uninteressant sind) und das Leerzeichen, das nicht als Steuerzeichen zählt. Am Anfang des ASCII-Alphabets stehen an den Positionen 0–31 Steuerzeichen, an Stelle 32 folgt das Leerzeichen, und anschließend kommen alle druckbaren Zeichen. An der letzten Position, 127, schließt ein Steuerzeichen (»delete«, früher »rubout«) die ASCII-Tabelle ab.

Die Tabelle in Abbildung 5.1 stammt aus dem Originalstandard von 1968 und gibt einen Überblick über die Position der Zeichen.

USASCII code chart							
b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	
b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	Column →
b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	Row ↓
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	2	2	2	2
0	0	1	1	3	3	3	3
0	1	0	0	4	4	4	4
0	1	0	1	5	5	5	5
0	1	1	0	6	6	6	6
0	1	1	1	7	7	7	7
1	0	0	0	8	8	8	8
1	0	0	1	9	9	9	9
1	0	1	0	10	10	10	10
1	0	1	1	11	11	11	11
1	1	0	0	12	FF	FF	FF
1	1	0	1	13	CR	GS	GS
1	1	1	0	14	SO	RS	RS
1	1	1	1	15	SI	US	US

Abbildung 5.1 Die ursprüngliche ASCII-Tabelle

### 5.1.2 ISO/IEC 8859-1

An dem ASCII-Standard gab es zwischendurch Aktualisierungen, sodass einige Steuerzeichen 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 usw. 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 weite Teile der Welt das lateinische Alphabet nutzen, sollte diese Kodierung natürlich alle diese 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 Standard *ISO/IEC 8859-1*, der 191 Zeichen beschreibt, was die Kodierung für eine große Anzahl von Sprachen wie Deutsch, Spanisch, Französisch, Afrikaans nützlich macht. 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 Steuerzeichen – etwa der Tabulator oder das Zeilenumbruchzeichen – *nicht* dazugehören. Von den 256 möglichen Positionen bleiben 65 Stellen frei. Das sind die Stellen 0 bis 31 sowie 127 von den ASCII-Steuerzeichen 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 Steuerzeichen 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 Steuerzeichen (wie Padding, Start einer Selektion, kein Umbruch) sind.

### Windows-1252 \*

Weil die Zeichen an der Stelle 128 bis 159 uninteressante Steuerzeichen 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 Neubelegung 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.

### 5.1.3 Unicode

Obwohl Latin-1 für die »großen« Sprachen alle Zeichen mitbrachte, fehlen Details, wie Œ, œ, ū, Ÿ für das Ungarische, das komplette griechische Alphabet, die kyrillischen Buchstaben, chinesische und japanische Zeichen, indische Schrift, mathematische Zeichen und vieles mehr. Um das Problem zu lösen, bildete sich das Unicode-Konsortium, um jedes elementare Zeichen der Welt zu kodieren und ihm einen eindeutigen Codepoint zu geben. Unicode enthält alle Zeichen aus ISO 8859-1, was die Konvertierung von vielen 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, Unicode 9, umfasst 135 Schriftsysteme und über 120.000 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. Prinzipiell umfasst der Bereich 1.114.112 mögliche Codepunkte, von U+0000 bis U+10FFFF.

### 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 U+00C7) und der spanischen Schrift (»ñ« mit Tilde unter U+00F1), und bei ALLG. INTERPUNKTIONSZEICHEN findet sich das umgedrehte (invertierte) Fragezeichen bei U+00BF.

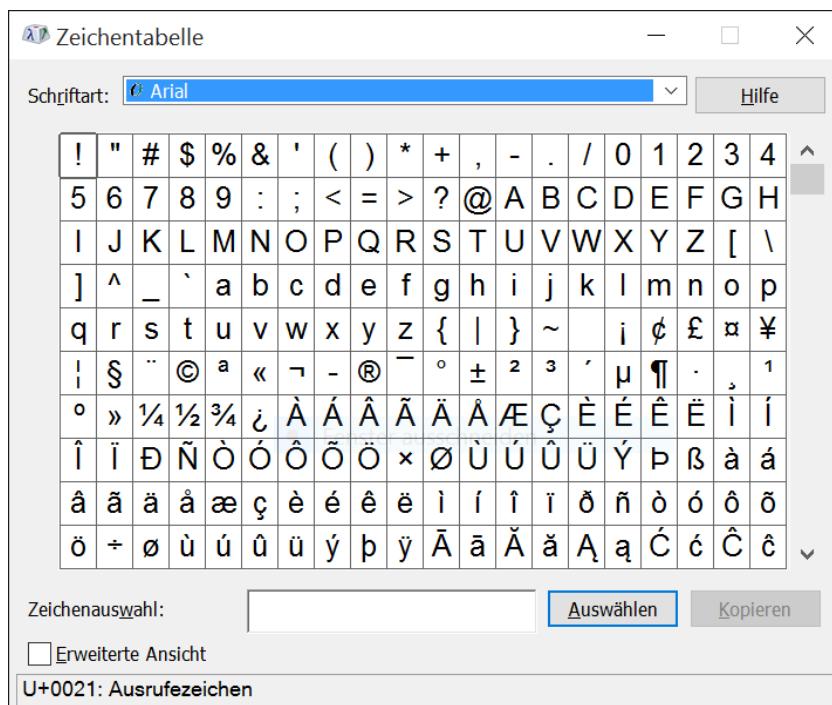


Abbildung 5.2 Programm Zeichentabelle unter Windows

### Anzeige der Unicode-Zeichen \*

Der Unicode-Standard definiert nur die Position von Zeichen, aber nicht wie sie grafisch aussehen. Die grafische Darstellung eines Zeichens nennt sich *Glyphe*, auch *Schriftzeichen* genannt. *Schriftarten* wie Times oder Arial enthalten Darstellungen für die Glyphen. Doch unterschiedliche Schriftarten enthalten nur Teilmengen aller Unicode-Zeichen; viele freie Schriftarten enthalten zum Beispiel überhaupt keine asiatischen Zeichen.

Auch 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 bzw. 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 verbesserungswürdig. Es wird Aufgabe der Betriebssystementwickler bleiben, dies zu ändern.<sup>1</sup>

### 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.



#### 5.1.4 Unicode-Zeichenkodierung

Da es im aktuellen Unicode-Standard 9 mehr als 120.000 Zeichen gibt, werden zur Kodierung eines Zeichens 4 Byte bzw. 32 Bit verwendet. Ein Dokument, das Unicode-Zeichen enthält, besitzt dann einen Speicherbedarf von  $4 \times (\text{Anzahl der Zeichen})$ . Wenn die Zeichen auf diese Weise kodiert werden, sprechen wir von einer *UTF-32-Kodierung*.

Für die meisten Texte ist UTF-32 reine Verschwendungen, 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 Unicode-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 Beispiel 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.

Glyph	A	ß	東	ð
Unicode-Codepoint	U+0041	U+00DF	U+6771	U+10400
UTF-32	00000041	000000DF	00006771	00010400
UTF-16	0041	00DF	6771	D801 DC00
UTF-8	41	C3 9F	E6 9D B1	F0 90 90 80

Tabelle 5.1 Zeichenkodierung in den verschiedenen Unicode-Versionen

<sup>1</sup> 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, so dass auch dort die Sonderzeichen erscheinen.

Werden Texte ausgetauscht, sind sie 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.

### 5.1.5 Escape-Sequenzen/Fluchtsymbole

Um spezielle Zeichen, etwa den Zeilenumbruch oder Tabulator, in einen String oder char setzen zu können, stehen Escape-Sequenzen<sup>2</sup> (zu Deutsch: Fluchtsymbole) 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 5.2 Escape-Sequenzen

#### 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 lispt wie Katja Burkard?\"";
String filename = "C:\\Dokumente\\Siemens\\Schweigegeld.doc";
```

<sup>2</sup> 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).

### 5.1.6 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.

#### Unicode-Escapes

Beliebige Unicode-Zeichen lassen sich für den Compiler über *Unicode-Escapes* schreiben. Im Quellcode steht dann \uxxxx, wobei x eine hexadezimale Ziffer ist – also 0...9, A...F (bzw. 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 5.3 Deutsche Sonderzeichen in Unicode

#### 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.

#### Beispiel

Zeige Pi und in einem GUI-Dialog einen Grinsemann:

```
System.out.println( "Pi: \u03c0" ); // Pi: π
javax.swing.JOptionPane.showMessageDialog( null, "\u263a" );
```

Der beliebte Smiley 😊 ist als Unicode unter \u263A (WHITE SMILING FACE) bzw. unter \u2639 (WHITE FROWNING FACE) 😆 definiert. Das Euro-Zeichen € ist unter \u20ac zu finden.

Die Unicode-Escape-Sequenzen sind also an beliebiger Stelle erlaubt, aber wirklich nützlich ist das eher für Zeichenketten. Im Prinzip ist es kein Problem, Bezeichner aus dem erlaubten Unicode-Alphabet zu nutzen.



### Beispiel

Deklariere und initialisiere eine Variable  $\pi$ :

```
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 {}
```

Wichtig zu verstehen ist, dass so etwas wie \u000d im Code identisch mit einem Zeilenumbruch ist. Den Versuch, im Java-Quellcode eine Zeichenkette "\u000d" unterzubringen, bestraft der Compiler mit einem Fehler, denn es wäre identisch mit

```
"
```

```
"
```

`System.out.println("\u0009");` klappt aber durchaus und gibt einen Tabulator aus. Das Unicode-Zeichen \uffff ist nicht definiert und kann bei Zeichenketten als Ende-Symbol verwendet werden.

### Enkodierung vom Quellcode festlegen \*

Enthält Quellcode Nicht-ASCII-Zeichen (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, denn standardmäßig liest der Compiler den Quellcode in der Kodierung ein, die er selbst hat, denn an einer Textdatei ist die Kodierung nicht abzulesen. Liegt der Quellcode in einem anderen Format vor, konvertiert ihn der Compiler unbeabsichtigt in ein falsches, nicht entsprechendes Unicode-Format. Um das Problem zu lösen, gibt es folgenden Ansatz: 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 <https://docs.oracle.com/en/java/javase/11/tools/javac.html>. Wenn allerdings unterschiedliche Dateien in unterschiedlichen Formaten vorliegen, dann hilft die globale Einstellung nichts.

### 5.1.7 Java-Versionen gehen mit Unicode-Standard Hand in Hand \*

In den letzten Jahren hat sich der Unicode-Standard erweitert, und Java ist den Erweiterungen gefolgt:

Java-Version	Unicode-Version	Anzahl Zeichen
1.0	1.1.5	34.233
1.1	2.0	38.950
1.1.7, 1.2, 1.3	2.1	38.952
1.4	3.0	49.259
5, 6	4.0	96.447
7	6.0	109.242
8	6.2	110.182
9	8	120.737
11	10	136.690

Tabelle 5.4 Java-Versionen und ihr unterstützter Unicode-Standard

Die Java-Versionen von 1.0 bis 1.4 nutzen einen Unicode-Standard, 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 aus dem Bereich U+0000 bis U+FFFF. Der Bereich heißt auch *BMP (Basic Multilingual Plane)*. Java 5 unterstützte erstmalig den Unicode-4.0-Standard, der 32 Bit (also 4 Byte) für die Abbildung eines Zeichens nötig macht. Doch mit dem Wechsel auf Unicode 4 wurde nicht die interne Länge für ein char-Zeichen angehoben, sondern es bleibt dabei, dass ein char 2 Byte groß ist und der Bereich 0x0 bis 0xFFFF ist. Das heißt aber auch, dass Zeichen, die größer als 65.536 sind, irgendwie anders kodiert werden müssen. Der Trick ist, ein »großes« Unicode-Zeichen aus zwei chars zusammenzusetzen. Dieses Pärchen aus zwei 16-Bit-Zeichen heißt *Surrogate-Paar*. Sie bilden in der *UTF-16-Kodierung* ein Unicode-4.0-Zeichen. Diese Surrogate-Paare vergrößern den Bereich der Basic Multilingual Plane. Hätten wir char als echtes Objekt, könnte es wohl besser intern verstecken, wie viele Bytes die Speicherung wirklich benötigt. Aber Java musste ja unbedingt primitive Datentypen haben ...

Mit der Einführung von Unicode 4 unter Java 5 gab es an den Klassen für Zeichen- und Zeichenkettenverarbeitung 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 den höheren Bereichen, etwa für die phönizische Schrift, die im Unicode-Block U+10900 bis U+1091F liegt – also kurz hinter 65.536, 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 diese 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.

## 5.2 Die Character-Klasse

`char` ist ein primitiver Datentyp und besitzt keine Methoden. Daher gibt es im Kernpaket `java.lang` die Klasse `Character` mit einer großen Anzahl Methoden, die im Umgang mit einzelnen Zeichen interessant sind. Die Methoden sind in der Regel statisch.

Dazu gehören Methoden zum Testen, etwa ob ein Zeichen eine Ziffer, ein Buchstabe oder ein Sonderzeichen ist.

### 5.2.1 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
<code>Character.isDigit( '0' )</code>	true
<code>Character.isDigit( '-' )</code>	false

Tabelle 5.5 Ergebnisse einiger `isXXX()`-Methoden

Ausdruck	Ergebnis
Character.isLetter( 'ß' )	true
Character.isLetter( '0' )	false
Character.isWhitespace( ' ' )	true
Character.isWhitespace( '-' )	false

Tabelle 5.5 Ergebnisse einiger isXXX()-Methoden (Forts.)

Alle diese Methoden »wissen« über die Eigenschaften der einzelnen Unicode-Zeichen Bescheid. Und der Codepoint jedes Unicode-Zeichens ist immer der gleiche, egal, ob ein Programm in Deutschland oder in der Mongolei ausgeführt wird.

### Hinweis

Mit »letter« ist nicht nur ein bekannter Buchstabe wie »a« oder »?« gemeint. Unicode enthält über hunderttausend Zeichen, davon hunderte von Buchstaben und Ziffern.



### 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 5.1 src/main/java/com/tutego/insel/string/IsNumeric.java, Ausschnitt

```
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 Einsatzgebiete.

Das Beispiel nutzt zwei String-Methoden: `length()` liefert die Länge eines Strings, und `charAt(int)` 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)`
- `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 so genannter *Weißraum*<sup>3</sup> (engl. *white space*), auch *Leerraum* genannt?

<sup>3</sup> Es wird Weißraum genannt, weil das ausgegebene Zeichen den Raum in der Regel weiß lässt, aber die Position der Ausgabe dennoch fortschreitet.

### Wortwahl

Ein *Leerzeichen* ist in diesem Buch nur das Zeichen » «, das die *Leertaste* generiert. Es hat den ASCII-Code 32, in Unicode \u0020. Der Begriff *Weißraum* steht dafür für alles das, was Leer-  
raum schafft, also Leerzeichen, Tabulator, Return usw.

## 5.2.2 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(int)`:

**Listing 5.2** src/main/java/com/tutego/insel/string/UppercaseWriter.java, Ausschnitt

```
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 Weißraum 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 liefern den passenden Groß- bzw. 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 `"ß".toUpperCase()`, die das Ergebnis »SS« liefert; einen String, der um eins verlängert ist.

### 5.2.3 Vom Zeichen zum String

Um ein Unicode-Zeichen in einen String zu konvertieren, können wir die statische überladene `String`-Methode `valueOf(char)` nutzen. Eine vergleichbare Methode gibt es auch in `Character`, und zwar die statische Methode `toString(char)`. Beide Methoden haben die Einschränkung, dass das Unicode-Zeichen nur 2 Byte lang sein kann. `String` deklariert dafür auch `valueOfCodePoint(int)`. So eine Methode fehlte bisher in `Character`; erst in Java 11 ist `toString(int)` eingezogen; intern delegiert sie an `valueOfCodePoint(int)`.

### 5.2.4 Von char in int: vom Zeichen zur Zahl \*

Wenn Zeichen aus Benutzereingaben stammen, stellt sich die Anforderung, sie in die Zahl zu konvertieren. Aus der Ziffer '5' soll der numerische Wert 5 werden. Nach alter Hacker-Tradition war die Lösung immer die, eine '0' abzuziehen. Die ASCII-Null '0' hat den `char`-Wert 48, '1' dann 49, bis '9' schließlich 57 erreicht. So ist logischerweise  $'5' - '0' = 53 - 48 = 5$ . Die Lösung hat den Nachteil, dass sie nur für ASCII-Ziffern funktioniert.

Eine ordentliche Java-Lösung besteht zum Beispiel darin, ein `char` in einen `String` zu konvertieren und diesen über eine `Integer`-Methode zu konvertieren, etwa so:

```
char c = '5';
int i = Integer.parseInt( String.valueOf(c) ); // 5
```

Die `parseInt(...)`-Methode ist voll internationalisiert und konvertiert ebenso Dezimalzahlen aus anderen Schriftsystemen, etwa Hindi/Sanskrit:

```
System.out.println( Integer.parseInt( "᳚" ) ); // 5
```

Das funktioniert, ist jedoch für einzelne Zeichen nicht besonders effizient in Schleifen. Es gibt zwei andere Möglichkeiten, mit statischen Methoden aus der Klasse `Character`.

### getNumericValue(...)-Methode

Die Character-Methode `getNumericValue(char)` liefert den numerischen Wert einer Ziffer zurück; natürlich arbeitet die Methode wieder internationalisiert:

```
int i = Character.getNumericValue( '5' );
System.out.println( i );                                // 5
System.out.println( Integer.parseInt( "\u0045" ) );    // 5
```

Die Methode ist viel leistungsfähiger, denn sie kennt den tatsächlichen »Wert« aller Unicode-Zeichen, zum Beispiel auch der römischen Ziffern I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, L, C, D, M ..., die im Unicode-Alphabet ab '\u2160' stehen:

```
System.out.println( Character.getNumericValue( '\u216f' ) ); // 1000
```

Das kann `Integer.parseInt(...)` nicht verarbeiten; eine Ausnahme ist die Folge.

### XXXdigit(...)-Methoden

Die Character-Klasse besitzt ebenso eine Umwandlungsmethode für Ziffern bezüglich einer beliebigen Basis, das auch in die andere Richtung.

```
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«.

#### Beispiel

Konvertiere eine Zeichenkette mit Ziffern in eine Ganzzahl.

```
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.



### 5.3 Zeichenfolgen

Eine Zeichenfolge 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.

Zeichenfolgen lassen sich als `char`-Arrays aufbauen, doch Arrays sind zu unflexibel. Daher sieht Java drei zentrale Klassen vor, die Zeichenfolgen komfortabel verwalten. Sie unterscheiden sich in zwei Punkten:

- ▶ Sind die Zeichenfolgen unveränderbar (*immutable*), dann können sie nach dem Anlegen später nicht mehr verändert werden. Oder sollen die Zeichenfolgen dauerhaft veränderbar (*mutable*) sein?
- ▶ 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 5.6 Drei Klassen, die Zeichenfolgen verwalten

#### Die Klasse `String`

Die Klasse `String` repräsentiert nicht änderbare Zeichenketten (allgemein heißen Objekte, deren Zustand sich nicht verändern lässt, *immutable*). Daher ist ein `String` immer threadsicher, denn eine Synchronisation ist nur dann nötig, wenn es Änderungen am Zustand 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 zusammengefügten Zeichenreihe.



#### 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 Anweisungen den Zustand eines Objekts modifizieren, etwa indem das erste Zeichen gelöscht wird. Die Referenz umzubiegen, verändert keinen Zustand.

## 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.

## Der Basistyp `CharSequence` für Zeichenketten

`CharSequence` ist die gemeinsame Schnittstelle von `String`, `StringBuilder` und `StringBuffer` und wird in der Bibliothek mehrfach verwendet. Nehmen wir ein Beispiel: Die Klasse `String` deklariert eine Methode `contains(CharSequence s)`, die testet, ob der Teil-`String` `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.

## Enthält ein `String` ein char-Array? \*

Die drei Klassen `String`, `StringBuilder`, `StringBuffer` entsprechen der idealen Umsetzung der objektorientierten Idee (mit der wir uns in [Kapitel 6](#), »Eigene Klassen schreiben«, intensiv auseinandersetzen): Wie genau die Zeichenfolgen gespeichert werden, dringt nicht nach außen. Interessanterweise ist ab Java 9 ein Wechsel vollzogen worden: Vor Java 9 speicherte immer ein char-Array alle Zeichen eines `StringBuilder`/`StringBuffer`/`String`-Objekts. In Java 9 gibt es eine Option auf eine *String-Verdichtung*,<sup>4</sup> sodass ein byte-Array verwendet wird, das den String entweder in der Latin-1- oder in UTF-16-Kodierung enthält, wobei im ausschließlichen Fall der Latin-1-Zeichen ein Byte verwendet wird und so der Speicherbedarf auf die Hälfte sinkt.

Das Schöne ist also, dass die Klassen uns die lästige Arbeit abnehmen, selbst Zeichenfolgen in Arrays zu verwalten, und wir von den ganzen internen Optimierungen nichts mitbekommen.

---

<sup>4</sup> Beschrieben in <http://openjdk.java.net/jeps/254>.

## 5.4 Die Klasse String und ihre Methoden

Die Entwickler von Java haben eine Symbiose zwischen `String` als Klasse und dem `String` als eingebautem Datentyp gebildet. 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.

Die Klasse `String` bietet mit `+` einen überladenen Operator, es gibt genau eine Konstante `CASE_INSENSITIVE_ORDER` vom Typ `Comparator<String>`, ein paar Konstruktoren, statische Methoden sowie Objektmethoden – diese schauen wir uns im Folgenden an.

### 5.4.1 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

`"Charlie Hebdo".length()` liefert die Länge der Zeichenkette. Das Ergebnis ist 13. Weißraum 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("String")` ist unsinnig; `s = "String"` ist korrekt.

### 5.4.2 Konkatenation mit `+`

Mehrere Beispiele haben schon gezeigt, dass `Strings` mit `+` konkateniert werden können.



### Beispiel

Haben die Glieder bei der Konkatenation unterschiedliche Datentypen und ist einer String, werden diese automatisch auf String gebracht:

```
int    age    = 39;
double height = 1.83;
String s = "Alter: " + age + ", Größe: " + height;
System.out.println( s );      // Alter: 39, Größe: 1.83
```



### 5.4.3 String-Länge und Test auf Leer-String

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 sie so implementiert, dass sie die Anzahl der Zeichen im String (die Länge des Strings) zurückgibt. Um herauszufinden, ob der String keine Zeichen hat, lässt sich neben `length() == 0` auch die Methode `isEmpty()` nutzen. In Java 11 ist die Methode `isBlank()` hinzugekommen, die testet, ob der String leer ist oder nur aus Weißraum besteht; Weißraum ist jedes Zeichen, bei dem `Character.isWhitespace(int)` wahr anzeigt.

Anweisung	Ergebnis
<code>"".length()</code>	0
<code>"".isEmpty()</code>	true
<code>" ".length()</code>	1
<code>" ".isEmpty()</code>	false
<code>" ".isBlank()</code>	true
<code>String s = null; s.length();</code>	NullPointerException



Tabelle 5.7 Ergebnisse der Methoden `length()`, `isEmpty()` und `isBlank()`

### Hinweis

Bei Arrays sind wir daran gewöhnt, dass `length` ein Attribut ist. Bei String ist `length()` eine Methode, benötigt also ein paar runde Klammern.

### Eine praktische Hilfsmethode: `isNullOrEmpty(String)`

Während das .NET-Framework etwa die statische Member-Funktion `IsNullOrEmpty(String)` anbietet, die testet, ob die übergebene Referenz `null` oder die Zeichenkette leer ist, muss das in Java getrennt getestet werden. Hier ist eine eigene statische Utility-Methode praktisch:

**Listing 5.3** src/main/java/com/tutego/insel/string/LengthAndEmptyDemo.java, Ausschnitt

```
/**
 * Checks if a String is {@code null} or empty ({@code ""}).
 *
 * <pre>
 * StringUtils.isNullOrEmpty(null) == true
 * StringUtils.isNullOrEmpty("") == true
 * StringUtils.isNullOrEmpty(" ") == false
 * StringUtils.isNullOrEmpty("bob") == false
 * StringUtils.isNullOrEmpty(" 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.isEmpty();
}
```

Ob der String nur aus Weißraum besteht, testet die Methode nicht.

#### 5.4.4 Zugriff auf ein bestimmtes Zeichen mit `charAt(int)`

Die vielleicht wichtigste Methode der Klasse `String` ist `charAt(int index)`.<sup>5</sup> 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 bzw. gleich der Anzahl der Zeichen im String, so löst die Methode eine `StringIndexOutOfBoundsException`<sup>6</sup> mit der Fehlerstelle aus.

---

<sup>5</sup> Der Parametertyp `int` verrät, dass Strings nicht größer als `Integer.MAX_VALUE` sein können, also nicht länger als 2.147.483.647 Zeichen.

<sup>6</sup> Mit 31 Zeichen gehört dieser Klassenname schon zu den längsten. Übertroffen wird er aber noch um fünf Zeichen von `TransformerFactoryConfigurationError`. Im Spring-Paket (einer Sammlung von Bibliotheken für Java EE-Entwicklung) gibt es aber `JdbcUpdateAffectedIncorrectNumberOfRowsException` – auch nicht von schlechten Eltern.

**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 );                  // .'
```

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(int) stattfindet, ist zu überlegen, ob der String bei mehrmaligem Zugriff nicht stattdessen einmalig in ein eigenes Zeichen-Array kopiert werden sollte.

**Hinweis \***

[&lt;]

Da Unicode-Zeichen auch aus 2 Java-chars zusammengesetzt sein können (Surrogate), gibt es eine alternative Methode int codePointAt(int index). Die Methode ist gegenüber charAt(...) langsamer, da eine lineare Suche erfolgen muss, denn es ist im Vorfeld nicht bekannt, an welcher Stelle ein Unicode-Zeichen aus genau einem char oder aus zwei chars steht. Im Buch gehen wir immer von Unicode-Zeichen aus, die in einem char kodiert werden können.

### 5.4.5 Nach enthaltenen Zeichen und Zeichenfolgen suchen

Die Objektmethode contains(CharSequence) testet, ob ein *Teil-String* (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 5.4** src/main/java/com/tutego/insel/string/EmailSpamChecker.java, Ausschnitt

```
public class EmailSpamChecker {

    public static void main( String[] args ) {
        String email1 = "Hallo Mohammed,...";
        System.out.println( containsSpam( email1 ) ); // false
        String email2 = "Kaufe Viagra! Noch billiga und macht noch härta!";
        System.out.println( containsSpam( email2 ) ); // true
    }

    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 bzw. Teil-Strings. Findet `indexOf(...)` nichts, liefert sie `-1`.



#### 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 durchnummert. Ist das Index-Argument kleiner 0, so wird dies ignoriert und der Index automatisch auf 0 gesetzt.



#### 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`<sup>7</sup>, um nach ganzen Zeichenfolgen zu suchen und die Startposition zurückzugeben.

---

<sup>7</sup> 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 mehr in [Abschnitt 5.6, »CharSequence als Basistyp«](#)).



### Beispiel

Aufruf von `indexOf(String, int)` mit der Suche nach einem Teil-String:

```
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 errechnet sich ausgehend von der alten Finder-Position. Das Ergebnis ist 57, da dort zum zweiten Mal das Wort »es« auftaucht.



### Vom Ende an suchen

Genauso wie am Anfang gesucht werden kann, ist es auch möglich, am Ende zu beginnen.

### Beispiel

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 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 Teil-Strings einer Zeichenkette \*

Bisher bietet die Java-Bibliothek keine direkte Methode, die Anzahl der Teil-Strings einer Zeichenkette herauszufinden. Eine solche Methode ist jedoch schnell geschrieben:

**Listing 5.5** src/main/java/com/tutego/insel/string/CountMatches.java, Ausschnitt

```
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
}
}

```

Die gewählte Implementierung liefert die Anzahl nicht überlappender Teil-Strings, würde also beim Aufruf `frequency("aaaa", "aa")` das Ergebnis 2 liefern, denn mit `pos += part.length()` geht die Implementierung immer so viele Schritte weiter nach rechts, wie der Teil-String lang ist. Je nach Interpretation der Aufgabe – und anderem Algorithmus – wäre auch 3 eine erlaubte Rückgabe, aber dann müsste die Schleife immer nur eine Position nach rechts gehen.

#### 5.4.6 Das Hangman-Spiel

Die Methoden, die wir bisher schon kennengelernt haben, lösen geschätzte 90 Prozent aller Praxisaufgaben:

- ▶ `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 5.6** src/main/java/com/tutego/insel/string/Hangman1.java, Ausschnitt

```

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 = 0; ; guesses++ ) {
        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 guessedChar = new java.util.Scanner( System.in ).next().charAt( 0 );

        if ( usedChars.indexOf( guessedChar ) >= 0 )
            System.out.printf( "%c hast du schon mal getippt!%n", guessedChar );
        else { // Zeichen wurde noch nicht benutzt
            usedChars += guessedChar;
            if ( hangmanWord.indexOf( guessedChar ) >= 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", guessedChar );
            }
            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",
                guessedChar );
        }
    }
}

```

```

    }
}
}
```

Das Spiel startet mit einer Ausgabe wie:

Runde 0. Bisher geraten: \_\_\_\_\_. Was wählst du für ein Zeichen?

Wir geben einen Buchstaben ein, etwa »e«, drücken dann , und das Programm reagiert:

e

Gut geraten, 'e' gibt es im Wort. Aber es fehlt noch was!

Runde 1. Bisher geraten: \_\_\_\_\_ e\_. Was wählst du für ein Zeichen?

a

Gut geraten, 'a' gibt es im Wort. Aber es fehlt noch was!

Runde 2. Bisher geraten: a\_\_\_\_a\_\_e\_. Was wählst du für ein Zeichen?

Dann geht es weiter bis zum Ende des Spiels.

### 5.4.7 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.
- ▶ Die `switch`-Anweisung ermöglicht den Vergleich von `String`-Objekten mit einer Liste von Sprungzielen. Intern führt `equals(...)` den Vergleich durch.
- ▶ 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 Skriptsprachen 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

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)
```

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 vorliegende "Passwort". Nur der equals(...)-Vergleich (2.1) und (2.2) ist hier korrekt, da hier die puren Zeichen verglichen werden, und die sind 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, Zeichenketten ohne Beachtung der Groß-/Kleinschreibung zu vergleichen; der Test findet Zeichen für Zeichen statt.



#### Beispiel

```
String str = "REISEPASS";
boolean result1 = str.equals( "Reisepass" );           // false
boolean result2 = str.equalsIgnoreCase( "ReISePaSS" ); // true
```

### Methodenvergleich

Der Vergleich `"naß".toUpperCase().equals("NASS".toUpperCase())` bzw. `"NASS".toUpperCase().equals("naß".toUpperCase())` ergibt in beiden Fällen true. Doch `"naß".equalsIgnoreCase("NASS")` bzw. `"NASS".equalsIgnoreCase("naß")` ergeben false. Da `Character.toUpperCase('ß')` bisher noch »ß« ist, kann »naß« nicht »NASS« sein.

### Lexikografische Vergleiche mit Größer-kleiner-Relation

Wie `equals(...)` und `equalsIgnoreCase(String)` vergleichen auch die Methoden `compareTo(String)` und `compareToIgnoreCase(String)` den aktuellen String mit einem anderen String. Nur ist der Rückgabewert von `compareTo(String)` kein boolean, sondern ein int. Das Ergebnis signalisiert, ob das Argument lexikografisch kleiner oder größer als das String-Objekt ist oder mit diesem übereinstimmt. Das ist zum Beispiel in einer Sortiermethode wichtig. Der Sortieralgorithmus muss beim Vergleich zweier Strings wissen, wie sie einzusortieren sind.



### Beispiel

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(String)` größer 0.

Der von `compareTo(String)` 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 müssen wir Collator-Klassen nutzen, die im zweiten Band vorgestellt werden.

`compareToIgnoreCase(...)` ist mit `equalsIgnoreCase(...)` vergleichbar, bei der die Groß-/Kleinschreibung 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 10.4, »Vergleichen von Objekten und Ordnung herstellen«](#), 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 endet.

#### Beispiel

Teste mit `endsWith(String)` eine Dateinamenendung und mit `startsWith(String)` eine Anrede:

```
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
```

[zB]

### 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";
// Position: 0      9
boolean b = s.regionMatches( 9, "Bewegungsarmut bei Kindern", 19, 6 );
// Position:          0            19
```

[zB]

Die Methode beginnt den Vergleich am neunten Zeichen, also bei »K« im String `s`, und beim 19. Buchstaben in dem Vergleichs-String, 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« `true`.

#### Beispiel

Sollte der Vergleich unabhängig von der Groß-/Kleinschreibung stattfinden, ist das erste Argument der überladenen Methode `true`:

[zB]

```
String s = "Deutsche KINDER sind zu dick";
boolean b = s.regionMatches( true, 9, "Bewegungsarmut bei kindern", 19, 6 );
```

### 5.4.8 String-Teile extrahieren

Die wichtigste Methode, `charAt(int index)`, der Klasse `String` haben wir schon mehrfach benutzt. Sie ist aber nicht die einzige Methode, auf gewisse Teile eines Strings zuzugreifen.

#### Teile eines Strings als String mit `substring(...)` erfragen

Wollen wir einen Teil-String 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.



#### 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: 0           19
String s2 = s1.substring( 19 );           // Penetration
```

Der Index von `substring(int)` gibt die Startposition (nullbasiert) 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.



#### 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. Diese Angabe ist in Java üblich. Das heißt, bei der Endposition gehört das Zeichen nicht mehr zur Teilzeichenkette.

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.

### Beispiel



Konvertiere das erste Zeichen eines Textes in Großbuchstaben:

```
public static String firstToUpper( String string ){
    if ( string.isEmpty() ) return "";
    return string.substring( 0, 1 ).toUpperCase().concat( string.substring( 1 ) );
}
```

Als Notiz sei angemerkt, dass die erste Variante einer anderen Lösung gegenübersteht:

```
return Character.toUpperCase( string.charAt( 0 ) ) + string.substring( 1 );
```

Doch die erste Version hat den Vorteil, dass die Konvertierung auch berücksichtigt, wenn ein String etwa mit »ß« beginnt, denn das wird bisher zu »SS«. Zwar gibt es diesen Fall im Deutschen nicht, aber es gibt noch weitere Fälle in anderen Sprachen, bei denen aus einem Zeichen nach der Konvertierung in Großbuchstaben plötzlich zwei Zeichen werden; siehe dazu auch [Abschnitt 5.4.9, »Strings anhängen, zusammenfügen, Groß-/Kleinschreibung und Weißraum«](#).

Selbstverständlich kommen nun diverse Indexüberprüfungen hinzu – eine `StringIndexOutOfBoundsException`<sup>8</sup> meldet fehlerhafte Positionsangaben wie bei `charAt(int)`.

### String vor/nach einem Trenn-String \*

Die String-Klasse bietet keine einfache Bibliotheksmethode für den Fall an, dass ein Trennzeichen gegeben und ein Teil-String vor oder nach diesem Trennzeichen gefragt ist.<sup>9</sup> 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", "." )` liefert "index".
- ▶ `substringAfter( "index.html", "." )` liefert "html".

<sup>8</sup> Die API-Dokumentation spricht von einer `IndexOutOfBoundsException`, doch `StringIndexOutOfBoundsException` ist eine Unterklasse davon.

<sup>9</sup> 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`. Mit regulären Ausdrücken oder mit `split(..)` von `String` lässt sich so etwas theoretisch lösen, aber elegant ist der Code auch nicht.

Die Implementierung der Methoden ist einfach: Im ersten Schritt suchen die Methoden mit `indexOf(...)` nach dem Trenner. Anschließend liefern sie mit `substring(...)` den Teil-String vor bzw. hinter diesem gefundenen Trenn-String. 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 Leer-String. 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 5.7** src/main/java/com/tutego/insel/string/StringUtils.java, Ausschnitt

```
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 first 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(int)` nur ein Zeichen vom String liefert, kopiert `getChars(int srcBegin, int srcEnd, char[] dest, int dstBegin)` mehrere Zeichen aus einem angegebenen Bereich des Strings in ein übergebenes Array.

#### Beispiel



Kopiere Teile des Strings in ein Array:

```
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 Array, 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. `chars()` ist eine Methode, die einen Strom von Zeichen für die Zeichenkette liefert; siehe dazu auch [Abschnitt 5.6, »CharSequence als Basistyp«](#).

### Weitere Möglichkeiten zur Zerlegung

Es gibt zwei weitere Möglichkeiten, aus einem String gewisse Teile zu extrahieren. Zum einen erlaubt `split(...)` das Zerlegen einer Zeichenkette mithilfe eines regulären Ausdrückes. Dann gibt es noch seit Java 11 die Methode `lines()`, die einen Strom von Zeilen liefert, getrennt durch den Zeilenumbruch.



#### Beispiel

Ein String besteht aus mehreren Zeilen. Schneide in jeder Zeile vorne und hinten den Weißraum ab und gib alle Zeilen aus:

```
String text = " Masterpiece Tweeter\t \n Dolomiten";
text.lines().forEach( line -> System.out.println( line.trim() ) );
```

Das Beispiel greift auf Lambda-Ausdrücke zurück, die Teil von [Kapitel 12, »Lambda-Ausdrücke und funktionale Programmierung«](#), sind.

### 5.4.9 Strings anhängen, zusammenfügen, Groß-/Kleinschreibung und Weißraum

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

Um an einen String einen anderen String anzuhängen, gibt es zwei naheliegende Möglichkeiten:

- ▶ mit der `String`-Methode `concat(String)`
- ▶ mit dem Plus-Operator. Das ist im Grunde flexibler, da Nicht-Strings vorher in Strings konvertiert werden.

Wir werden später sehen, dass es die `StringBuilder`-/`StringBuffer`-Klassen noch weiter treiben und überladene `append(...)`-Methoden für unterschiedliche Datentypen anbieten.



### Beispiel

Hänge die aktuelle Zeit hinter eine Zeichenkette:

```
String s1 = "Die Uhrzeit ist: ";
String s2 = LocalTime.now().toString();
String s3 = s1.concat( s2 ); // Die Uhrzeit ist: 20:08:39.267490200
```

Ähnlich wie im obigen Beispiel können wir Folgendes schreiben:

```
String s4 = "Die Uhrzeit ist: " + LocalTime.now().toString();
```

Es geht sogar noch kürzer, denn der Plus-Operator ruft automatisch `toString()` bei Objekten auf:

```
String s5 = "Die Uhrzeit ist: " + LocalTime.now();
```

`concat(String)` legt ein internes Array an, kopiert die beiden Zeichenreihen per `getChars(...)` hinein und liefert mit einem `String`-Konstruktor die resultierende Zeichenkette.

### Mehrere Strings zusammenfügen

Zwei überladene Klassenmethoden `join(...)` können mehrere Zeichenketten zusammenfügen und zwischen den Gliedern einen Trenn-String setzen. Die Methoden unterscheiden sich in der Parameterliste: Einmal kommen die Glieder direkt über ein Vararg und einmal über einen Iterable.

- `static String join(CharSequence delimiter, CharSequence... elements)`
- `static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)`

Die Argumente müssen Zeichenfolgen sein und können nicht etwa allgemeine Objekte sein, die automatisch in Strings konvertiert werden. Auch `null` dürfen die Glieder nicht sein.



### Beispiel

Füge die Wörter mit einem Leerzeichen zu einem Satz zusammen:

```
String s = String.join( " ", "Geht", "ein", "Mullah", "in", "die", "Bar." );
```

Der Trenn-String darf nicht `null` sein, kann aber durchaus leer sein.

In `StringBuilder/StringBuffer` gibt es keine vergleichbare Methode, wohl aber eine kleine Klasse `StringJoiner`, die auch die eigentliche Arbeit von `join(...)` vollzieht.

### Groß-/Kleinschreibung

Die Klasse `Character` deklariert einige statische Methoden, die einzelne Zeichen in Groß-/Kleinbuchstaben umwandeln. 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 »SPASZ 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. Und seit dem 29. Juni 2017 ist das große Eszett Bestandteil der amtlichen deutschen Rechtschreibung; es liegt im Unicode-Alphabet an Position U+1E9E. Warten wir ab, ob der Konverteralgorithmus umgestellt wird.



### Beispiel

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 passt die neue Zeichenkette möglicherweise nicht mehr in den Speicherbereich. 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 `toXXXXXCase(...)`-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 Ergebnis-String hat das i keinen i-Punkt!



### Hinweis

Die parameterlosen Methoden `toUpperCase()` und `toLowerCase()` wählen die Sprachumgebung gemäß den Ländereinstellungen des Betriebssystems aus. Am Beispiel `toLowerCase()`:

```
public String toLowerCase() {
    return toLowerCase( Locale.getDefault() );
}
```

Die Voreinstellung muss nicht die beste sein.

### Weißraum entfernen

In einer Benutzereingabe oder Konfigurationsdatei steht nicht selten vor oder hinter dem wichtigen Teil eines Textes Weißraum wie Leerzeichen oder Tabulatoren. Vor der Bearbeitung sollten sie entfernt werden. Die String-Klasse bietet dazu `trim()` und seit Java 11 `strip()`, `stripLeading()` und `stripTrailing()` an. Der Unterschied:

Methode	Entfernt ...
<code>trim()</code>	... am Anfang und am Ende des Strings alle Codepoints kleiner oder gleich dem Leerzeichen 'U+0020'.
<code>strip()</code>	... alle Zeichen am Anfang und am Ende des Strings, die nach der Definition von <code>Character.isWhitespace(int)</code> Leerzeichen sind.
<code>stripLeading()</code>	wie <code>strip()</code> , allerdings nur am Anfang des Strings
<code>stripTrailing()</code>	wie <code>strip()</code> , allerdings nur am Ende des Strings

**Tabelle 5.8** Unterschiede zwischen `trim()` und `stripXXX()`

Alle vier Methoden entfernen keinen Weißraum inmitten des Strings.



### 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.'
```



### Beispiel

Teste, ob ein String mit Abzug allen Weißraums leer ist:

```
boolean isBlank = "".equals( s.trim() );
```

Alternativ:

```
boolean isBlank = s.trim().isEmpty();
```

### 5.4.10 Gesucht, gefunden, ersetzt

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 alle Auftreten des Zeichens `oldChar` durch `newChar`.
- `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, den ein regulärer Ausdruck beschreibt.

#### Ersetzen ohne reguläre Ausdrücke

Die `replace(char, char)`-Methode ersetzt einzelne Zeichen.



#### 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 char-Array, führt die Ersetzungen durch und konvertiert das interne Zeichen-Array in ein String-Objekt, das 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 Ersetzungs-String kann auch leer sein, und so werden die Zeichen im Ergebnis gelöscht.

### Beispiel

[zB]

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 und nehmen Ersetzungen vor; `replaceFirst(...)` ersetzt, wie der Name schon sagt, nur das erste Auftreten.

### Beispiel

[zB]

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?
```

Weil der Such-String 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 und auch schneller.

### Beispiel

[zB]

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 ein Backslash (\) den Punkt ausmaskiert – wegen des Sonderstatus des Backslashes als Escape-Sequenz in Strings 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. Im 2. Band werden wir uns intensiver mit regulären Ausdrücken beschäftigen. Auch die API-Dokumentation <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html> gibt neben einer vollständigen Dokumentation kleine Beispiele.

#### 5.4.11 String-Objekte mit Konstruktoren und aus Wiederholungen erzeugen \*

Die String-Klasse hat diverse Methoden, die neue String-Objekte aufbauen, insbesondere weil String-Objekte immutable sind. Zu den Methoden zählen `substring(...)`, `join(...)`, `format(...)`.

##### Neue String-Objekte mit Konstruktoren aufbauen

Liegt die Zeichenkette nicht als String-Literal vor, lassen sich mit den unterschiedlichen Konstruktoren der String-Klasse neue String-Objekte aufbauen. Die meisten Konstruktoren 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`. Der Konstruktor ist im Prinzip unnötig, da String-Objekte unveränderbar (*immutable*) sind.
- `String(char[] value)`  
Erzeugt ein neues Objekt und kopiert die im `value` vorhandenen Zeichen in das neue String-Objekt.
- `String(char[] value, int offset, int length)`  
Erzeugt wie `String(char[])` einen String aus einem Ausschnitt eines char-Arrays. Der verwendete Ausschnitt beginnt bei dem Index `offset` und umfasst `length` Zeichen.

- `String(byte[] bytes)`  
Erzeugt ein neues Objekt aus dem Array `bytes`. 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-Arrays.
- `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.

[zB]

### Beispiel

Erzeuge einen String einer gegebenen Länge:

```
public static String generateStringWithLength( int len, char fill ) {
    char[] chars = new char[ len ];
    Arrays.fill( chars, fill );
    return new String( chars );
}
```

In der `String`-Klasse gibt es keine Methode, die eine Zeichenkette einer vorgegebenen Länge aus einem einzelnen Zeichen erzeugt.

[zB]

### Beispiel

Teste, ob zwei Zeichenketten – unabhängig von der Groß-/Kleinschreibung – 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(...)` sortiert ein Array, in dem Fall das char-Array. Ist die Groß-/Kleinschreibung relevant, kann gleich `Arrays.equals(char[], char[])` herangezogen werden.

### Über den Konstruktoraufruf `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. Der Konstruktor ist im Grunde unnötig.

### 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 die JVM sie im Programmverlauf benutzt und welche Klassen den String referenzieren. Dieses String-Objekt »lebt« in einem Bereich, der *Konstantenpool* genannt wird.<sup>10</sup>



#### 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.

---

<sup>10</sup> Die Java-Bibliothek implementiert hier das Entwurfsmuster *Fliegengewicht* (Flyweight-Pattern) der Gang of Four.

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

[zB]

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.;"
```

Der Compiler fügt die Zeichenketten<sup>11</sup> automatisch zu einer großen Zeichenkette zusammen, sodass keine Konkatenation zur Laufzeit nötig ist. Konstante Ausdrücke »rechnet« der Compiler immer direkt selbst aus.

### 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 einen 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
```

<sup>11</sup> Das Zitat stammt übrigens von Bill Joy (eigentlich heißt er William Nelson Joy), der Sun Microsystems mitgründete. 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 und dem vi-Editor.

`printXXX(null)` führt zu der Konsolenausgabe »null« und zu keiner Ausnahme, da es eine Fallunterscheidung in `printXXX(Object)` und `printXXX(String)` gibt, die die `null`-Referenz als Sonderfall betrachtet.<sup>12</sup> Der Zugriff auf `s` über `s.length()` führt dagegen zur unbeliebten `NullPointerException`.

### Strings aus Wiederholungen generieren

In Java 11 ist eine Objektmethode `repeat(int count)` eingezogen, die einen gegebenen String vervielfacht.



#### Beispiel

Wiederhole den String `s` dreimal:

```
String s = "tu";
System.out.println( s.repeat( 3 ) );    // tututu
```

Bevor es die Methode in Java 11 gab, sah eine alternative Lösung etwa so aus:

```
int    n = 3;
String t = new String( new char[ n ] ).replace( "\0", s );
System.out.println( t );                // tututu
```

## 5.5 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 Klassen `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(String)`-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 Zeichen-Array.

Die Klasse `StringBuilder` bietet die gleichen Methoden wie `StringBuffer`, nur nicht synchronisiert. Bei nebenläufigen Programmen kann daher die interne Datenstruktur des `StringBuilder`-Objekts inkonsistent werden, sie ist aber dafür bei nichtnebenläufigen Zugriffen ein wenig schneller.

---

<sup>12</sup> In der Implementierung von `PrintStream`: `public void print( String s ) { if ( s == null ) s = "null"; write( s ); }`

## 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.

### Hinweis

Wegen der symmetrischen API von `StringBuffer` und `StringBuilder` werden wir im Folgenden immer nur von den Methoden von `StringBuilder` sprechen.



### 5.5.1 Anlegen von `StringBuilder`-Objekten

Mit mehreren Konstruktoren lassen sich `StringBuilder`-Objekte aufbauen:

```
final class java.lang.StringBuilder
    implements Appendable, CharSequence, Comparable<StringBuilder>, Serializable
```

- `StringBuilder()`

Legt ein neues Objekt an, das die leere Zeichenreihe enthält und Platz für (zunächst) bis zu 16 Zeichen bietet. Spätere Einfügeoperationen füllen den Puffer und vergrößern ihn automatisch weiter.

- `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.

- `StringBuilder(String str)`

Baut ein Objekt, das eine Kopie der Zeichen aus `str` enthält. Zusätzlich plant der Konstruktor bereits Platz für 16 weitere Zeichen ein.

- `StringBuilder(CharSequence seq)`

Erzeugt ein neues Objekt aus einer `CharSequence`. Damit können auch die Zeichenfolgen anderer `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`-Exemplare anzulegen. Alle `String`-Literale in Anführungszeichen sind ja schon Exemplare der Klasse `String`.

### Hinweis

Weder in der Klasse `String` noch in `StringBuilder` existiert ein Konstruktor, der explizit ein `char` als Parameter zulässt, um aus dem angegebenen Zeichen eine Zeichenkette aufzubauen. Dennoch gibt es bei `StringBuilder` einen Konstruktor, der ein `int` annimmt, wobei die über-



gebene 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 den Aufbau einer veränderbaren Zeichenkette, gefüllt mit dem Startzeichen `c`, nur Folgendes: `new StringBuilder(Character.toString(c))` oder `new StringBuilder().append(c)`.

### 5.5.2 **StringBuilder in andere Zeichenkettenformate konvertieren**

`StringBuilder` 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.StringBuilder
implements Appendable, CharSequence, Comparable<StringBuilder>, 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`-Array.

### 5.5.3 **Zeichen(folgen) erfragen**

Die bekannten Anfragemethoden aus `String` finden wir auch beim `StringBuilder` wieder. So verhalten sich `charAt(int)` 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` unnötig, und ein `String`-Objekt selbst reicht.

### 5.5.4 **Daten anhängen**

Die häufigste Anwendung von `StringBuilder`-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`. Die `append(...)`-Methoden hängen sich immer an das Ende an und vergrößern den internen Platz – ein internes `char`-Array –, falls es nötig ist. Ein neues `StringBuilder`-Objekt erzeugen sie nicht.

Nutzen wir dies für eine eigene Methode `enumerate(...)`, die ein `String`-Array abläuft und eine Bildschirmaufzählung erzeugt:

**Listing 5.8** src/main/java/com/tutego/insel/string/Enumerator.java, Ausschnitt

```

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] ).append( System.lineSeparator() );
        }

        return sb.toString().trim();
    }

    public static void main( String[] args ) {
        System.out.println( enumerate( "Aufstehen", "Frühstück" ) );
    }
}

```

Die Ausgabe ist:

1. Aufstehen
2. Frühstück

Die Zusammenfassung listet alle append(...)-Methoden auf.

```

final class java.lang.StringBuilder
implements Appendable, CharSequence, Comparable<StringBuilder>, Serializable

```

- `StringBuilder append(boolean b)`
- `StringBuilder append(char c)`
- `StringBuilder append(char[] str)`
- `StringBuilder append(char[] str, int offset, int len)`
- `StringBuilder append(CharSequence s)`
- `StringBuilder append(CharSequence s, int start, int end)`
- `StringBuilder append(double d)`
- `StringBuilder append(float f)`

- `StringBuilder append(int i)`
- `StringBuilder append(long lng)`
- `StringBuilder append(Object obj)`
- `StringBuilder append(String str)`
- `StringBuilder append(StringBuilder 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` 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«.



### Tipp

Den Plus-Operator innerhalb von `append(...)`-Strings zu nutzen ist kontraproduktiv. Heißt es etwa `sb.append(", " + value)`, sollte es besser `sb.append(", ").append(value)` heißen.

## 5.5.5 Zeichen(folgen) setzen, löschen und umdrehen

Da sich bei einem `StringBuilder` Zeichen verändern lassen, gibt es neben den `append(...)`-Methoden weitere Modifikationsmethoden, die in der Klasse `String` fehlen.

### Einzelne Zeichen setzen

```
final class java.lang.StringBuilder  
implements Appendable, CharSequence, Comparable<StringBuilder>, Serializable
```

- `void setCharAt(int index, char ch)`

Setzt an die Stelle `index` das Zeichen `ch` und überschreibt das alte Zeichen.



### 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(int, char)` steht für die Position des zu setzenden Zeichens.

### Zeichenfolgen einfügen

Diverse `insert(...)`-Methoden fügen die Zeichenketten-Repräsentation eines Werts an eine bestimmte Stelle ein. Sie ähnelt der überladenen `append(...)`-Methode.

```
final class java.lang.StringBuilder
implements Appendable, CharSequence, Comparable<StringBuilder>, Serializable
```

- `StringBuilder insert(int offset, boolean b)`
- `StringBuilder insert(int offset, char c)`
- `StringBuilder insert(int offset, char[] str)`
- `StringBuilder insert(int index, char[] str, int offset, int len)`
- `StringBuilder insert(int dstOffset, CharSequence s)`
- `StringBuilder insert(int dstOffset, CharSequence s, int start, int end)`
- `StringBuilder insert(int offset, double d)`
- `StringBuilder insert(int offset, float f)`
- `StringBuilder insert(int offset, int i)`
- `StringBuilder insert(int offset, long l)`
- `StringBuilder insert(int offset, Object obj)`
- `StringBuilder insert(int offset, String str)`



### Beispiel \*

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 5.9** src/main/java/com/tutego/insel/string/ReverseFile.java, Ausschnitt

```
InputStream resource = ReverseFile.class.getResourceAsStream( "EastOfJava.txt" );
try ( Scanner input = new Scanner( resource ) ) {
```

```
StringBuilder result = new StringBuilder();
while ( input.hasNextLine() )
    result.insert( 0, input.nextLine() + System.lineSeparator() );
System.out.println( result.toString().trim() );
}
```

Die Konstruktion mit dem try ist neu und wird in [Abschnitt 8.6.1](#), »try mit Ressourcen«, näher erklärt.

Für char-Arrays existiert insert(...) in einer abgewandelten Art: insert(int index, char[] str, int offset, int len). Die Methode übernimmt nicht das komplette Array in den StringBuilder, sondern nur einen 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.



#### Beispiel

Ersetze den Teil-String an den Positionen 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.



#### 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 );
```

### 5.5.6 Länge und Kapazität eines StringBuilder-Objekts \*

Wie bei einem String lässt sich die Länge und die Anzahl der enthaltenen Zeichen mit der Methode `length()` erfragen. `StringBuilder`-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 Array benötigt würde.

#### Beispiel

[zB]

```
StringBuilder sb = new StringBuilder( "www.tutego.de" );
System.out.println( sb.length() );                                // 13
System.out.println( sb.capacity() );                            // 29
```

Bei der Längenabfrage liefert `sb.length()` 13, aber `sb.capacity()` ergibt  $13 + 16 = 29$ .

Die Startgröße sollte mit der erwarteten Größe initialisiert werden, um ein späteres teures internes Vergrößern zu vermeiden. Falls der `StringBuilder` 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` mehr Daten aufnehmen, so ändert `setLength(int)` die Länge in 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.

#### Beispiel

[zB]

Hänge alle Strings eines Arrays mit einem Trenner zusammen und schneide den letzten Trenner zum Schluss ab:

```
String[] elements = { "Manila", "Cebu", "Ipil" };
String separator = ", ";
StringBuilder sb = new StringBuilder();
for ( String elem : elements )
    sb.append( elem ).append( separator );
sb.setLength( sb.length() - separator.length() );
System.out.println( sb );           // Manila, Cebu, Ipil
```

Ist `setLength(int)` größer, so vergrößert sich der Puffer, und die Methode füllt die übrigen Zeichen mit Nullzeichen '\0000' auf. Die Methode `ensureCapacity(int)` 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`-Objekt repräsentiert wird.

### 5.5.7 Vergleich von `StringBuilder`-Exemplaren und `String` mit `StringBuilder`

Zum Vergleichen von `Strings` bietet sich die bekannte `equals(...)`-Methode an. Diese ist aber bei `StringBuilder` nicht wie erwartet implementiert. Dazu gesellen sich andere Methoden, die zum Beispiel unabhängig von der Groß-/Kleinschreibung vergleichen.

#### `equals(...)` bei der `String`-Klasse

Die Klasse `String` implementiert die `equals(Object)`-Methode, sodass ein `String` mit einem anderen `String` verglichen werden kann. Allerdings vergleicht `equals(Object)` von `String` nur `String/String`-Paare. Die Methode beginnt erst dann den Vergleich, wenn das Argument auch vom Typ `String` ist. Das bedeutet, dass der Compiler alle Übergaben auch vom Typ `StringBuilder` bei `equals(Object)` zulässt, doch zur Laufzeit ist das Ergebnis immer `false`, da eben ein `StringBuilder` nicht vom Typ `String` ist. Ob die Zeichenfolgen dabei gleich sind, spielt keine Rolle.

#### `contentEquals(...)` beim `String`

Eine allgemeine Methode zum Vergleichen eines `Strings` mit entweder einem anderen `String` oder mit `StringBuilder` 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.



#### Beispiel

Vergleiche einen `String` mit einem `StringBuilder`:

```
String      s = "Elektrisch-Zahnbürster";
StringBuilder sb = new StringBuilder( "Elektrisch-Zahnbürster" );
System.out.println( s.equals(sb) );                      // false
System.out.println( s.equals(sb.toString()) );          // true
System.out.println( s.contentEquals(sb) );               // true
```

## Kein eigenes equals(...) bei StringBuilder

Wollen wir zwei `StringBuilder`-Objekte miteinander vergleichen, so geht das *nicht* mit der `equals(...)`-Methode. Es gibt zwar die übliche von `Object` geerbte Methode, doch das heißt, nur Objektreferenzen werden verglichen. Anders gesagt: `StringBuilder` überschreibt die `equals(...)`-Methode nicht. Wenn also zwei verschiedene `StringBuilder`-Objekte mit gleichem Inhalt mit `equals(...)` verglichen werden, kommt trotzdem immer `false` heraus.



### Beispiel

Um den inhaltlichen Vergleich von zwei `StringBuilder`-Objekten zu realisieren, können wir sie erst mit `toString()` in `String`s umwandeln und dann mit `String`-Methoden vergleichen:

```
StringBuilder sb1 = new StringBuilder( "The Ocean Cleanup" );
StringBuilder sb2 = new StringBuilder( "The Ocean Cleanup" );
System.out.println( sb1.equals( sb2 ) ); // false
System.out.println( sb1.toString().equals( sb2.toString() ) ); // true
System.out.println( sb1.toString().contentEquals( sb2 ) ); // true
```

Ab Java 11 gibt es bessere Lösungen ...

## Lexikografische Vergleiche (`StringBuilder` ist Comparable)

Seit Java 11 bietet `StringBuilder` eine Methode `int compareTo(StringBuilder another)`, sodass lexikografische Vergleiche möglich sind. (`StringBuilder` implementiert die Schnittstelle `Comparable<StringBuilder>`.) Somit realisieren `String` und `StringBuilder` beide eine Ordnung, siehe den Abschnitt »Lexikografische Vergleiche mit Größer-kleiner-Relation« in [Abschnitt 5.4.7](#).

Eine Begleiterscheinung ist die Tatsache, dass bei gleichen Zeichenfolgen die Rückgabe von `compareTo(...)` gleich 0 ist. Das ist deutlich besser, als erst den `StringBuilder` in einen `String` zu konvertieren.

Die `compareTo(...)`-Methode `StringBuilder` vergleicht nur mit anderen `StringBuilder`-Objekten. Flexibler ist da die in Java 11 hinzugekommene statische Methode `compare(CharSequence, CharSequence)` in `CharSequence`; hiermit ist ein lexikografischer Vergleich aller erdenklichen `CharSequence`-Exemplare möglich.



### Beispiel

```
StringBuilder sb1 = new StringBuilder( "The Ocean Cleanup" );
StringBuilder sb2 = new StringBuilder( "The Ocean Cleanup" );
System.out.println( sb1.compareTo( sb2 ) == 0 ); // true
System.out.println( CharSequence.compare( sb1, sb2 ) == 0 ); // true
```

### 5.5.8 hashCode() bei StringBuilder \*

Die obige Betrachtung zeigt, dass eine Methode `equals(...)`, die den Inhalt von `StringBuilder`-Objekten vergleicht, nicht schlecht wäre. Dennoch besteht das Problem, wann `StringBuilder`-Objekte als gleich angesehen werden sollen. Das ist interessant, denn `StringBuilder`-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` erbt die Implementierung aus der Klasse `Object` unverändert. Mit anderen Worten: Die Klassen selbst bieten keine Implementierung an.

## 5.6 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`. Dieser Typ steht für eine unveränderliche, nur lesbare Sequenz von Zeichen (Schnittstellen und Basistypen sowie Implementierungen werden präziser in [Abschnitt 7.7 »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; als Rückgabe ist `CharSequence` eher unpraktisch, hier ist `String` praktischer. 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.

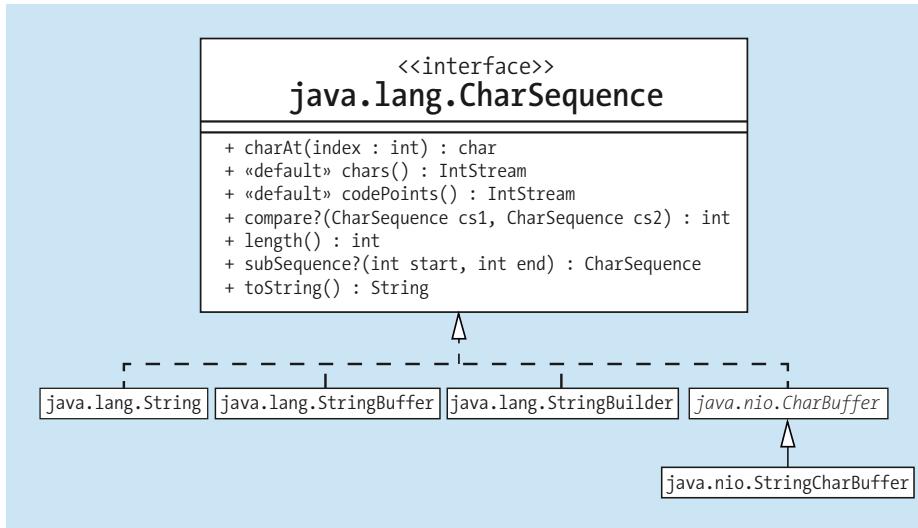


Abbildung 5.3 Einige implementierende Klassen der Schnittstelle CharSequence

### Beispiel

[zB]

Soll eine Methode eine Zeichenkette bekommen und ist der Typ egal, so implementieren wir etwa

```
void process( CharSequence s ) {
    ...
}
```

statt der beiden Methoden:

```
void process( String s ) { ... }
void process( StringBuilder sb ) {
    process( sb.toString() );
}
```

### Basisoperationen der Schnittstelle

Die Schnittstelle CharSequence schreibt für die implementierenden Klassen zwingend drei Methoden vor:

```
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. subSequence(...) liefert ein CharSequence-Objekt als Rückgabe – die Methode macht im Prinzip nichts anderes als ein substring(begin, end) bei einem String, nur der Rückgabetyp ist anders.

Die Methode, die sowieso schon über die absolute Oberklasse `java.lang.Object` vorhanden ist, ist:

- String `toString()`  
Gibt einen String der Sequenz zurück. Die Länge des `toString()`-Strings entspricht genau der Länge der Sequenz.

Unter anderem sind `String/StringBuilder/StringBuffer` implementierende Klassen der Schnittstelle `CharSequence` und realisieren diese Methoden.

```
class java.lang.String implements CharSequence, ...
class java.lang.StringBuilder implements CharSequence, ...
class java.lang.StringBuffer implements CharSequence, ...
```

- CharSequence `subSequence(int beginIndex, int endIndex)`  
Liefert eine nur lesbare Teilzeichenkette.

### Statische compare(...)-Methode in CharSequence

Seit Java 11 gibt es in `CharSequence` eine neue Methode `compare(...)`, die zwei `CharSequence`-Objekte lexikografisch vergleicht.

```
interface java.lang.CharSequence
```

- static int `compare(CharSequence cs1, CharSequence cs2)`  
Vergleicht die beiden Zeichenketten lexikografisch.

Die statische Methode hat den Vorteil, dass nun alle Kombinationen von `CharBuffer`, `Segment`, `String`, `StringBuffer`, `StringBuilder` mit nur dieser einen Methode geprüft werden können. Und wenn der Vergleich 0 ergibt, so wissen wir auch, dass die Zeichenfolgen die gleichen Zeichen enthalten.

## Default-Methoden in der Schnittstelle CharSequence \*

Die Schnittstelle hat zwei Default-Methoden:

```
interface java.lang.CharSequence
```

- default IntStream chars()
- default IntStream codePoints()

Die Bedeutung von `default` und Schnittstellen im Allgemeinen bleibt ein Detail aus [Kapitel 7](#), »Objektorientierte Beziehungsfragen«, an dieser Stelle wollen wir nur den Vorteil betonen, dass `chars()` gut dafür verwendet werden kann, über die Zeilen zu laufen. Allerdings hat das nichts mit dem erweiterten `for` zu tun, sondern mit einem anderen Programmieridom.

### Beispiel

Laufe über eine Zeichenkette und gib jedes Zeichen aus:

```
"Drama is life with the dull bits left out. (Hitchcock)".chars().forEach( c ->
    System.out.print( (char) c )
);
```

Das Beispiel greift hier syntaktisch schon weit voraus und nutzt so genannte Lambda-Ausdrücke, die im späteren [Kapitel 12](#), »Lambda-Ausdrücke und funktionale Programmierung«, vorgestellt werden.

[zB]

## 5.7 Konvertieren zwischen Primitiven und Strings

Bevor ein Datentyp auf dem Bildschirm ausgegeben, zum Drucker geschickt oder in einer ASCII-Datei gespeichert werden kann, muss ihn das Java-Programm 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.

### 5.7.1 Unterschiedliche Typen in String-Repräsentationen konvertieren

Die statischen überladenen `String.valueOf(...)`-Methoden liefern die String-Repräsentation eines primitiven Werts oder eines Objekts.



### Beispiel

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-Präsentation der primitiven Elemente.
- `static String valueOf(char[] data)`
- `static String valueOf(char[] data, int offset, int count)`  
Liefert vom char-Array oder einem Ausschnitt des char-Arrays ein String-Objekt.

### Die Methode `valueOf(Object)`

Der `valueOf(Object)`-Methode kann ein beliebiges Objekt übergeben werden.



### 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( java.nio.file.Paths.get( "." ) ); // .
String t = String.valueOf( java.time.LocalTime.now() ); // 09:48:28.047834
```

Da jedes Objekt eine `toString()`-Methode besitzt, delegiert `valueOf(Object)` einfach auf diese.

**Listing 5.10** java/lang/String.java, valueOf()

```
public static String valueOf( Object obj ) {
    return (obj == null) ? "null" : obj.toString();
}
```

Die Implementierung von `valueOf(Object obj)` fragt für die String-Umsetzung einfach das Objekt `obj` selbst. 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(Object)` ruft intern `String.valueOf(...)` auf. Genauso ergibt `System.out.println(null + "0")` die Ausgabe `"null0"`, 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«`.

### 5.7.2 String-Inhalt 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(String)`-Methoden, wie Tabelle 5.9 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 5.9** Methoden zum Konvertieren eines Strings in einen primitiven Typ

Für jeden primitiven Typ gibt es eine so genannte *Wrapper-Klasse* mit `parseXXX(String)`-Konvertiermethoden. Die Bedeutung der Klassen erklärt Abschnitt 10.5, »Wrapper-Klassen

und Autoboxing«, genauer. An dieser Stelle betrachten wir nur die Konvertierungsfunktionallität.

### Beispiel mit Double.parseDouble(String)

Die Methode `Double.parseDouble(String)`<sup>13</sup> wollen wir in einem Beispiel nutzen. Der Benutzer soll in einem grafischen Dialog nach einer Fließkommazahl gefragt werden, und von dieser Zahl sollen dann der Sinus und Kosinus auf dem Bildschirm ausgegeben werden:

**Listing 5.11** src/main/java/com/tutego/insel/string/SinusAndCosinus.java, Ausschnitt

```
String s = javax.swing.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(String) und mögliche NumberFormatException-Fehler

Kann eine `parseXXX(String)`-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(String)` als Dezimaltrenner ein Komma statt eines Punktes empfängt. Bei der statischen Methode `parseBoolean(String)` 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(String)` gibt es *nicht*. Eine Alternative ist, einen regulären Ausdruck zu verwenden und dagegen zu testen, etwa so:

```
stringWithNumber.matches("\\p{Digit}+")
```

### parseXXX(...) und das Verhalten mit Plus, Minus, Leerzeichen

Repräsentiert ein String negative Zahlen, so beginnt der String mit einem »-«, positive Zahlen dürfen auch mit einem »+« beginnen. Die statische Konvertierungsmethode `Integer.parseInt(String)` 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.)

---

<sup>13</sup> In der Version 6 gab es einen Fehler, der zu viel Aufregung führte: Eine Zahl konnte nicht geparsst werden, und der Compiler bzw. die Laufzeitumgebung ging in eine Endlosschleife. Mehr zu dem Fehler unter <http://www.exploringbinary.com/java-hangs-when-converting-2-2250738585072012e-308>.

**Beispiel**

[zB]

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.printXXX(...)` basiert –, kann `parseXXX(...)` wieder in den gleichen Wert zurückverwandeln.

**Hinweis**

[&lt;&lt;]

Eine Methode `Character.parseCharacter(String)` fehlt. Eine vergleichbare Realisierung ist, auf das erste Zeichen eines Strings zuzugreifen, etwa so:

```
char c = s.charAt( 0 )
```

**Lokalisiertes parseXXX(String)**

Die `parseXXX(...)`-Methoden sind nicht lokalisiert, können also nur englisch formatierte Zeichenfolgen in primitive Werte umsetzen. Bei den Ganzzahlen ist das kein Problem, nur bei Fließkommazahlen ist das ein großes Problem, denn ein `parseDouble("1,3")` führt zur `NumberFormatException`, nur `parseDouble("1.3")` nicht. Für die Lösung des Problems gibt es zwei Ansätze. Einer ist, vorher den Dezimaltrenner in einen Punkt zu konvertieren:

```
String s = "3,1";
double d = Double.parseDouble( s.replace( ',', '.' ) );           // 3.1
```

Eine andere Lösung greift auf Klassen zurück, die lokalisierter Parsen ermöglichen, etwa `Scanner` – wir kommen in [Abschnitt 5.9.2](#) noch einmal auf diese Klasse zurück:

```
String s = "3,1";
double d = new Scanner( s ).useLocale( Locale.GERMANY ).nextDouble(); // 3.1
```

**5.7.3 String-Präsentation im Format Binär, Hex, Oktal \***

Neben den überladenen statischen `String.valueOf(primitive)`-Methoden, die eine Zahl als String-Präsentation 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 eine beliebige Radix, und zum anderen gibt es Spezialmethoden für die 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) oder 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) oder 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` bzw. `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(16)</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 5.10 Beispiele für die `toHexString()`-Methode

Die Ausgaben mit `printf(...)` – bzw. die Formatierung mit `String.format(...)` – bieten eine Alternative, die in Abschnitt 5.10.1, »Formatieren und Ausgeben mit `format()`«, 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
```

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 einer beliebigen Radix ist das nicht so.



### Parsen von String mit Radix

Eine Methode zum Konvertieren eines Strings in eine Ganzzahl für eine gegebene Basis findet sich in den Klassen `Integer` und `Long`. 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)

Einige Anwendungsfälle:

Konvertierungsaufruf	Ergebnis
parseInt("0", 10)	0
parseInt("473", 10)	473
parseInt("-0", 10)	0
parseInt("-FF", 16)	-255
parseInt("1100110", 2)	102
parseInt("2147483647", 10)	2147483647
parseInt("-2147483648", 10)	-2147483648
parseInt("2147483648", 10)	💀 throws NumberFormatException
parseInt("99", 8)	💀 throws NumberFormatException
parseInt("Papa", 10)	💀 throws NumberFormatException
parseInt("Papa", 27)	500050

Tabelle 5.11 Beispiele für Integer.parseInt() mit unterschiedlichen Zahlenbasen

[zB]

### Beispiel

Die Radix geht bis 36 (zehn Ziffern und 26 Kleinbuchstaben). Mit Radix 36 können zum Beispiel ganzzahlige IDs in Textform kompakter dargestellt werden, als wenn sie dezimal wären:

```
String string = Long.toString( 2656437647773L, 36 );
System.out.println( string ); // xwcmdz8d
long l = Long.parseLong( string, 36 );
System.out.println( l ); // 2656437647773
```

### Rätsel

Welches Ergebnis ergibt der Ausdruck Long.parseLong( "" + 1 / 0., 35 )?

## Parsen von Binär-/Oktal-/Hexadezimalzahlen

Im Fall von String-Konvertierung existieren für die Standardbasen 2, 8 und 16 spezielle Methoden wie `toHexString(...)`, aber zum Parsen gibt es sie nicht. Eine Hexadezimalzahl verarbeitet `parseInt(s, 16)`, denn eine Methode wie `parseHex(String)` steht nicht bereit.

### Hinweis

Die Methoden `parseInt(...)` und `parseLong(...)` verhalten sich bei der String-Präsentation von negativen Zahlen nicht so, wie zu erwarten wäre:

```
int i = Integer.parseInt( "7fffffff", 16 ); // 2147483647
int j = Integer.parseInt( "80000000", 16 ); // ☠ NumberFormatException
```

`0x7fffffff` ist die größte darstellbare positive `int`-Zahl. Statt bei `0x80000000` den Wert –`2.147.483.648` 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
```

### 5.7.4 `parseXXX(...)`- und `printXXX()`-Methoden in `DatatypeConverter`\*

Das `javax.xml.bind`-Paket bietet eine Klasse `DatatypeConverter`, die eigentlich für die Abbildung von XML-Typen auf Java-Typen gedacht ist, doch auch so einige nützliche Methoden bereitstellt. Wir finden in der Klasse statische `parseXXX(String)`-Methoden und `printXXX(...)`-Methoden: Die ersten konvertieren einen String in diverse Datentypen – etwa `short parseShort(String)` –, und die zweiten formatieren einen bestimmten Datentyp in einen String – etwa `String printShort(short)`. Für die meisten Methoden gibt es mit `String.valueOf(...)` und den `parseXXX(...)/toString(...)`-Methoden in den Wrapper-Klassen bessere Alternativen, und die Umwandlung von Datumswerten und Fließkommazahlen sind nicht lokalisiert, doch hervorzuheben sind folgende zwei Methoden:

```
final class javax.xml.bind.DatatypeConverter
```

- `static byte[] parseHexBinary(String lexicalXSDHexBinary)`
- `static String printHexBinary(byte[] val)`

Mit diesen statischen Methoden können leicht Byte-Arrays in die String-Präsentationen hexadezimal konvertiert werden. Das ist nötig, wenn etwa Bytes in einer Text-Konfigura-

tionsdatei abgelegt werden sollen. DatatypeConverter bietet auch Methoden für eine Base64-Kodierung, allerdings sind die geschachtelten Klassen in `java.util.Base64` üblicher.



### Beispiel

Konvertiere ein Byte-Array in eine String-Präsentation, einmal im klassischen Hex-Format, einmal in Base64-Kodierung:

```
byte[] bytes = { 1, 2, 3, (byte) 254, (byte) 255 };
String s1 = DatatypeConverter.printHexBinary( bytes );
String s2 = DatatypeConverter.printBase64Binary( bytes );
System.out.println( s1 ); // 010203FEFF
System.out.println( s2 ); // AQID/v8=
// Arrays.equals( bytes, DatatypeConverter.parseHexBinary( s1 ) ) == true
// Arrays.equals( bytes, DatatypeConverter.parseBase64Binary( s2 ) ) == true
```

## 5.8 Strings zusammenhängen (konkatenieren)

Um aus Teilstings einen neuen String zu generieren, bietet Java diverse Möglichkeiten:

- ▶ den Plus-Operator bei involvierten `String`-Objekten
- ▶ die `concat(String)`-Methode der Klasse `String`
- ▶ die Klassen `StringBuilder` und `StringBuffer` mit ihren `append(...)`-Methoden
- ▶ eine Methode `join(...)` bei der Klasse `String` bzw. die Hilfsklasse `java.util.StringJoiner` zum Zusammenhängen von Teilen mit einem gemeinsamen Trennzeichen
- ▶ `format(...)` von `String` bzw. die Hilfsklasse `java.util.Formatter`

Dazu gesellen sich ein paar Methoden und Typen, die der Konkatenation zuzuordnen sind:

- ▶ zwei besondere `appendXXX(...)`-Methoden zum Anhängen von Zeichenfolgen im Zuge einer Ersetzung im Zusammenhang mit regulären Ausdrücken
- ▶ ein besonderer `StringWriter` mit `append(...)`- und `write(...)`-Methoden



### Hinweis

Intern setzt der Java Compiler von Oracle bis Version 9 die Aneinanderreihung mit `+` über Methoden der Klassen `String` und `StringBuilder`<sup>14</sup> um. Aus

```
String s = "a" + b + c;
```

generiert der Java-Compiler (vor Java 9):

<sup>14</sup> In älteren Java-Versionen, in denen es noch kein `StringBuilder` gab, kam `StringBuffer` zum Einsatz.

```
String s = new StringBuilder().append( "a" ).append( b ).append( c ).toString();
```

Das bedeutet: Die Konkatenation über + ist insbesondere in Schleifen nicht performant, hier sollten Entwickler vor der Schleife ein `StringBuilder` aufbauen und im Schleifenrumpf die `append(...)`-Methoden aufrufen. Tipp: Der neu aufgebaute `StringBuilder` sollte im Idealfall im Konstruktor gleich die passende Puffergröße bekommen. Ab Java 9 versucht die JVM, direkt die Konkatenation zu optimieren.

### 5.8.1 Strings mit StringJoiner zusammenhängen

Um Strings zu einem großen Ergebnis zusammenzuhängen und dabei einen gemeinsamen Trenner zu verwenden, bietet `String` die praktische Hilfsmethode `join(...)`. Dahinter steht eine kleine Klasse `StringJoiner`, die auch direkt genutzt werden kann.

#### Beispiel

```
StringJoiner sj = new StringJoiner( ", " );
sj.add( "1" ).add( "2" ).add( "3" );
System.out.println( sj.toString() ); // 1, 2, 3
```

[zB]

Der Delimiter – der natürlich auch ein Leerstring "" sein kann – wird zwischen jedes Element gesetzt, das hinzugefügt wurde. Die im Beispiel eingesetzte Methode `add(CharSequence)` nimmt einen Join-String vom Typ `CharSequence` an und liefert den aktuellen `StringJoiner` zurück, sodass sich die `add(...)`-Aufrufe kaskadieren lassen. Mit der Methode `merge(StringJoiner)` lässt sich der Inhalt eines anderen `StringJoiner` integrieren.

### Zusammenhängen mit Infix, Präfix und Suffix

Nicht nur das Trennzeichen selbst lässt sich angeben, sondern auch ein Startzeichen und Endzeichen.

#### Beispiel

Die Ausgabe soll mit einem »{« beginnen und mit einem »}« enden:

```
StringJoiner sj = new StringJoiner( " ", "{", "}" );
```

[zB]

### Nichts zum Zusammenhängen gegeben

Es kommt vor, dass dem `StringJoiner` nichts zum Zusammenfügen gegeben wird. Dann wird er dennoch Präfix und Suffix einsetzen:

```
StringJoiner sj = new StringJoiner( ", ", "{", "}" );
System.out.println( sj.toString() ); // {}
```

Ist das unerwünscht, gibt `setEmptyValue(CharSequence)` ein Substitut an, das genau dann zum Zuge kommt, wenn kein `add(...)` etwas dem `StringJoiner` hinzugefügt hat:

```
StringJoiner sj = new StringJoiner( ", ", "{", "}" ).setEmptyValue("<nix>");
System.out.println( sj.toString() ); // <nix>
```

Zusammengefasst bietet die Klasse zwei Konstruktoren und fünf Methoden:

```
class java.util.StringJoiner
```

- `StringJoiner(CharSequence delimiter)`
- `StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`
- `StringJoiner add(CharSequence newElement)`
- `StringJoiner merge(StringJoiner other)`
- `StringJoiner setEmptyValue(CharSequence emptyValue)`
- `int length()`
- `String toString()`

#### Hinweis

Eigentlich ist die Klasse `StringJoiner` ziemlich schwach, und selbst `String.join(...)` ermöglicht es, Zeichenketten als Sammlung vom Typ `Iterable` anzunehmen, was `StringJoiner`, die Implementierung von `String.join(...)`, selbst nicht erlaubt.

## 5.9 Zerlegen von Zeichenketten

Die Java-Bibliothek bietet einige Klassen und Methoden, mit denen wir nach bestimmten Mustern große Zeichenketten in kleinere zerlegen können. In diesem Kontext sind die Begriffe *Token* und *Delimiter* zu nennen: Ein Token ist ein Teil eines Strings, den bestimmte Trennzeichen (engl. *delimiter*) von anderen Tokens trennen. 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« usw.

Die Java-Bibliothek bietet eine Reihe von Möglichkeiten zum Zerlegen von Zeichenfolgen, von denen die zwei ersten in den nachfolgenden Abschnitten vorgestellt werden:

- ▶ `split(...)` von `String`: Aufteilen mit einem Delimiter, den ein regulärer Ausdruck beschreibt
- ▶ `lines()` von `String`: Liefert einen `Stream<String>` von Zeilen.
- ▶ `Scanner`: schöne Klasse zum Ablaufen einer Eingabe, auch zeilenweise
- ▶  `StringTokenizer`: der Klassiker aus Java 1.0. Delimiter sind nur einzelne Zeichen.
- ▶ `BreakIterator`: Findet Zeichen-, Wort-, Zeilen- oder Satzgrenzen.
- ▶ `Matcher` in Zusammenhang mit der `Pattern`-Klasse zerlegt mithilfe von regulären Ausdrücken.

Die Methoden und Klassen sind sozusagen die Gegenspieler der Konkatenationsmöglichkeiten: `split(...)` steht `join(...)` gegenüber, `StringTokenizer` dem `StringJoiner`.

### 5.9.1 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 Array bzw. ein Iterable mit Teilzeichenketten.

#### Beispiel

[zB]

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 `Pattern.quote(String)`, die einen »entschärften« Regex-String zurückgibt. Andernfalls liefert `split(".")` auf jedem String ein Array der Länge 0.

Ein häufiger Trenner ist `\s`, also Weißraum.

#### Beispiel

[zB]

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. Alternativ kann auch der Ausdruck `"[\\s\\p{Punct}]+"` eingesetzt 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.

### 5.9.2 Yes we can, yes we scan – die Klasse Scanner

Die Klasse `java.util.Scanner` kann eine Zeichenkette in Tokens zerlegen und einfache Dateien 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.

#### Aufbauen eines Scanners

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 ist das `System.in`), einem `Path`-Objekt oder diversen anderen Eingabenquellen. 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.

```
final class java.util.Scanner
implements Iterator<String>, Closeable
```

- `Scanner(String source)`
- `Scanner(Path source) throws IOException`
- `Scanner(Path source, String charsetName) throws IOException`
- `Scanner(Path source, Charset charset) throws IOException (Neu in Java 10)`
- `Scanner(File source) throws FileNotFoundException`
- `Scanner(File source, String charsetName) throws FileNotFoundException`
- `Scanner(File source, Charset charset) throws IOException (Neu in Java 10)`
- `Scanner(InputStream source)`
- `Scanner(InputStream source, String charsetName)`
- `Scanner(Readable source)`

- Scanner(ReadableByteChannel source)
- Scanner(ReadableByteChannel source, String charsetName)
- Scanner(ReadableByteChannel source, Charset charset) (neu in Java 10)
- Scanner(InputStream source, Charset charset) (neu in Java 10)

### Zeilenweises Einlesen einer Datei

Ist das Scanner-Objekt angelegt, lässt sich mit dem Paar `hasNextLine()` und `nextLine()` einfach eine Datei zeilenweise auslesen:

**Listing 5.12** src/main/java/com/tutego/insel/string/PrintAllLines.java, Ausschnitt

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class ReadAllLines {
    public static void main( String[] args ) throws FileNotFoundException {
        try ( Scanner scanner = new Scanner( Paths.get( "EastOfJava.txt" ),
                StandardCharsets.ISO_8859_1.name() ) ) {
            while ( scanner.hasNextLine() )
                System.out.println( scanner.nextLine() );
        }
    }
}
```

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. Die Konstruktion mit `try (...) { ... }` nennt sich *try mit Ressourcen* und schließt automatisch die Datei nach der Nutzung. Den Umgang mit Exceptions und das besondere `try` erklärt beides [Kapitel 8](#), »Ausnahmen müssen sein«, genauer. Auch sollte immer die Kodierung angegeben werden, die in unserem Fall ISO 8859-1, also Latin-1, ist. Der Kodierungs-String, der für den Scanner-Konstruktor nötig ist, stammt von einer Konstanten aus StandardCharsets.

```
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.

**Tipp**

Wenden wir Scanner auf einen String an, so kann dieser String vom Scanner Zeile für Zeile zerlegt werden. Seit Java 11 bietet die String-Methode `lines()` eine Alternative.

**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.)

**Beispiel**

Von der Standardeingabe soll ein String gelesen werden:

```
Scanner scanner = new Scanner( System.in );
String s = scanner.next();
```

Wichtig: Der Scanner sollte in diesem Fall nicht geschlossen werden!

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 dieses Typs folgt.

**Beispiel**

Die einzelnen `nextXXX()`- und `hasNextXXX()`-Methoden in einem Beispiel:

**Listing 5.13** src/main/java/com/tutego/insel/string/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
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) bzw. 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)  
Liefert true, wenn ein Token des gewünschten Typs gelesen werden kann.
- 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.

## 5.10 Ausgaben formatieren

Immer wieder müssen Zahlen, Datumsangaben und Text auf verschiedenste Art und Weise formatiert werden. Zur Formatierung bietet Java mehrere Lösungen:

- ▶ Die `format(...)`- und `printf(...)`-Methoden erlauben eine formatierte Ausgabe, so wie sie schon seit Urzeiten 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- und 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.

### 5.10.1 Formatieren und Ausgeben mit `format()`

Die Klasse `String` stellt mit der statischen Methode `format(...)` eine Möglichkeit bereit, Zeichenketten nach einer Vorgabe zu formatieren.



#### Beispiel

```
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 so genannte *Formatspezifizierer*, 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 Array direkt) mit den Werten, auf die sich die Formatspezifizierer beziehen.

Spezifizierer	Steht für ...	Spezifizierer	Steht für ...
%n	neue Zeile	%b	Boolean
%%	Prozentzeichen	%s	String
%c	Unicode-Zeichen	%d	Dezimalzahl
%x	Hexadezimal-schreibweise	%t	Datum und Zeit
%f	Fließkommazahl	%e	wissenschaftliche Notation

Tabelle 5.12 Die wichtigsten Formatspezifizierer im Überblick

**Tipp**

Der Zeilenvorschub ist vom Betriebssystem abhängig, und %n gibt uns ein gutes Mittel an die Hand, an dieses Zeilenvorschubzeichen (oder diese Zeichenfolge) zu kommen. Aber statt mit `String.format("%n")` den Separator zu erfragen, ist `System.lineSeparator()` eine bessere Lösung.

```
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 mit 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(...)`).



### Beispiel

Gib die Zahlen von 0 bis 15 hexadezimal aus:

```
for ( int i = 0x0; i <= 0xf; i++ )
    System.out.printf( "%x%n", i ); // 0 1 2 ... e f
```

Auch bei `printf(...)` ist als erstes Argument ein `Locale` möglich.

### Pimp my String mit Formatspezifizierern \*

Die Anzahl der Formatspezifizierer 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 oder 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` bzw. `%T` sind erlaubt: `Long`, `Calendar` und `Date`. `%t` benötigt zwingend ein Suffix.
- ▶ Den Hashcode schreibt `%h` bzw. `%H`. Beim Wert `null` ist auch das Ergebnis die Zeichenkette mit dem Inhalt »null«.

Zusätzliche Flags, etwa für Längenangaben und die Anzahl an Nachkommastellen, sind möglich und werden im folgenden Beispiel gezeigt:

**Listing 5.14** src/main/java/com/tutego/insel/string/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 );            // 12/12/12
o.printf( "%1$te. %1$tB%n" , t );   // 12. Dez

```

Im Fall von Fließkommazahlen werden diese nach dem RoundingMode.HALF\_UP gerundet, so dass 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 erscheint, und ein Leerzeichen besagt, ob dann bei positiven Zeichen ein Platz frei

bleibt. Eine öffnende Klammer setzt bei negativen Zahlen kein Minus, sondern setzt diese in Klammern.

[zB]

### Beispiel

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
```

### Formatspezifizierer für Datumswerte

Aus dem größeren 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 5.13 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 Jahrs (001–366)
%tM	zweistellige Minutenangabe (00–59)
%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

Tabelle 5.13 Suffixe für Datumswerte

Symbol	Beschreibung
%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 5.13 Suffixe für Datumswerte (Forts.)

### Positionsangaben

Im vorangegangenen Beispiel `PrintfDemo` lautete eine Zeile:

```
System.out.printf( "%te. %1$tb%n", t );      // 28. Okt
```

Der Formatierungsstring enthält eine Positionsangabe *Position\$*, bei der sich `1$` auf das erste Argument, `2$` auf das zweite usw. bezieht (interessant ist, dass hier die Nummerierung nicht bei null beginnt).

Die Positionsangabe im Format-String 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 dieser Lösung lässt sich `%1$te. %1$tb%n` schreiben, also auch für das erste Argument ausdrücklich die Position 1 vorschreiben.
- ▶ Die Reihenfolge der übergebenen Argumente bleibt in der Regel immer gleich, aber der Format-String kann die Argumente an unterschiedliche Positionen setzen.

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 5.15 src/main/java/com/tutego/insel/string/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 Format-String, 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:

```
LocalDate c1 = LocalDate.of( 1973, Month.MARCH, 1 );
LocalDate c2 = LocalDate.of( 1985, 9, 2 );
System.out.printf( "%te. %<tb %<ty, %2$te. %<tb %<ty%n",
                   c1,                           c2 ); // 1. März 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.

## 5.11 Zum Weiterlesen

Das erweiterte Insel-Buch »Java SE 9 Standard-Bibliothek« vertieft die Zeichenkettenverarbeitung und geht insbesondere auf reguläre Ausdrücke detailliert ein, die in Java an unterschiedlichen Stellen zu Tage treten. Dabei wird auch noch einmal ein Blick auf Scanner geworfen.

Wenn bei der Zeichenkettenverarbeitung sehr große Datenmengen verarbeitet werden, ist die Frage der Optimierung interessant. Die Standardimplementierung des JDK arbeitet nur mit einem einfachen Suchalgorithmus, der bei großen Mustern und Such-Strings sehr ineffizient ist. Hier hat die Informatik in den letzten Jahrzehnten sehr interessante Ansätze hervorgebracht, wie zum Beispiel den Optimal-Mismatch-Algorithmus. Ein gewisses Problem stellt aber der komplette Unicode-Standard dar, insbesondere seit Unicode 4 mit 32 Bit. Enterprise-Bibliotheken zum Beispiel zum Parsen von XML oder JSON unterstützen Unicode 4 in der Regel nicht.

# Kapitel 6

## Eigene Klassen schreiben

»Das Gesetz ist der abstrakte Ausdruck des allgemeinen an und für sich seienden Willens.«

– Georg Wilhelm Friedrich Hegel (1770–1831)

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.

### 6.1 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
- ▶ geschachtelte Klassen, Schnittstellen und 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 6.1** src/main/java/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. Details zum Konstruktor folgen in Abschnitt 6.5.3, »Der Standard-Konstruktor (default constructor)«.

### 6.1.1 Attribute deklarieren

Diese Player-Klasse hat bisher keine Attribute und kann daher nichts speichern. 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 6.2** src/main/java/com/tutego/insel/game/v2/Player.java, Player

```
class Player {
    String name;
    String item;
}
```

#### Hinweis

Bei Objektvariablen ist kein var möglich, die Datentypen müssen genannt sein.

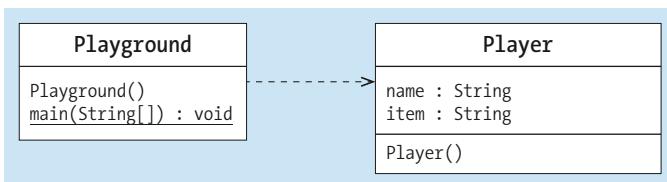
Eine zweite Klasse, Playground, erzeugt in der statischen main(...)-Methode für den mutigen Spieler ein Player-Objekt, schreibt und liest die Attribute:

**Listing 6.3** src/main/java/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 6.1** Das UML-Diagramm zeigt Abhängigkeiten von Playground und Player.



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]+[Esc]**. 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]+[Tab]** drücken. Das Ganze lässt sich auch für PREVIOUS EDITOR und **[Strg]+[Shift]+[Tab]** wiederholen.

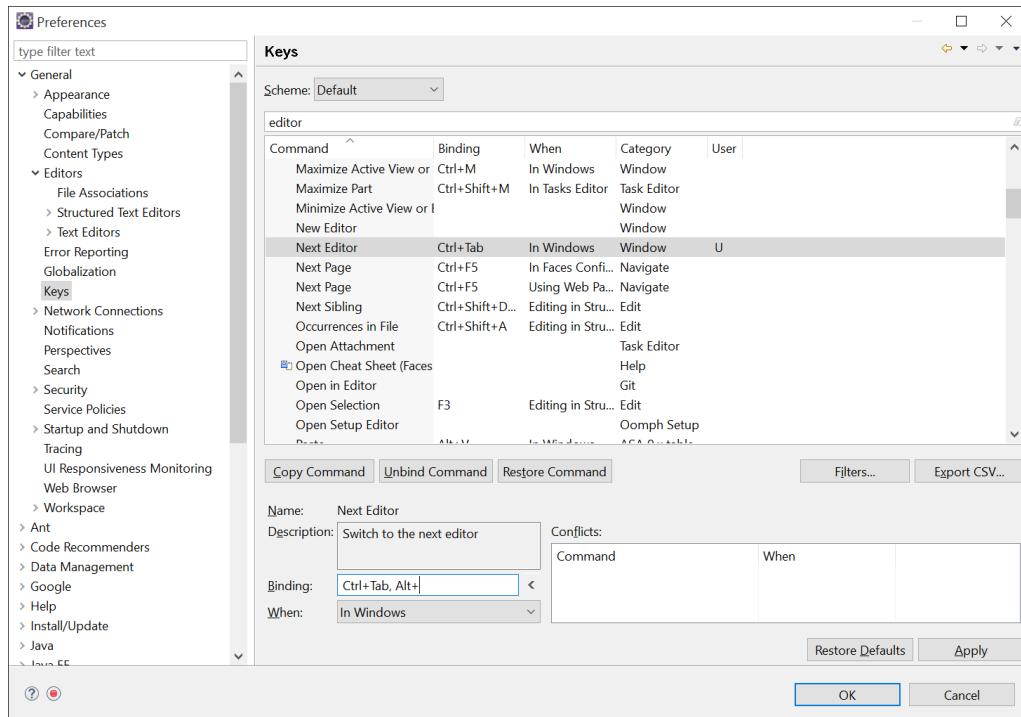


Abbildung 6.2 Tastaturkürzel in Eclipse einstellen

### 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. Bezeichner können auch nicht so heißen wie Schlüsselwörter.

### 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 = "";
}
```

Es würde ein `"".equals(new Player().name)` also true ergeben.



### Designtipp

Es ist eine Gratwanderung, einfache Datentypen zu verwenden oder doch andere Objekte zu referenzieren. Wenn Player z. B. eine Postleitzahl speichert, wäre ein `int` (oder `String`) problematisch, da der Spieler auch eine PLZ-Validierung durchführen müsste, was nicht seine Aufgabe ist. Die allgemeine OOP-Regel lautet: Setze bei Daten, die validiert werden müssen und vielleicht auch mit anderen Attributen zusammenhängen, einen neuen Typ ein.

### 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, an dem das Objekt mit `new` aufgebaut wurde, und sie lebt so lange, bis der Garbage-Collector das Objekt wegräumt. Sichtbar und gültig ist die Variable aber immer im gesamten Objekt und in allen Blöcken.<sup>1</sup>

#### 6.1.2 Methoden deklarieren

Zu Attributen gesellen sich Methoden, die üblicherweise auf den Objektvariablen arbeiten. Geben wir dem Spieler zwei Methoden:

- ▶ `carry(String newItem)`: Der Spieler soll einen neuen Gegenstand tragen.
- ▶ `doesCarry(String anItem)`: Fragt an, ob der Spieler einen Gegenstand trägt.

Anders als unsere bisherigen statischen Methoden werden sie für Player Objektmethoden sein, also den Modifizierer `static` nicht tragen.

**Listing 6.4** src/main/java/com/tutego/insel/game/v3/Player.java, Player

```
class Player {

    String name = "";
    String item = "";
}
```

---

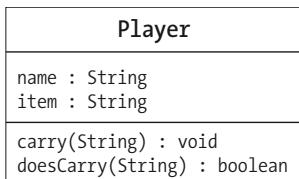
<sup>1</sup> Das gilt nicht für statische Methoden und statische Initialisierungsblöcke, aber diese werden erst später vorgestellt.

```

void carry( String newItem ) {
    if ( newItem != null && !newItem.trim().isEmpty() )
        item += newItem + ";";
}

boolean doesCarry( String anItem ) {
    if ( anItem == null )
        return false;
    return (";" + item).contains( ";" + anItem + ";" );
}
}

```



**Abbildung 6.3** Das Klassendiagramm von Player zeigt zwei Attribute und zwei Methoden.

Die Methode `carry(...)` fragt zunächst ab, ob der neu aufzunehmende Gegenstand `newItem` – der hier bei uns ein String ist – gültig ist, also nicht `null` und nicht leer ist. Ist er gültig, wird er an die existierende String-Objektvariable `item` angehängt und mit einem Semikolon abgeschlossen – das trennt die Einträge. Die Abfragemethode `doesCarry(...)` wird bei `null` das Ergebnis `false` liefern und andernfalls testen, ob `anItem` in `item` vorkommt. Wir nutzen allerdings kein einfaches `contains(...)`, denn sonst würde bei einem `item` wie »Sprechende Socke« eine Abfrage wie `"Sprechende Socke;".contains("Socke;")` wahr ergeben, was falsch ist. Der Trick ist, ein künstliches Startzeichen `";"` voranzustellen, denn dann ist `";Sprechende Socke;".contains(";Socke;")` falsch.

Testen wir die Methoden mit einem Spieler:

**Listing 6.5** src/main/java/com/tutego/insel/game/v3/Hogwurz.java, main()

```

Player parry = new Player();
parry.name = "Parry Hotter";
System.out.printf( "%s trägt: %s%n",
                   parry.name, parry.item );
parry.carry( "Denkarium" );
System.out.printf( "%s trägt: %s%n", parry.name, parry.item );
parry.carry( "Das goldene Ei" );
System.out.printf( "%s trägt: %s%n", parry.name, parry.item );
System.out.println( parry.doesCarry( "Denkarium" ) );

```

```
System.out.println( parry.doesCarry( "Stein der Weisen" ) );
System.out.println( parry.doesCarry( "Denk" ) );
System.out.println( parry.doesCarry( "Ei" ) );
```

Wie zu erwarten, ist die Ausgabe:

```
Parry Hotter trägt:  
Parry Hotter trägt: Denkarium;  
Parry Hotter trägt: Denkarium;Das goldene Ei;  
true  
false  
false  
false
```



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 zum 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.

### 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 `doesCarry(...)` direkt auf das nötige Attribut `item` zu, um die Programmlogik auszuführen. Dies wird oft für Nebeneffekte (Seiteneffekte) genutzt. Die Methode `carry(...)` ändert ausdrücklich eine Objektvariable und verändert so den Zustand des Objekts. `doesCarry()` 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, sodass 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-Programmen abgebildet wird. Dazu ein Beispiel: Die Klasse `PlayerData` ist ein reiner Datencontainer für den Zustand, aber Verhalten wird hier nicht deklariert:

**Listing 6.6** src/main/java/com/tutego/insel/oop/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 6.7** src/main/java/com/tutego/insel/oop/PlayerFunctions.java

```
class PlayerFunctions {

    static void carry( PlayerData data, String newItem ) {
        if ( newItem != null && !newItem.trim().isEmpty() )
            data.item += newItem + ";";
    }

    static boolean doesCarry( PlayerData data, String anItem ) {
        if ( anItem == null )
            return false;
        return (";" + data.item).contains( ";" + anItem + ";" );
    }
}
```

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 nichtobjektorientierte Ansatz gut zu sehen. Setzen wir links den falschen und rechts den korrekt objektorientiert modellierten Weg ein:

Prozedural	Objektorientiert
<pre>PlayerData parry = new PlayerData(); parry.name = "Parry Hotter"; PlayerFunctions.hasCompoundName( parry ); PlayerFunctions.clearName( parry );</pre>	<pre>Player parry = new Player(); parry.name = "Parry Hotter"; parry.hasCompoundName(); parry.clearName();</pre>

**Tabelle 6.1** 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.

### 6.1.3 Verdeckte (shadowed) Variablen

Führt eine Methode eine neue Variable ein, so kann der gleiche Variablenname nicht noch einmal in einem inneren Block verwendet werden. Folgendes führt zu einem Compilerfehler:

```
public static void main( String[] args ) {
    int args = 1;      // ☠ Duplicate local variable args
}
```

Beim Verlassen eines Blocks ist der Variablenname jedoch wieder frei, Folgendes ist daher in Ordnung:

```
for ( int i = 0; i < 10; i++ )
    System.out.println( i );
for ( int i = 0; i < 10; i++ )
    System.out.println( i );
```

Grundsätzlich keine Probleme gibt es, wenn eine lokale Variable/Parametervariable genauso heißt wie eine Objektvariable. Wir sprechen in diesem Fall von einer *verdeckten Variablen* – im Englischen *shadowed variable* genannt.<sup>2</sup> Der Compiler wird dann die lokale Variable nutzen und nicht mehr die Objektvariable.



#### Beispiel

Die lokale Variable name aus `toString()` verdeckt die Objektvariable name.

```
class Player {
    String name, item;

    public String toString() {
        String name = "Der Experte";
        return name + " trägt" + item; // Der Experte trägt ...
    }
}
```

Verdeckte Variablen sind praktisch, denn ohne so ein Konzept könnte eine neu eingefügte Objektvariable schnell viele Methoden kaputt machen, die zufälligerweise im Rumpf ebenfalls den gleichen Namen nutzen.<sup>3</sup>

Ein Nachteil ist das nur, wenn wir die lokale Variable gleichzeitig mit der Parametervariablen nutzen wollen, doch dafür gibt es Lösungen.

2 <https://docs.oracle.com/javase/specs/jls/se11/html/jls-6.html#jls-6.4>

3 Fast alle Programmiersprachen besitzen verdeckte Variablen, eine Ausnahme ist TypeScript.

### 6.1.4 Die this-Referenz

In jeder Objektmethode und jedem Konstruktor<sup>4</sup> 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 oder 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 haben wir eben im vorherigen Abschnitt gesehen. Das ist so weit nicht tragisch und wird nur dann zum Problem, wenn später Zugriff auf die Objektvariable nötig ist.

Das folgende Beispiel deklariert in der `WiFi`-Klasse eine Objektvariable `password` sowie eine Methode `authenticate(String password)`; das heißt, die Parametervariable heißt genauso wie die Objektvariable. Jeder Zugriff auf `password` in der Methode bezieht sich auf die Parametervariable und nicht auf die Objektvariable. Der folgende Versuch einer Authentifizierung, in der eigentlich die Parametervariable mit der Objektvariablen verglichen werden soll, ist also semantisch falsch, wobei syntaktisch kein Problem besteht und die Laufzeitumgebung sogar immer `true` gibt, wenn `password` nicht `null` ist.

**Listing 6.8** src/main/java/com/tutego/insel/oop/WiFi.java

```
class WiFi {
    String password = "+covfefe";
    boolean isAuthenticated;

    boolean authenticate( String password ) {
        return isAuthenticated = password.equals( password ); // semantisch falsch!
    }

    boolean isAuthenticated() {
        return isAuthenticated;
    }
}
```

---

<sup>4</sup> Sowie Exemplarinitialisierer, doch der Code landet im Konstruktor.

Das heißt aber nicht, dass auf die Objektvariable nicht mehr zugegriffen werden kann. Die `this`-Referenz zeigt auf das aktuelle Objekt, und damit ist auch ein Zugriff auf Objekteigenschaften jederzeit möglich:

```
public boolean authenticate( String password ) {
    return isAuthenticated = this.password.equals( password );
}
```

### this für Setter nutzen

Häufiger Einsatzort für das `this` sind Methoden oder Konstruktoren, die Zustände initialisieren. Gerne nennen Entwickler die Parametervariablen so wie die Exemplarvariablen, um damit eine starke Zugehörigkeit auszudrücken. Schreiben wir also eine Methode `setName(String)`:

```
class Player {

    String name;

    void setName( String name ) {
        this.name = name;
    }
}
```

Das an `setName(String)` übergebene Objekt 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.

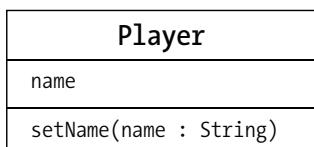


Abbildung 6.4 Klassendiagramm für Player mit Setter

### 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 können also Methoden kaskadiert anhängen oder es bleiben lassen.

Wir wollen diese Möglichkeit bei einem Spieler programmieren, sodass die Methoden `name(String)` und `item(String)` Spielername und Gegenstand zuweisen. Beide Methoden liefern ihr eigenes Player-Objekt über die `this`-Referenz zurück:

**Listing 6.9** src/main/java/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 6.5** Player-Klasse mit kaskadierbaren Methoden

Erzeugen wir einen Player, und kaskadieren wir einige Methoden:

**Listing 6.10** src/main/java/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 internem Zustand bekommt.



### Hinweis

Bei Anfragemethoden könnten wir versucht sein, diese praktische Eigenschaft überall zu verwenden. Üblicherweise sollten Objekte jedoch ihre Eigenschaften nach der JavaBeans-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.

### Sich selbst mit this übergeben

Möchte sich ein Objekt A einem anderen Objekt B mitteilen, damit B das andere Objekt A »kennt«, so funktioniert das gut mit der `this`-Referenz. Demonstrieren wir das an einem »Ich bin dein Vater«-Beispiel: Zwei Klassen `Luke` und `Darth` repräsentieren zwei Personen, wobei `Luke` ein Attribut `dad` für seinen Vater hat:

**Listing 6.11** src/main/java/com/tutego/insel/oop/LukeAndDarth.java, Teil 1

```
class Luke {
    Darth dad;
}

class Darth {
    void revealTruthTo( Luke son ) {
        son.dad = this;
    }
}
```

Spannend ist die Methode `revealTruthTo(Luke)`, denn sie setzt beim übergebenen Luke-Objekt das `dad`-Attribut mit der `this`-Referenz. Damit kennt Luke seinen Vater, getestet in folgender Klasse:

**Listing 6.12** src/main/java/com/tutego/insel/oop/LukeAndDarth.java, Teil 2

```
public class LukeAndDarth {
    public static void main( String[] args ) {
        Luke luke = new Luke();
        Darth darth = new Darth();
        System.out.println( luke.dad ); // null
        darth.revealTruthTo( luke );
        System.out.println( luke.dad ); // Darth@15db9742
    }
}
```

### Hinweis

In statischen Methoden steht die `this`-Referenz nicht zur Verfügung, da wir uns in Klassenmethoden nicht auf ein konkretes Objekt beziehen.



## 6.2 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 `paketsichtbar` (ohne Modifizierer) erklärt werden; auf `protected` (geschützt) kommen wir in [Abschnitt 7.2](#), »Vererbung«, zurück.

### 6.2.1 Für die Öffentlichkeit: `public`

Der Sichtbarkeitsmodifizierer `public` an Klassen, Konstruktoren, Methoden und sonstigen Klasseninnereien bestimmt, dass alle diese markierten Elemente von außen für jeden sicht-

bar 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«.

### 6.2.2 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(String, String)` soll eine Änderung des Passwortes zulassen, wenn das alte Passwort bekannt ist, und eine weitere öffentliche Methode `check(String)` erlaubt das Prüfen eines Passwortes. Am Anfang ist das Passwort der leere String:

**Listing 6.13** src/main/java/com/tutego/insel/oop/Password.java

```
public class Password {

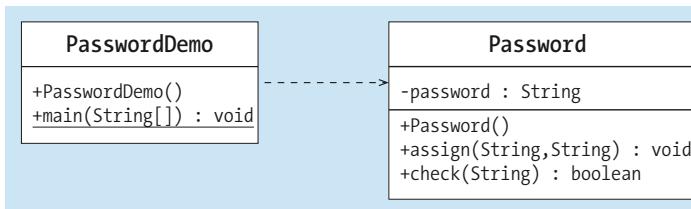
    private String password = "";

    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 6.6** In der UML werden private Eigenschaften mit einem führenden Minus gekennzeichnet.

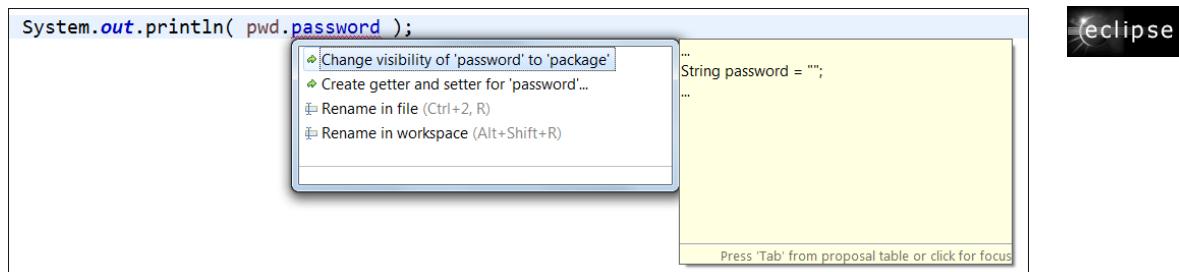
Eine zweite Klasse PasswordDemo will nun auf das Passwort von außen zugreifen:

**Listing 6.14** src/main/java/com/tutego/insel/oop>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- bzw. Laufzeit Verstöße und meldet diese.



**Abbildung 6.7** Bei einem durch die falsche Sichtbarkeit verursachten Fehler bietet Eclipse über die Tastenkombination **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.

### 6.2.3 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. Da 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.

#### 6.2.4 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 6.15** src/main/java/com/tutego/insel/oop/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(Key)` 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(Key)`-Methode in der `Key`-Klasse deklariert und der Parameter ebenfalls vom Typ `Key` ist. Mit Unterklassen (siehe [Abschnitt 7.2.1, »Vererbung in Java«](#)) funktioniert das schon nicht mehr. Private Attribute und Methoden sind also gegen Angriffe von außerhalb der deklarierenden Klasse geschützt.

#### 6.2.5 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, wer-

den nur in der Bibel genannt. Wenn wir das Alter privat machen, kann eine Zugriffsme~~thode~~ 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 die Spielerklasse 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. Die interne Arbeitsweise sollte durch `private` (Hilfs-)Methoden geheim gehalten werden, und das gilt auch für Variablen und Variablentypen. Möchten Entwickler etwa ihr internes Attribut von `char` in `String` ändern und damit beliebig lange Zeichenketten verwalten, hätten wir ein beträchtliches Problem, da an jeder Stelle des Vorkommens ein Objekt eingesetzt werden müsste. Wollten wir zwei Variablen einführen – ein `char`, damit die alte, derzeit benutzte Software ohne Änderung auskommt, und einen neuen `String` –, hätten wir ein Konsistenzproblem.
- ▶ Nicht immer müssen dem Aufrufer haarklein alle Eigenschaften angeboten werden. Der Nutzer möchte so genannte *höherwertige Dienste* vom Objekt angeboten bekommen, so dass der Zugriff auf die unteren Attribute vielleicht gar nicht nötig ist.

### 6.2.6 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 werden eine Schreib- und eine 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 des Setters sein.

Bei boolean-Attributten 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()`, `isVolljährig()` oder `set 顔文字 (String 顔文字)`.

### Sprachvergleich

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.<sup>5</sup> 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 6.16** src/main/java/com/tutego/insel/game/v5/Player.java, Player

```
public class Player {

    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;
    }
}
```

---

<sup>5</sup> Klammern bei Methodenaufrufen sind optional. Außerdem gilt: In Java testet der `==`-Operator die Identität und `equals()` die Gleichheit. In Ruby ist das genau andersherum: `==` testet auf Gleichheit und `equal?` auf Identität. Und dazu kommt die Methode `eq?`, die testet, ob zwei Objekte die gleichen Werte haben und vom gleichen Typ sind.

```

public void setItem( String item ) {
    if ( item != null && !item.trim().isEmpty() )
        this.item = item;
}
}

```

Statt sich bei ungültigen Werten taub zu stellen, können Sie alternativ loggen oder etwa in Form einer Ausnahme (Exception) unerwünschte Werte melden.

Player	
-	name item
+	getName() : String setName(name : String) getItem() : String setItem(item : String)

Abbildung 6.8 Player-Klasse mit Setter/Getter

Der Nutzer der Klasse muss wegen der Methodenaufrufe etwas mehr schreiben:

Listing 6.17 src/main/java/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 zu Tage, 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.



### 6.2.7 Paketsichtbar

Die Sichtbarkeiten public und private sind Extreme. Steht kein ausdrücklicher Sichtbarkeitsmodifizierer an den Eigenschaften, gilt die *Paketsichtbarkeit*. Sie sagt aus, dass die paket-

sichtbaren 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 6.9 Vier Klassen, verteilt auf zwei Pakete, für das Beispiel

Listing 6.18 src/main/java/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 6.19 src/main/java/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 deklariert sie 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 6.20 src/main/java/com/tutego/insel/protecteda/C.java

```
package com.tutego.insel.protecteda;
public class C {
    static int c;
}
```

Und dies ist die Klasse D:

Listing 6.21 src/main/java/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.

### 6.2.8 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. In Java gibt es wegen des Modulsystems allerdings die Einschränkung, dass öffentliche Typen auch exportiert werden müssen, sonst sind sie für andere Module dennoch 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 geschachtelte 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 geschachtelte 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 Standardsichtbarkeit 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 Standardsichtbarkeit reichen) oder aus einem anderen Paket. Eine Kombination aus `private` `protected` wäre wünschenswert, um die Eigenschaften nur für die

Unterklassen sichtbar zu machen und nicht gleich für die Klassen aus dem gleichen Paket.<sup>6</sup>

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 6.2 Wer sieht welche Eigenschaften bei welcher Sichtbarkeit?

Der Einsatz der Sichtbarkeitsstufen über die Schlüsselwörter public, private und protected und den Standard »paketsichtbar« 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
#	geschützt (protected)
~	paketsichtbar

Tabelle 6.3 UML-Symbole für die Sichtbarkeit von Attributen und Operationen

<sup>6</sup> Die Java Programmers' FAQ (<http://tutego.de/go/privateprotected>) führen 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.«



### 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. Zwar ist die Reihenfolge im Grunde egal, doch 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 an das Ende der Zeile oder alleinstehend in die nächste Zeile. In Eclipse und IntelliJ kann ein Codeformatierer den Quellcode automatisch korrigieren.

## 6.3 Eine für alle – 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.random()` liefert eine Zufallszahl.
- ▶ `Math.sin(...)` berechnet den Sinus.
- ▶ `Integer.parseInt(...)` konvertiert einen String in eine Ganzzahl.
- ▶ `JOptionPane.showInputDialog(...)` zeigt einen Eingabedialog.
- ▶ `Color.HSBtoRGB(...)`<sup>7</sup> konvertiert Farben vom HSB-Farbraum in den RGB-Farbraum.

Dazu gesellen sich Zustände, die nicht an ein individuelles Objekt gebunden sind:

- ▶ `Math.PI` bestimmt die Zahl 3,1415...
- ▶ `Integer.MAX_VALUE` ist die größte darstellbare `int`-Ganzzahl.
- ▶ `Font.MONOSPACED` steht für eine Schriftart fester Breite.
- ▶ `MediaSize.ISO.A4` definiert die Größe einer DIN-A4-Seite, nämlich 210 mm × 297 mm.

Diese genannten Eigenschaften sind keinem konkreten Objekt zugeordnet, sondern vielmehr der Klasse. Diese Art von Zugehörigkeit unterstützt Java durch *statische Eigenschaften*. 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_VALUE` ist ein statisches Attribut der Klasse `Integer` und `out` der Klasse `System`.

### 6.3.1 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 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(String)` 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.

---

<sup>7</sup> Ja, die Methode beginnt unüblicherweise mit einem Großbuchstaben.

### 6.3.2 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.

- ▶ 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. Zum Testen greifen wir auf `matches("\\w+")` zurück, eine String-Methode, die mit einem regulären Ausdruck prüft, ob der gesamte String nur Wortzeichen<sup>8</sup> enthält.
- ▶ 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 6.22** src/main/java/com/tutego/insel/oop/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
GameUtils()
isGameIdentifier(String) : boolean

**Abbildung 6.10** Statische Eigenschaften werden in der UML unterstrichen.

Die statischen Eigenschaften werden mit dem Klassennamen `GameUtils` angesprochen:

**Listing 6.23** src/main/java/com/tutego/insel/oop/GameUtilsDemo.java, Ausschnitt

```
System.out.println( GameUtils.isGameIdentifier( "Superpig" ) ); // true
System.out.println( GameUtils.isGameIdentifier( "Superpig II" ) ); // false
```

---

<sup>8</sup> Laut <https://docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern.html> ist `\w` gleichwertig zu `[a-zA-Z_0-9]`, enthält also zum Beispiel deutsche Umlaute nicht.



Eclipse stellt statische Eigenschaften kursiv dar.



### Tipp

Falls eine Klasse nur statische Eigenschaften deklariert, spricht nichts dagegen, einen privaten Konstruktor anzugeben – das verhindert den äußeren Aufbau von Objekten. [Abschnitt 6.5.1, »Konstruktoren schreiben«](#), erklärt Konstruktoren genauer und erläutert auch weitere Anwendungsfälle für private Konstruktoren.

### Gültigkeitsbereich, Sichtbarkeit und Lebensdauer

Bei statischen und nichtstatischen Variablen können wir deutliche Unterschiede in der Lebensdauer festmachen. Eine Objektvariable beginnt ihr Leben mit dem new, und sie endet mit der automatischen Speicherbereinigung, dem Garbage-Collector. Eine statische Variable dagegen beginnt ihr Leben in dem Moment, in dem die Laufzeitumgebung die Klasse lädt und initialisiert. Das Leben der statischen Variablen 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.

### 6.3.3 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:

```
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 nichtstatisch ist. Das zu wissen ist aber wichtig. Aus diesem Grund sollten wir statische Eigenschaften immer über ihren Klassennamen ansprechen.



Eclipse gibt eine Meldung aus, wenn statische Eigenschaften über eine Referenz verwendet werden, und bietet ein Refactoring an.



### Hinweis

Bei statischen Zugriffen spielt die Referenz keine Rolle, und sie kann auch null sein. Wer im Wettbewerb um das schlechteste Java-Programm weit vorn sein möchte, der schreibt:

```
System.out.println( ((GameUtils) null).MAX_ID_LEN );
```

### 6.3.4 Warum die Groß- und Kleinschreibung wichtig ist \*

Die Vorgabe der Namenskonvention besagt: Klassennamen sind mit Großbuchstaben zu vergeben und Variablen-/Methodennamen mit Kleinbuchstaben (sofern die Variable keine Konstante beschreibt). Treffen wir auf eine Anweisung wie `Math.random()`, so wissen wir sofort, dass `random()` 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 Variablen 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 wäre und `move()` eine Objektmethode. Wir sehen an diesem Beispiel, dass es wichtig ist, sich an die Groß-/Kleinschreibung zu halten.

### 6.3.5 Statische Variablen zum Datenaustausch \*

Die Belegung einer statischen Variablen 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 ein Attribut `PI` statisch und `size` ein Objektattribut, so ergibt sich bei zwei Exemplaren folgendes Bild:

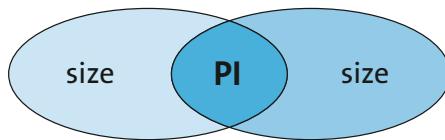


Abbildung 6.11 Zwei Klassen teilen sich das statische Attribut `PI`.

Während also beide Exemplare das gleiche `PI` nutzen, können beide Exemplare `size` völlig unterschiedlich belegen.

### Aufgebaute Exemplare mitzählen

Mit diesem Wissen wird es möglich, einen Austausch von Informationen über die Objektgrenze hinaus zu erlauben. Wir wollen das in einem Beispiel nutzen, in dem der Konstruktor die Anzahl der erzeugten Objekte mitzählt; eine statische Methode liefert später die bis dahin gebauten Exemplare:

Listing 6.24 src/main/java/com/tutego/insel/oop/Rollercoaster.java

```
public class Rollercoaster {

    private static int numberofInstances;

    {
        numberofInstances++;
    }

    public static int getNumberofInstances() {
        return numberofInstances;
    }

    public static void main( String[] args ) {
        new Rollercoaster();
        new Rollercoaster();

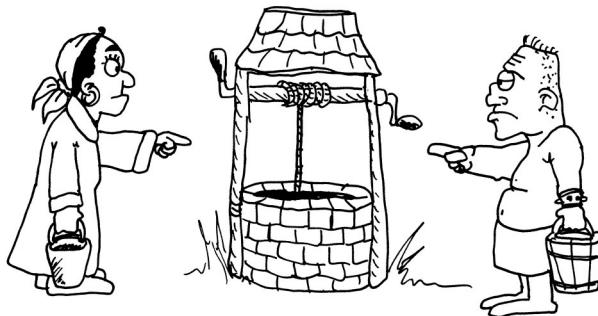
        System.out.println( Rollercoaster.getNumberofInstances() ); // 2
    }
}
```

Die statische Variable `numberofInstances` wird bei jedem neuen Exemplar über den Konstruktor hochgesetzt. Direkt ausgeschrieben ist der Konstruktor nicht, sondern es findet ein Exemplarinitialisierer Anwendung – siehe dazu [Abschnitt 6.6.1, »Initialisierung von Objekten«](#).

variablen» –, da der Compiler den Code automatisch in jeden Konstruktor kopiert. Das hat den Vorteil, dass Entwickler später problemlos neue Konstruktoren für den Rollercoaster hinzufügen können, ohne das Inkrement der statischen Variablen immer im Hinterkopf behalten zu müssen.

### 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 wohlüberdacht sein.



### 6.3.6 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; auf welches Objekt sollte `this` auch zeigen?

**Listing 6.25** src/main/java/com/tutego/insel/oop/InStaticNoThis.java, Ausschnitt

```
class InStaticNoThis {

    String name;

    static void setName() {
        name = "Amanda";           // 💀 Compilerfehler
        // Cannot make a static reference to the non-static field name
        System.out.println( this ); // 💀 Compilerfehler
        // Cannot use this in a static context
    }
}
```

Auch im statischen Initialisierungsblock – siehe dazu [Abschnitt 6.6.2](#), »Statische Blöcke als Klasseninitialisierer« – ist this nicht erlaubt.

## 6.4 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 Maße einer DIN-A4-Seite. Die Werte 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:

- ▶ Selbstdefinierte öffentliche statische finale Variablen nehmen konstante Werte auf.
- ▶ Aufzählungen über ein `enum` (die intern aber auch nur öffentliche finale statische Werte sind)

### 6.4.1 Konstanten über 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 Variablen nur einmal ein Wert zugewiesen werden darf. Für Variablen bedeutet dies: Es sind Konstanten; jeder spätere Schreibzugriff wäre ein Fehler. In der Regel sind die Konstanten öffentlich, aber natürlich können sie auch privat sein, wenn sie nur die Klasse etwas angehen.

Konstante Werte haben wir schon bei GameUtils eingesetzt:

**Listing 6.26** 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.

### 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!" );
```

## 6.4.2 Typsichere Aufzählungen

Konstanten sind eine wertvolle Möglichkeit, den Quellcode aussagekräftiger und klarer zu gestalten, was wichtig ist, denn Quellcode wird öfter gelesen als geschrieben. Oftmals finden sich Konstanten für mathematische Konstanten oder Größenbeschränkungen.

Eine besondere Form bilden Konstanten, wenn sie als Elemente von Aufzählungen verwendet werden. Aufzählungen erinnern an abgeschlossene Mengen, so wie:

- ▶ Tage der Woche (Montag, Dienstag ...)
- ▶ Monate eines Jahrs (Januar, Februar ...)
- ▶ Font-Stil (fett, kursiv ...)
- ▶ vordefinierte Linienmuster (durchgezogen, gestrichelt ...)

Die Tage der Woche und auch Monate des Jahrs werden zum Beispiel von der Klasse `java.util.Calendar` über öffentliche statische `int`-Konstanten angeboten.

[zB]

### Beispiel

Die Frau wird im Juni schwanger. Wann müssen Windeln gekauft werden?

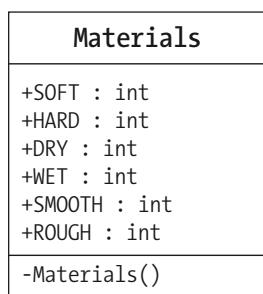
```
int month = Calendar.JUNE;
int conception = (month + 9) % 12;
System.out.println( conception ); // 2
```

Soll eine Klasse `Materials` zum Beispiel Konstanten für die Beschaffenheit eines Materials deklarieren, kann das so aussehen:

**Listing 6.27** src/main/java/com/tutego/insel/oop/Materials.java

```
public class Materials {
    private Materials() { } // Privater Konstruktor
    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`.



**Abbildung 6.12** UML-Diagramm der Klasse `Materials` mit Konstanten

### Problem mit dem Datentyp int auf Konstantentyp

Einfache Konstantentypen – wie bei uns `int` – bringen den Nachteil mit sich, dass die Konstanten nicht unbedingt von jedem angewendet werden müssen und ein Programmierer die Zahlen oder Zeichenketten eventuell direkt einsetzt. Das ist in dem Moment problematisch,

sobald sich die Belegung einer Konstanten einmal ändert. Bei einer Konstanten wie `Math.PI` wird das nicht passieren, aber die Maximallänge eines Strings kann durchaus einmal angepasst werden, genauso wie der Materialtyp, wenn zum Beispiel ein neues Material eingeschoben wird.

Im Idealfall haben Aufzählungen einen eigenen Typ. Sind sie »nur« vom Typ `int`, führt das leicht zu Fehlern. Die `Font`-Klasse der Java-API illustriert das Problem: Die Parametertypen beim Konstruktor sind: `Font(String, int, int)`. Versagt unser Gedächtnis, für was welches `int` steht, heißt es im Aufruf vielleicht:

```
Font f = new Font( "Dialog", 12, Font.BOLD );
```

Leider ist dies falsch, denn die Argumente für die Größe und den Schriftstil sind vertauscht: Es müsste `new Font( "Dialog", Font.BOLD, 12 )` heißen, denn das erste `int` steht für den Stil, der zweite `int` für die Größe. `Font.BOLD` ist eine `int`-Konstante.

Konstanten sind nur Namen für Werte eines frei zugänglichen Grundtyps (hier `int`), und nur die Variablenbelegung, also der Wert, wird an den Konstruktor übergeben. Niemand kann verbieten, dass die Werte direkt eingetragen werden. Das führt dann zu Fehlern wie im oberen Fall, in dem `12` die Ganzzahl für den Schriftstil ist, obwohl es dafür nur die Werte `0, 1, 2` geben sollte. Mit Zeichenketten als Werten der Konstanten kommen wir der Lösung auch nicht näher, wohl aber, wenn der Typ kein `int` wäre, sondern ein Objekttyp, denn der würde sich vom `int` unterscheiden.

### Hinweis

Ganzzahlen haben aber durchaus ihren Vorteil, wenn es verknüpfte 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. Hier muss nicht auf eigene Klassendeklarationen zurückgegriffen werden, sondern Java bietet ein eigenes Sprachmittel.

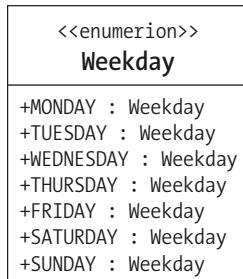
#### 6.4.3 Aufzählungstypen: typsichere Aufzählungen mit enum

Damit Konstanten von Aufzählungen einen eigenen Typ bekommen und nicht mehr Ganzzahlen oder Strings sind, bietet Java ein Sprachkonstrukt über das Schlüsselwort `enum`. Die Schreibweise für Aufzählungen erinnert ein wenig an die Deklaration von Klassen und Variablen, nur dass das Schlüsselwort `enum` statt `class` gebraucht wird und dass die Variablen automatisch statisch und öffentlich sind und kein Typ angegeben ist. Aufzählungen für Wochentage sind ein gutes Beispiel:

**Listing 6.28** src/main/java/com/tutego/weekday/Weekday.java, Weekday

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Weekday ist ein *Aufzählungstyp*. Die Konstantennamen werden wie üblich großgeschrieben, so wie auch statische Variablen, die als Konstanten benutzt werden, großgeschrieben werden.

**Abbildung 6.13** Eine Aufzählung in UML wird über den Stereotyp ausgedrückt.

### Aufzählungen nutzen

Um zu verstehen, wie sich Aufzählungstypen 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 statische Variablen (Konstanten) vom Typ der Aufzählung:

```
public class Weekday {
    public static final Weekday MONDAY = new Weekday( ... );
    public static final Weekday TUESDAY = new Weekday( ... );
    ...
}
```

Jetzt ist es einfach, diese Aufzählungen 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 – weiterverarbeiten lassen.

```
if ( day == Weekday.MONDAY )
    System.out.println( "I hate Mondays' (Garfield)" );
```

Auch implementieren die enum-Konstanten die `toString()`-Methode, die den Namen der Konstanten liefert.

## Geschichte

Sun reservierte zu Beginn der Entwicklung von Java diverse Schlüsselwörter, 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 Variablen-typ `java.util.Enumeration` war gern der Variablenname `enum` gewählt worden.

## Aufzählungsvergleiche mit ==

Wie die Umsetzung der Aufzählungstypen zeigt, wird für jede Konstante ein Objekt konstruiert, und das sind so genannte *Singletons*, also Objekte, die nur einmal erzeugt werden. Eigene neue Aufzählungsobjekte 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.

## 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 auch bei `switch` auf Strings verfolgt.

Initialisieren wir eine Variable vom Typ `Weekday`, und nutzen wir eine Fallunterscheidung mit der Aufzählung für einen Test auf das Wochenende:

**Listing 6.29** src/main/java/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 vom Aufzählungstyp 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 Referenzvariablen 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; das kann die gegebene Hilfsmethode Objects.requireNonNull(...) übernehmen:

```
public void setWeekday( Weekday day ) {
    this.day = Objects.requireNonNull( day, "Weekday day darf nicht null sein" );
}
```

### Aufzählungstypen als geschachtelten Typ deklarieren \*

Es gibt »normale« Aufzählungen, die an normale Klassen erinnern, und auch »geschachtelte« Aufzählungen, die in einen anderen Typ hineingesetzt werden. Mit anderen Worten: Weekday kann auch innerhalb einer anderen Klasse/anderen Schnittstelle deklariert werden. Ist die geschachtelte 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. Aufzählungen innerhalb von Typen sind immer implizit statisch, das Schlüsselwort static ist also nicht nötig.



#### Beispiel

Beide Deklarationen sind identisch, die ohne static ist vorzuziehen:

```
class Point { enum Color { RED, BLUE, GREEN } } // Üblich
```

versus

```
class Point { static enum Color { RED, BLUE, GREEN } } // Unüblich
```

### Statische Importe von Aufzählungen \*

Die Aufzählung Weekday hatten wir in das Paket com.tutego.insel.weekday gesetzt. Um auf eine Konstante wie MONDAY zugreifen zu können, wollen wir unterschiedliche import-Varianten nutzen.

Import-Deklaration	Zugriff
import com.tutego.insel.weekday.Weekday;	Weekday.MONDAY
import com.tutego.insel.weekday.*;	Weekday.MONDAY
import static com.tutego.insel.weekday.insel.Weekday.*;	MONDAY

Tabelle 6.4 Welche Zugriffe sind mit welchen import-Deklarationen möglich?

Nehmen wir im Paket als zweites Beispiel eine geschachtelte Aufzählung der Klasse Week hinzu:

**Listing 6.30** src/main/java/com/tutego/insel/weekday/Week.java

```
package com.tutego.insel.weekday;

public class Week {
    public enum Weekday {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
    }
}
```

Import-Deklaration	Zugriff
import com.tutego.insel.weekday.Week;	Week.Weekday.MONDAY
import com.tutego.insel.weekday.Week.Weekday;	Weekday.MONDAY
import static com.tutego.insel.weekday.Week.Weekday.*	MONDAY

Tabelle 6.5 Welche Zugriffe sind mit welchen import-Deklarationen möglich?

### Standardmethoden der Aufzählungstypen \*

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`,<sup>9</sup> aber nicht `Cloneable`, da Aufzählungsobjekte nicht geklont werden können. Die Methode `toString()` liefert den Namen

<sup>9</sup> Die Ordnung der Konstanten ist die Reihenfolge, in der sie geschrieben sind.

der Konstanten, sodass `Weekday.SUNDAY.toString().equals("SUNDAY")` wahr ist. Zusätzlich erbt jedes Aufzählungsobjekt von der Spezialklasse `Enum`, die in [Abschnitt 10.7, »Die Spezial-Oberklasse `Enum`«](#), näher erklärt wird.

## 6.5 Objekte anlegen und zerstören

Legt ein Programm Objekte mit dem Schlüsselwort `new` an, reserviert die Speicherverwaltung des Laufzeitystems auf dem System-Heap Speicher. Anschließend werden die Objektzustände initialisiert. Wird das Objekt nicht mehr referenziert, so räumt die *automatische Spechbereinigung (Garbage-Collector)* in bestimmten Abständen auf und gibt den Speicher an das Laufzeitystem zurück.

### 6.5.1 Konstruktoren schreiben

Legt ein Programm mit `new` ein neues Objekt an, so wird zur Initialisierung automatisch ein Konstruktor aufgerufen. Jede Klasse muss in Java einen Konstruktor haben. Entweder erzeugt der Compiler automatisch einen Konstruktor, oder wir legen einen eigenen an. Mit einem eigenen Konstruktor erreichen wir, dass ein Objekt nach seiner Erzeugung einen benutzerdefinierten Anfangszustand aufweist. Dies kann bei Klassen, die Variablen beinhalten, notwendig sein, weil sie ohne vorherige Zuweisung bzw. Initialisierung keinen Sinn ergäben.

#### Konstruktordeklarationen

Konstruktordeklarationen enthalten besondere Initialisierungen und sehen ähnlich wie Methodendeklarationen aus – so gibt es auch die bekannten Sichtbarkeiten, Parameterlisten und Überladung –, doch bestehen zwei deutliche Unterschiede:

- ▶ Konstruktoren tragen immer denselben Namen wie die Klasse.
- ▶ Konstruktordeklarationen besitzen keinen Rückgabetyp, also noch nicht einmal `void`.



#### Beispiel

Soll eine Klasse `Dungeon` einen Konstruktor bekommen, schreiben wir:

```
class Dungeon {
    Dungeon() { } // Konstruktor der Klasse Dungeon
}
```

Ein Konstruktor, der keinen Parameter besitzt, nennt sich *parameterloser Konstruktor* oder auch auf Englisch *no-arg-constructor* oder *nullary constructor*.

### Aufrufreihenfolge

Dass der Konstruktor während der Initialisierung und damit vor einem äußeren Methodenaufruf aufgerufen wird, soll ein kleines Beispiel zeigen:

**Listing 6.31** src/main/java/com/tutego/insel/oop/Dungeon.java

```
class Dungeon {
    Dungeon() {
        System.out.println( "2. Konstruktor" );
    }

    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.



Dungeon
Dungeon()
play() : void
main(String[]) : void

**Abbildung 6.14** Die Klasse »Dungeon« mit einem Konstruktor und zwei Methoden

### 6.5.2 Verwandtschaft von Methode und Konstruktor

Methoden und Konstruktoren besitzen beide Programmcode, haben eine Parameterliste, Modifizierer, können auf Objektvariablen zugreifen und this verwenden – das sind ihre Gemeinsamkeiten. Ein schon erwähnter Unterschied ist, dass Methoden einen Rückgabetyp besitzen (auch wenn er nur void ist), Konstruktoren aber nicht. Zwei weitere Unterschiede betreffen die Syntax und Semantik.

Konstruktoren tragen immer den Namen ihrer Klasse, und da Klassennamen per Konvention großgeschrieben werden, sind auch Konstruktoren immer großgeschrieben – Methoden werden in der Regel immer kleingeschrieben. Und Methoden sind in der Regel Verben, die das Objekt anweisen, etwas zu tun; Klassennamen sind Nomen und keine Verben.

Der Programmcode eines Konstruktors wird automatisch nach dem Erzeugen eines Objekts von der JVM genau einmal aufgerufen, und zwar als Erstes vor allen anderen Methoden. Methoden lassen sich beliebig oft aufrufen und unterliegen der Kontrolle des Benutzers. Konstruktoren lassen sich später nicht noch einmal auf einem schon existierenden Objekt erneut aufrufen und so ein Objekt reinitialisieren. Der Konstruktorauftrag ist implizit und automatisch mit new verbunden und kann nicht getrennt vom new gesehen werden.

Zusammenfassend können wir sagen, dass ein Konstruktor eine Art spezielle Methode zur Initialisierung eines Objektes ist.

#### JVM-Interna

Ein Java-Compiler setzt Konstruktoren als void-Methoden um, die <init> heißen.

### 6.5.3 Der Standard-Konstruktor (default constructor)

Wenn wir in unserer Klasse überhaupt keinen Konstruktor angeben, legt der Compiler automatisch einen an, da es immer einen Konstruktor geben muss. Diesen Konstruktor nennt die Java-Sprachdefinition (JLS) *default constructor*, was wir im Deutschen *Standard-Konstruktor* nennen wollen.

Schreiben wir

```
class Player { }
```

dann macht der Compiler daraus eine Version, die im Bytecode identisch ist mit:

```
class Player {
    Player() { }
}
```

Der vorgegebene Konstruktor hat immer die gleiche Sichtbarkeit wie die Klasse. Ist die Klasse paketsichtbar, ist es auch der Konstruktor. Und setzen die Modifizierer public/privat-

te/protected<sup>10</sup> die Typsichtbarkeit, wird auch der automatisch eingeführte Konstruktor public/private/protected sein.

### Standard-Konstruktor oder parameterloser 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 parameterlosen Konstruktor selbst anlegen oder ob wir uns einen vorgegebenen Konstruktor vom Compiler generieren lassen: Im Bytecode lässt sich das nicht mehr unterscheiden, und auch für den Nutzer der Klasse ist es irrelevant. Selbst eine generierte Java-doc-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...&lt;init</code>	<code>672F 4F62 6A65 6374 0100 063C 696E 6974 g/Object...&lt;init</code>
<code>3E01 0003 2829 5601 0004 436F 6465 0A00 &gt;...()V...Code..</code>	<code>3E01 0003 2829 5601 0004 436F 6465 0A00 &gt;...()V...Code..</code>
<code>0300 090C 0005 0006 0100 0F4C 696E 654E .....</code>	<code>0300 090C 0005 0006 0100 0F4C 696E 654E .....</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 0000 0000 0100 .....</code>	<code>0C00 0100 0000 0500 0C00 0000 0000 0100 .....</code>
<code>0E00 0000 0200 0F .....</code>	<code>0E00 0000 0200 0F .....</code>

Abbildung 6.15 Es gibt keinen Unterschied im Bytecode bezüglich der Konstruktoren in den Klassen C1 und C2.

Der Standard-Konstruktor ist also immer ein parameterloser Konstruktor. Auch wenn der Compiler einen Standard-Konstruktor anlegt, ist es oft sinnvoll, einen eigenen parameterlosen 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 privat sein soll, dazu gleich mehr.

### Private Konstruktoren

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 so genannten *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.

<sup>10</sup> Nur innere Typen können private oder protected sein.

#### 6.5.4 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:

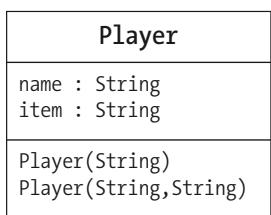
**Listing 6.32** src/main/java/com/tutego/insel/game/v6/Player.java, Player

```
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 6.16** UML-Diagramm für Player mit zwei Konstruktoren

Die Nutzung kann so aussehen:

**Listing 6.33** src/main/java/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
```

Die parametrisierten Konstruktoren verbinden sozusagen einen parameterlosen Konstruktor mit den Settern. Es spricht auch nichts dagegen, in diesen Konstruktoren an die Setter zum Setzen der Werte weiterzuleiten, denn so kann der Setter gleich validieren, und diese Aufgabe muss dann nicht noch der Konstruktor mit erledigen.

### **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 parametisierte 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 Compilerfehler, 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 Konstruktoren erzwingen, dass beim Erzeugen ein Spielername angegeben wird, 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 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, diese Attribute im Konstruktor zu setzen. Ein parameterloser 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 parameterlosen Konstruktor erzeugen, aber auch mit einem parametrisierten, der gleich die Koordinatenwerte entgegennimmt; so sind vor dem ersten Zugriff alle Werte gegeben.

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 unveränderbare (immutable) Werteklassen, die einmal im Konstruktor einen Wert bekommen und ihn beibehalten. In der Java-Bibliothek gibt es eine Reihe solcher Klassen, die keinen parameterlosen 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.

### 6.5.5 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:

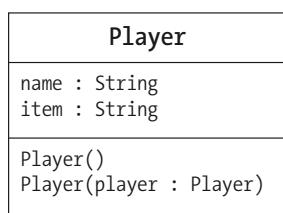
**Listing 6.34** src/main/java/com/tutego/insel/game/v7/Player.java, Player

```
public class Player {

    public String name;
    public String item;

    public Player() { }

    public Player( Player player ) {
        name = player.name;
        item = player.item;
    }
}
```



**Abbildung 6.17** UML-Klassendiagramm für Player mit 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 6.35** src/main/java/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

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 *Oracle Certified Java Programmer* auf eine solche Frage einstellen. Im Übrigen gilt bei den Methoden das gleiche Prinzip.



## 6.5.6 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 6.36** src/main/java/com/tutego/insel/game/v8/Player.java, main()

```
public class Player {
```

```
    public String name;
    public String item;
```

```
    public Player( Player player ) {
        name = player.name.trim();
        item = player.item.trim();
    }
```

```

public Player( String name, String item ) {
    this.name = name.trim();
    this.item = item.trim();
}
}

```

Zu erkennen ist, dass beide Konstruktoren die Objektvariablen initialisieren und zudem den Weißraum entfernen, also letztlich sehr ähnlich sind. 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 und Weißraumentfernung mehrfach ausprogrammiert werden. Java lässt eine solche Konstruktorverkettung mit dem Schlüsselwort `this` zu:

**Listing 6.37** src/main/java/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.trim();
        this.item = item.trim();
    }
}

```

Der Gewinn gegenüber der vorherigen Lösung ist, dass es nur eine zentrale Stelle gibt, die im Fall von Änderungen angefasst werden müsste. An `trim()` lässt sich der Vorteil schon ablesen: Vorher war das Trimmen in jedem Konstruktor eingepflegt, nach der Änderung nur lokal an einer Stelle. Nehmen wir an, wir hätten zehn Konstruktoren für alle erdenklichen Fälle in genau diesem Stil implementiert. Tritt der 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 aus-

geführt 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.

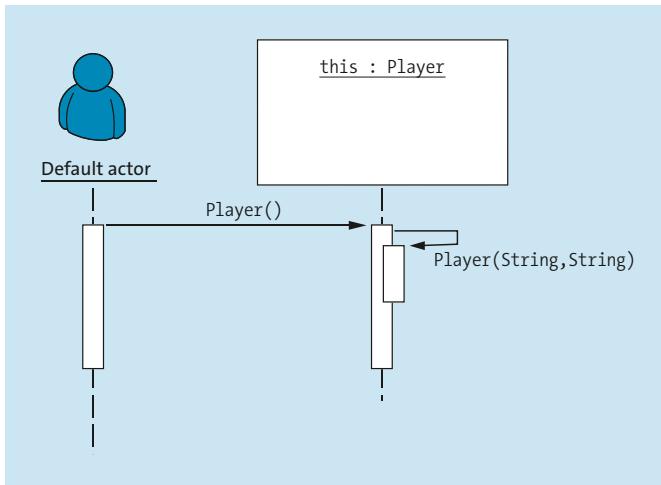


Abbildung 6.18 Das Sequenzdiagramm mit dem Aufruf von zwei Konstruktoren

### 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, wenn es mit Klammern genutzt wird. Den Klassennamen als Methodenaufruf zu verwenden, also statt `this(player.name, player.item)` etwa `Player(player.name, player.item)` zu schreiben, funktioniert syntaktisch nicht, denn es könnte tatsächlich eine großgeschriebene Methode `Player(...)` geben, die jedoch mit dem Konstruktor nichts zu tun hat.



### 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 angeprochen 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 6.38** src/main/java/com/tutego/insel/oop/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.

### 6.5.7 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-Collector<sup>11</sup> dieses Objekt und kümmert sich um alles Weitere – der Entwicklungsprozess wird dadurch natürlich vereinfacht. Der Einsatz der automatischen Speicherbereinigung verhindert zwei große Probleme:

---

<sup>11</sup> Eine lange Tradition hat die automatische Speicherbereinigung unter LISP und unter Smalltalk, aber auch Visual Basic benutzt einen GC, und selbst das C64-BASIC nutzte eine Garbage-Collection für nicht mehr benötigte Zeichenketten.

- ▶ 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++, PHP, Python, Swift kennen auch das Konzept eines Destruktors, also eines besonderen Anweisungsblocks, der immer dann aufgerufen wird, wenn das Objekt nicht mehr benötigt wird. Allgemeine Destruktoren kennt Java nicht. Es gibt jedoch mit der `finalize()`-Methode (sie wird in [Abschnitt 10.1.7, »Aufräumen mit finalize\(\)«](#), vorgestellt) eine Möglichkeit, die an einen Destruktor erinnert, allerdings mit einigen Einschränkungen – außerdem ist die Methode veraltet. Zudem ruft `try` mit Ressourcen (vorgestellt in [Abschnitt 8.6, »Automatisches Ressourcen-Management \(try mit Ressourcen\)«](#)) eine `close()`-Methode auf, sodass sich auf diese Weise wieder Ressourcen freigeben lassen.



### Prinzipielle Arbeitsweise des Müllaufsammlers

Die automatische Speicherbereinigung erscheint hier als ominöses Ding, das die Objekte clever verwaltet. Doch wie arbeitet ein Garbage-Collector? Implementiert wird er als unabhängiger Thread mit niedriger Priorität. Er verwaltet die Wurzelobjekte, von denen aus das gesamte Geflecht der lebendigen Objekte (der so genannte *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 »Kindergarten« erzeugt werden, der sehr oft nach toten Objekten durchsucht wird und in der Größe beschränkt ist. Überlebende Objekte kommen nach einiger Zeit aus dem Kindergarten in eine andere Generation, die nur selten vom GC durchsucht wird. Damit folgt der GC der Philosophie von Aufenberg, der meinte: »Verbesserungen müssen zeitig glücken; im Sturm kann man nicht mehr die Segel flicken.« Das heißt, die automatische Speicherbereinigung arbeitet ununterbrochen und räumt auf. Sie 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 die automatische Speicherbereinigung das nicht mehr benötigte Objekt hinter der Referenzvariablen `ref` entfernen, wenn die Laufzeitumgebung den inneren Block verlässt:

```
{
    {
        StringBuilder ref = new StringBuilder();
    }
    // StringBuilder 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 Array referenziert wird. Wenn dann das referenzierte Objekt nicht mehr benötigt wird, sollte die Variable (oder der Array-Eintrag) mit `null` belegt werden, da andernfalls die automatische Speicherbereinigung das Objekt aufgrund der starken Referenzierung nicht wegräumen würde. Zwar findet die automatische Speicherbereinigung jedes nicht mehr referenzierte Objekt, aber die Fähigkeit zur Wahrsagerei, Speicherlecks durch unbenutzte, aber referenzierte Objekte aufzuspüren, hat er nicht.

## 6.6 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.

### 6.6.1 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, da – vereinfacht gesagt – außerhalb von Konstruktoren und Methoden kein Code stehen darf.

Betrachten wir dies zuerst für eine Objektvariable:

**Listing 6.39** src/main/java/com/tutego/insel/oop/Joystick.java

```
class Joystick {

    int numberOfButtons = 6;

    Joystick() { }

    Joystick( int numberOfButtons ) {
        this.numberOfButtons = numberOfButtons;
    }

    Joystick( String producer ) { }
}
```

Die Variable `numberOfButtons` wird mit 6 belegt. Allerdings baut der Compiler daraus Code, der die Initialisierung in jeden Konstruktor setzt:

```
class Joystick {

    int numberOfButtons;

    Joystick() {
        numberOfButtons = 6;
    }

    Joystick( int numberOfButtons ) {
        this.numberOfButtons = 6;
        this.numberOfButtons = numberOfButtons;
    }

    Joystick( String producer ) {
        numberOfButtons = 6;
    }
}
```

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.<sup>12</sup> 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.

### 6.6.2 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.<sup>13</sup> Der Block heißt *Klasseninitialisierer* oder *statischer Initialisierungsblock*:

**Listing 6.40** src/main/java/com/tutego/insel/oop/StaticBlock.java, Ausschnitt

```
class StaticBlock {

    static {
        System.out.print( "Gut, dass " );
    }

    public static void main( String[] args ) {
        System.out.println( "zum Nachbarn hat." );
    }
}
```

---

<sup>12</sup> Wir wollen hier den Fall, dass der Konstruktor der Oberklasse `i` einen Wert ungleich 0 setzt, nicht betrachten.

<sup>13</sup> 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.print( "Neuer Boateng " );
}
}

```

Lädt der Klassenlader die Klasse StaticBlock, so führt er zuerst den ersten static-Block aus und dann den zweiten static-Block. Da die Klasse StaticBlock auch das `main(...)` besitzt, führt die virtuelle Maschine anschließend die Startmethode aus, sodass auf dem Bildschirm kommt: »Gut, dass Neuer Boateng zum Nachbarn hat.«

### 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(int)` die Bearbeitung abgebrochen wird:

**Listing 6.41** src/main/java/com/tutego/insel/oop/StaticNowMain.java, Ausschnitt

```

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`, der an Stelle einer vernünftigen Exception kommt.

### 6.6.3 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>class Beer {     static String isFreeFor = "Homer"; }</pre>	<pre>class Beer {     static String isFreeFor;     static {         isFreeFor = "Homer";     } }</pre>

Tabelle 6.6 Wie der Compiler initialisierte statische Variablen realisiert

Klasseninitialisierer sind nicht ganz ungefährlich, denn wenn der Code eine Ausnahme auslöst, dann gibt es einen harten `java.lang.ExceptionInInitializerError`. Leser können das testen, indem sie ändern:

```
static String isFreeFor = "Homer".substring( -1 );
```

und dann aus dem Hauptprogramm aufrufen:

```
System.out.println( Beer.isFreeFor );
```

#### 6.6.4 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 Aufstelle 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 anzugeöhnen, bei einer Änderung einer Konstanten 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.

### 6.6.5 Exemplarinitialisierer (Instanzinitialisierer)

Neben den Konstruktoren haben die Sprachschöpfer eine weitere Möglichkeit vorgesehen, 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 6.42** src/main/java/com/tutego/insel/oop/JavaInitializers.java, Ausschnitt

```
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 Code-duplizierung im Quellcode vermieden. Allerdings hat die Technik gegenüber einer langwierigen 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 werden schnell übersehen. Ein Blick auf den Konstruktor verrät uns dann nicht mehr, was er alles macht, da verstreute Exemplarinitialisierer 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:

sierung der Umsetzung: Der Programmcode der Exemplarinitialisierer wird an den Anfang aller Konstruktoren gesetzt. Objektvariablen wurden schon initialisiert. Ein Programmcode wie der folgende

**Listing 6.43** src/main/java/com/tutego/insel/oop/WhoIsAustin.java, Ausschnitt

```
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 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 anonyme Klassen gedacht, ein Konzept, das später in [Kapitel 9, »Geschachtelte.Typen«](#), vorgestellt wird.

## 6.6.6 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 dahin führt, dass auch sie in Konstruktoren bzw. in Initialisierungsblöcken zugewiesen werden. Im Unterschied zu nichtfinalen 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 – das kann auch später passieren. 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 dann nur noch gelesen wird:

**Listing 6.44** src/main/java/com/tutego/insel/oop/Pattern.java, Ausschnitt

```
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. Wir können auch Werte in eine Datei legen und damit die finale statische Konstantenvariable belegen. Eine Änderung der Konstanten erzwingt also keine Neuübersetzung des Java-Programms.

Im nächsten Beispiel soll eine Datei im Klassenpfad eine Konstante enthalten, die Hubble-Konstante<sup>14</sup>:

**Listing 6.45** src/main/resources/hubble-constant.txt

72

Die Klasse liest in einem static-Block den Wert aus der Datei und belegt die finale statische Konstante:

**Listing 6.46** src/main/java/com/tutego/insel/oop/LateConstant.java, Ausschnitt

```

public class LateConstant {

    public final static int HUBBLE;
    public final String ISBN;
}

```

---

<sup>14</sup> 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, weshalb heute der Begriff *Hubble-Parameter* vorgezogen wird. Weitere Details unter <https://de.wikipedia.org/wiki/Hubble-Konstante>.

```

static {
    try ( java.util.Scanner scanner = new java.util.Scanner(
        LateConstant.class.getResourceAsStream( "/hubble-constant.txt" ) ) ) {
        HUBBLE = scanner.nextInt();
    }
}

public LateConstant() {
    ISBN = "3572100100";
}

public static void main( String[] args ) {
    System.out.println( HUBBLE );                                // 77

    System.out.println( new LateConstant().ISBN );      // 3572100100
}
}

```

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 im Modulpfad 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.

## 6.7 Zum Weiterlesen

Aus einem UML-Diagramm bzw. einer Modellierung haben wir in diesem Kapitel die ersten Klassen aufgebaut. Allerdings stehen Klassen nicht allein da, sondern stehen mit anderen Klassen in Beziehung. Lies weiter, und finde heraus, wie diese Beziehungen in Java realisiert werden.

# Kapitel 7

## Objektorientierte Beziehungsfragen

»Aus einer schlechten Verbindung kann man sich schwerer lösen als aus einer guten.«

– Whitney Elizabeth Houston (1963–2012)

Objekte leben nicht in Isolation, sondern in Beziehungen zu anderen Objekten. Was wir uns in diesem Kapitel anschauen wollen, sind die Objektbeziehungen und Typbeziehungen, die Objekte und Klassen/Schnittstellen eingehen können. Im Grunde läuft das auf zwei einfache Beziehungstypen hinaus: Ein Objekt ist mit einem anderen Objekt über eine Referenz verbunden, oder eine Klasse erbt von einer anderen Klasse, sodass die Objekte Eigenschaften von der Oberklasse erben können. Insofern betrachtet das Kapitel Assoziationen für die Objektverbindungen und Vererbungsbeziehungen. Darüber hinaus geht das Kapitel auf abstrakte Klassen und Schnittstellen ein, die besondere Vererbungsbeziehungen darstellen, da sie für die Unterklassen Verhalten vorschreiben können.

### 7.1 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 Konstruktion von Objektverbänden.

#### Assoziationsarten

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 hat oder haben kann. Übliche Beziehun-

gen sind 1:1 und 1:n. Weiterhin können wir beschreiben, ob ein Teil existenzabhängig ist oder alleine existieren kann.

### 7.1.1 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 sieht das so aus:

**Listing 7.1** src/main/java/com/tutego/insel/game/va/Player.java, Player

```
class Player {  
    Room room;  
}
```

**Listing 7.2** src/main/java/com/tutego/insel/game/va/Room.java, Room

```
class Room { }
```

Zur Laufzeit müssen natürlich noch die Verweise gesetzt werden:

**Listing 7.3** src/main/java/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 7.1** Gerichtete Assoziation im UML-Diagramm

Die Multiplizität wird angegeben 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`.

### 7.1.2 Zwei Freunde müsst ihr werden – bidirektionale 1:1-Beziehungen

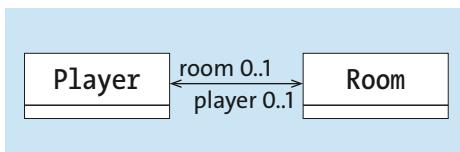
Diese gerichteten Assoziationen sind in Java sehr einfach umzusetzen, wie wir im Beispiel gesehen haben. Beidseitige Assoziationen erscheinen auf den ersten Blick auch einfach, da nur die Gegenseite um eine Verweisvariable erweitert werden muss. Beginnen wir mit dem Szenario, dass der Spieler seinen Raum und der Raum seinen Spieler kennen soll:

**Listing 7.4** src/main/java/com/tutego/insel/game/vb/Player.java, Player

```
class Player {
    Room room;
}
```

**Listing 7.5** src/main/java/com/tutego/insel/game/vb/Room.java, Room

```
class Room {
    Player player;
}
```



**Abbildung 7.2** Bei bidirektionalen Beziehungen gibt es im UML-Diagramm zwei Pfeilspitzen.

Verbinden wir das:

**Listing 7.6** src/main/java/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 7.7** src/main/java/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(...)`.

### 7.1.3 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. Im besten Fall ist die Anzahl der assoziierten Objekte fix und überschaubar, dann lassen sich mehrere Variablen verwenden.



#### Beispiel

Ein Raum hat verbundene Räume in alle vier Himmelsrichtungen:

```
class Room {
    Room north;
    Room west;
    Room east;
    Room south;
}
```

Oder natürlich kürzer – aber nicht unbedingt lesbarer – `Room north, west, east, south;.`

Soll ein Objekt mehr als eine feste Anzahl Referenzen aufnehmen, etwa dann, wenn sich in einem Raum mehrere Spieler befinden oder wenn ein Spieler eine beliebige Anzahl Gegenstände mit sich trägt, sind Datenstrukturen gefragt. Wir verwenden auf der 1-Seite einen speziellen Container, der entweder eine feste oder eine dynamische Anzahl anderer Referenzen 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 17, »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`.

### Ein Raum mit vielen Spielern

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 7.8** src/main/java/com/tutego/insel/game/vc/Player.java, Player

```
public class Player {
    public String name;

    public Player( String name ) {
        this.name = name;
    }
}
```

Vorher hat der Raum nur eine Referenzvariable vom Typ `Player` gehabt, die nur genau einen Spieler referenzieren konnte. Mit dem Einsatz von Datenstrukturen bleibt das sogar gleich: Wir referenzieren einen Container im Raum, und dieser Container verwaltet für uns die `Player`.

Um von nur einem Spieler zu einer Sammlung von Spielern zu kommen, ändern wir

Player player;

in

```
ArrayList<Player> players = new ArrayList<Player>();
```

Der Raum bekommt ein internes Attribut `players` vom Typ der `ArrayList`. 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).<sup>1</sup>

Die Details zu Generics sind Teil von [Kapitel 11](#), »Generics<T>«, doch unser Wissen ist an dieser Stelle ausreichend, um die Raum-Klasse fertigzustellen:

**Listing 7.9** src/main/java/com/tutego/insel/game/vc/Room.java, Room

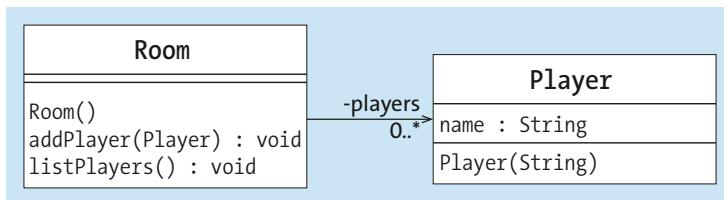
```
import java.util.ArrayList;

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 7.3** UML-Diagramm, bei dem der Room beliebig viele Player referenziert

<sup>1</sup> Die Schreibweise lässt sich auch noch ein wenig abkürzen zu `ArrayList<Player> players = new ArrayList<>();`, doch das ist jetzt nicht wichtig.

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 7.10** src/main/java/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
```

### Schnelleinstieg 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 Zählsvariable 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, Einhörner oder rostige Fähren? Es wäre sinnvoll, nicht nur die Liste selbst als Typ zu haben, sondern sozusagen rekursiv in die Liste hineinzugehen und genau hinzuschauen, was die Liste eigentlich referenziert. Genau das ist die Aufgabe von Generics. Die Datenstruktur wünscht sich 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, 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 Compilerwarnung 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 11](#), »Generics<T>«, genauer.

## 7.2 Vererbung

Schon von Kindheit an lernen wir, Objekte in Beziehung zu setzen. Assoziationen bilden dabei die Hat-Beziehung zwischen Objekten ab: Ein Teddy hat (direkt 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-Beziehung bzw. die Generalisierung<sup>2</sup>/Spezialisierung. Apfel und Birne sind Obstsorten, Lotad, Seedot und Wingull sind verschiedene Pokémon, und »Berg« ist der Sammelbegriff und die Kategorie für K2 und Mount Everest.<sup>3</sup>

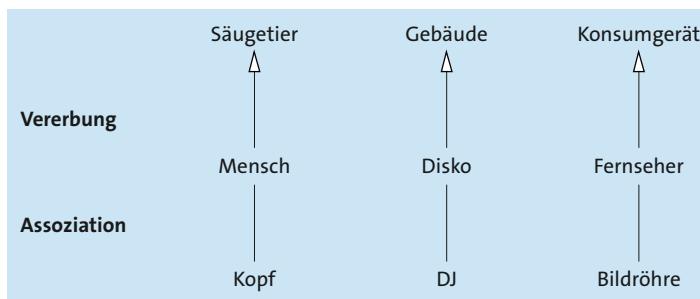


Abbildung 7.4 Vererbung und Assoziation

Das Besondere bei der Ist-eine-Art-von-Beziehung ist die Tatsache, dass die Gruppe gewisse Merkmale für alle Elemente der Gruppe vorgibt.<sup>4</sup> Bei Obst haben wir eine intuitive 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.

### 7.2.1 Vererbung in Java

Java ordnet Typen in hierarchischen Relationen an, in denen sie Ist-eine-Art-von-Beziehungen bilden. Eine neu deklarierte Klasse erweitert durch das Schlüsselwort `extends` eine ande-

2 »All generalizations are false, including this one.« (Mark Twain)

3 So etwas gibt es auch in der Linguistik; dort heißt der Oberbegriff eines Begriffs *Hyperonym* und der Unterbegriff eines Begriffs *Hyponym*.

4 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.

re 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*). Auch die Begriffe *Vorfahre* und *Nachkomme* werden von manchen Autoren verwendet.

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.



## 7.2.2 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 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 ein Attribut für einen 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, 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 7.11** src/main/java/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 7.12** src/main/java/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`. Damit erbt Player 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.

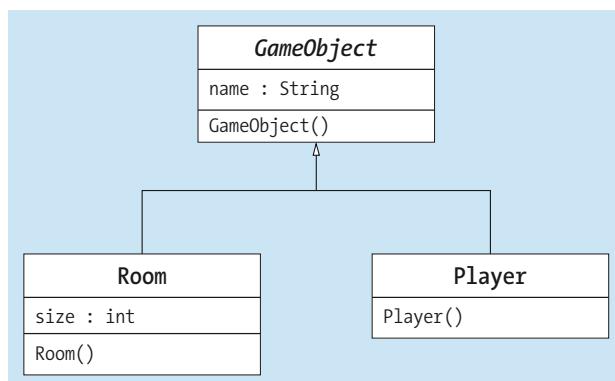
Der Raum soll neben dem geerbten Namen noch eine Größe besitzen:

**Listing 7.13** src/main/java/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 7.5** Room und Player sind zwei Unterklassen von GameObject.

Die Unterklassen Room und Player besitzen alle sichtbaren Eigenschaften der Oberklasse und zusätzlich ihre hinzugefügten:

**Listing 7.14** src/main/java/com/tutego/insel/game/vd/Playground.java, Ausschnitt

```
Room clinic = new Room();
clinic.name = "Clinic";           // Zugriff auf geerbtes Attribut
clinic.size = 120000;             // Zugriff auf eigenes Attribut

Player theDoc = new Player();
theDoc.name = "Dr. Schuwibschö"; // Zugriff auf geerbtes Attribut
```

### 7.2.3 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 mit:

```
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()`. Daher tauchen in der Tastaturvervollständigung einer IDE auch immer Methoden auf, die nicht von den eigenen Klassen stammen.

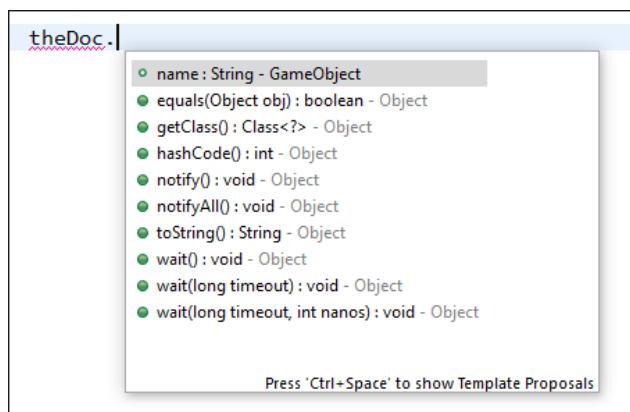


Abbildung 7.6 Methoden aus der absoluten Oberklasse java.lang.Object

### 7.2.4 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 objekt-

orientierte Programmiersprachen, wie C++<sup>5</sup>, 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 Unterklasse `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 keinen Grund zur Sorge geben. 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. Nein, nicht wirklich. Java erlaubt zwar keine multiplen Oberklassen, es erlaubt aber immer noch, mehrere Schnittstellen (Interfaces) zu implementieren und so unterschiedliche Typen anzunehmen. Mit Schnittstellen beschäftigen wir uns in [Abschnitt 7.7](#).

### 7.2.5 Sehen Kinder alles? Die Sichtbarkeit `protected`

Eine Unterklass erbt alle sichtbaren Eigenschaften. Wir kennen schon `public`, `protected` und `private`. Daraus folgt:

- ▶ Sind in der Oberklasse die Eigenschaften `public`, so sehen auch die Unterklassen die Eigenschaften, überhaupt sieht sie jeder.
- ▶ Hat die Oberklasse paketsichtbare Eigenschaften, sieht es die Unterklasse nur dann, wenn sie sich im gleichen Paket befindet, überhaupt sieht die Eigenschaften jeder Typ im gleichen Paket.
- ▶ Die Vererbung kann durch `private` eingeschränkt werden, dann sieht keine andere Klasse die Eigenschaften, weder fremde Klassen noch Unterklassen.

Bei öffentlichen, paketsichtbaren und privaten Eigenschaften haben Unterklassen also keine Sonderstellung, sie sehen nicht mehr.

---

<sup>5</sup> Bjarne Stroustrup führte Mehrfachvererbung erst in C++ 2.0 (1985–1987) ein.

### Sprachvergleich

Die Programmiersprache Eiffel hat ein interessantes Feature: Steht an einer neuen Klasse `inherit {NONE} Oberklasse`, so erbt die Klasse von der genannten Oberklasse, aber die Eigenschaften sind in der neuen Klasse privat und stehen weiteren Unterklassen dieser neuen Klasse nicht zur Verfügung.

### Schlüsselwort `protected`

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`

### Designtipp

Ist in einer Klasse eine Eigenschaft `protected`, so ist sie in allen Unterklassen sichtbar, egal, in welchem Paket sich die Unterklasse befindet. Wenn etwa A eine `protected`-Variable hat und B von A erbt und C von B, dann sieht C ebenfalls die `protected`-Variable. Das ist in der Regel ein Problem für Objektvariablen, weil schnell ein Implementierungsdetail nach außen kommt und der Code fragil wird. Es sollte folglich auf `protected`-Variablen verzichtet werden.



## 7.2.6 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 parameterlosen Konstruktor der Oberklasse auf, damit die Oberklasse »ihre« Attribute initialisieren kann. Es ist dabei egal, ob der Konstruktor in der Unterklassse parametrisiert ist oder nicht; jeder Konstruktor der Unterklassse 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 parametrisierten 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 sind die vorangehende Schreibweise und die folgende völlig gleich:

```
class GameObject extends Object {
    GameObject() {
        super();           // Ruft parameterlosen Konstruktor von Object auf
    }
}
```

```
class Player extends GameObject {
    Player() {
        super();           // Ruft parameterlosen Konstruktor von GameObject auf
    }
}
```

### Hinweis

`super(...)` muss immer die erste Anweisung im Konstruktor sein. Beim Aufbau neuer Objekte läuft die Laufzeitumgebung im Konstruktor daher als Erstes die Hierarchie nach `java.lang.Object` ab und beginnt dort von oben nach unten mit der Initialisierung. Kommt die JVM nach der Initialisierung der Oberklasse durch den Aufruf von `super()` zurück zum eigenen Konstruktor, haben alle Konstruktoren der Oberklassen ihre Zustände schon initialisiert, und wir können später im eigenen Konstruktor von vollständig initialisierten Variablen aller Basistypen ausgehen.



### `super(...)` auch bei parametrisierten Konstruktoren

Alle Konstruktoren (also die parameterlosen und parametrisierten) rufen mit `super(...)` standardmäßig den parameterlosen Konstruktor der Oberklasse auf. Nehmen wir eine Klasse für Außerirdische mit einem parametrisierten Konstruktor für den Namen des Planeten an:

**Listing 7.15** src/main/java/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 parameterlosen Konstruktor gibt, sondern nur einen parametrisierten, ruft auch dieser automatisch den parameterlosen Konstruktor der Basisklasse `GameObject` auf. Explizit ausgeschrieben heißt das:

```
public Alien( String planet ) {
    super();           // Ruft automatisch den parameterlosen Konstruktor von GameObject auf
    this.planet = planet;
}
```

Natürlich muss `super(...)` wieder als Erstes stehen.

### super(...) mit Argumenten füllen

Mitunter ist es nötig, aus der Unterkasse nicht nur den parameterlosen Konstruktor anzu-steuern, sondern einen anderen (parametrisierten) Konstruktor der Oberklasse anzuspre-chen. Dazu gibt es das `super(...)` mit Argumenten.

Der Aufruf von `super(...)` kann parametrisiert erfolgen, sodass nicht der parameterlose Kon-struktur, sondern ein parametrisierter Konstruktor aufgerufen wird. Gründe dafür könnten sein:

- ▶ Ein parametrisierter Konstruktor der Unterkasse leitet die Argumente an die Oberklasse weiter; es soll nicht der parameterlose Konstruktor aufgerufen werden, da der Oberklas-sen-Konstruktor das Attribut annehmen und verarbeiten soll.
- ▶ Wenn wir keinen parameterlosen Konstruktor in der Oberklasse vorfinden, müssen wir in der Unterkasse mittels `super(Argument)` einen speziellen, parametrisierten Konstruktor 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 Plane-tennamen erwartet:

**Listing 7.16** src/main/java/com/tutego/insel/oop/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 Standard-Konstruktor, der mit `super(...)` nach einem parameterlosen Konstruktor in `Alien` sucht – den gibt es aber nicht. Wir müssen daher entweder einen parameterlosen 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 7.17** src/main/java/com/tutego/insel/oop/Grob.java

```
public class Grob extends Alien {
    public Grob() {
        super( "Locutus" ); // Alle Grobs leben auf Locutus
    }
}
```

Es spielt dabei keine Rolle, ob `Grob` einen parameterlosen 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 das Konstruktorargument 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 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 7.18** src/main/java/com/tutego/insel/oop/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 );
    }

    private void initialize( Color color ) {
        // ...
    }
}
```

```

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(Color)`, die alle Konstruktoren aufrufen. Hier wird dann der Code platziert, den alle Konstruktoren ausführen sollen.

### 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 7.1 fasst die Unterschiede und Gemeinsamkeiten zusammen:<sup>6</sup>

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>
Bezeichnername	Gleicher Name wie die Klasse. Beginnt mit einem Großbuchstaben.	Beliebig. Beginnt mit einem Kleinbuchstaben.
<code>this</code>	<code>this</code> ist eine Referenz in Objektmethoden und Konstruktoren, die sich auf das aktuelle Exemplar bezieht.	
	<code>this(...)</code> bezieht sich auf einen anderen Konstruktor der gleichen Klasse. Wird <code>this(...)</code> benutzt, muss es in der ersten Zeile stehen.	

**Tabelle 7.1** Gegenüberstellung von Konstruktoren und Methoden

<sup>6</sup> Schon seltsam, dass `synchronized` nicht erlaubt ist, aber ein Konstruktor implizit `synchronized` ist.

Benutzung	Konstruktoren	Methoden
super	super ist eine Referenz mit dem Namensraum der Oberklasse. Damit lassen sich überschriebene Objektmethoden aufrufen.	
	super(...) ruft einen Konstruktor der Oberklasse auf. Wird es benutzt, muss es die erste Anweisung sein.	
Vererbung	Konstruktoren werden nicht vererbt.	Sichtbare Methoden werden vererbt.

Tabelle 7.1 Gegenüberstellung von Konstruktoren und Methoden (Forts.)

## 7.3 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.

### 7.3.1 Automatische und explizite Typumwandlung

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 Typumwandlung

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 Raum.
- ▶ Ein Spieler ist ein Spieler.
- ▶ 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.

Kodieren wir das in Java:

```
Listing 7.19 src/main/java/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 Typumwandlung durch. Wir werden uns dieses so genannte *liskovsche Substitutionsprinzip* im folgenden Abschnitt anschauen.



#### Hinweis

Wenn wir var nutzen, dann wird der Typ der Variablen automatisch so sein wie der auf der rechten Seite.

### Was wissen Compiler und Laufzeitumgebung über unser Programm?

Wichtig ist, zu beobachten, dass Compiler und Laufzeitumgebung unterschiedliche Dinge wissen. Durch den Einsatz von new gibt es zur Laufzeit nur zwei Arten von Objekten: Player und Room. Auch dann, wenn es

```
GameObject goIsRoom = new Room();
```

heißt, referenziert goIsRoom zur Laufzeit ein Room-Objekt. Der Compiler aber »vergisst« dies und glaubt, goIsRoom 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( goIsRoom.name );
println( goIsRoom.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 Typ, den der Compiler kennt, von dem Typ, den die JVM kennt, zu unterscheiden, nutzen wir die Begriffe *Referenztyp* und *Objekttyp*. Im Fall von `GameObject p = new Player();` ist `GameObject` der Referenztyp und `Player` der Objekttyp (Merkhilfe: Es steht ein Objekt zur Laufzeit im Speicher). Der Compiler sieht nur den Referenztyp, aber nicht den Objekttyp. Vereinfacht gesagt: Der Compiler interessiert sich bei einer Konstruktion wie `GameObject p = new Player();` nur für den linken Teil `GameObject p` und die Laufzeitumgebung nur für den rechten Teil `p = new Player()`.

### Explizite Typumwandlung

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 7.20** src/main/java/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: c.t.i.g.vd.Player cannot  
be cast to c.t.i.g.vd.Room  
at c.t.i.g.vd.ClassCastExceptionDemo.main(ClassCastExceptionDemo.java:8)
```



### Hinweis

Schreiben wir

```
GameObject go = new Room();
```

haben wir uns gezielt bei go für den Typ GameObject entschieden. Wir sollten uns bewusst sein, dass bei einer Variablendeklaration mit var die neue Variable exakt den Typ der rechten Seite bekommt.

```
var go = new Room();
```

Hier ist go ein Room, »hat« also mehr als ein GameObject. Es kann bei der Programmierung relevant sein, den Typ zu beschränken.

## 7.3.2 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 Fritt-fett 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 Unterklasse nur spezieller werden. Auch können wir weitere Unterklassen von Player und Room übergeben, da auch die Unterklasse 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 an Stelle eines Objekts auch ein Objekt der Unterklasse auftauchen kann, sprechen wir von *Substitution*. Das Prinzip wurde von der Professorin Barbara Liskov<sup>7</sup> formuliert und heißt daher auch *liskovsches Substitutionsprinzip*.

Die folgende Klasse `AskForNameOfGameObject` nutzt diese Eigenschaft. Sie fordert in der Methode `printQuestion(GameObject)` irgendein `GameObject`, von dem bekannt ist, dass es ein Attribut `name` hat, und formuliert eine Frage, woher der Name kommt. Im Hauptprogramm kann `printQuestion(GameObject)` ein Spieler oder Raum übergeben werden:

**Listing 7.21** src/main/java/com/tutego/insel/game/vd/AskForNameOfGameObject.java,  
Ausschnitt

```
public static void printQuestion( GameObject go ) {
    System.out.println( "Woher kommt " + go.name + "?" );
}

public static void main( String[] args ) {
    Player player = new Player();
    player.name = "Godman";
    printQuestion( player );           // Woher kommt Godman?

    GameObject room = new Room();
    room.name = "Hogwurz";
    printQuestion( room );           // Woher kommt Hogwurz?
}
```

---

<sup>7</sup> Die Zeitschrift »Discover« zählt sie zu den 50 wichtigsten Frauen in der Wissenschaft.

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. Das ist viel flexibler, als für die beiden Typen `Room` und `Player` eine Methode `quote(Room)` und `printQuestion(Player)` zu schreiben. Denn wenn es im Spiel später neue `GameObject`-Typen gibt, behandelt `printQuestion(GameObject)` diese ganz selbstverständlich mit.

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.

### 7.3.3 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 7.22** src/main/java/com/tutego/insel/game/vd/InstanceofDemo.java, Ausschnitt 1

```
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() == 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:

**Listing 7.23** src/main/java/com/tutego/insel/game/vd/InstanceOfDemo.java, Ausschnitt 2

```

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

```

### Keine beliebigen Typtests mit instanceof

Der Compiler lässt aber nicht alles durch. Liegen zwei Typen überhaupt nicht in der Typiehierarchie, 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 in [Abschnitt 7.5](#), »Drum prüfe, wer sich dynamisch bindet«, vorgestellt.



### instanceof und null

Ein instanceof-Test mit einer Referenz, die null ist, gibt immer false zurück:

```
String ref2 = null;
System.out.println( ref2 instanceof String );      // false
System.out.println( ref2 instanceof Object );      // false
```

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 )
```

## 7.4 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.

### 7.4.1 Methoden in Unterklassen mit neuem Verhalten ausstatten

Die Methoden sind das Angebot eines Objekts und die Schnittstelle nach außen. In erster Linie ist das ein Was, aber kein Wie. Unterklassen müssen bedingungslos das Gleiche können wie ihre Oberklasse, allerdings kann das Wie abweichen. In so einem Fall kann die Unterklasse eine Methode der Oberklasse *überschreiben*. 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.

Damit eine Methode eine andere Methode überschreibt, muss die Unterklasse eine Methode mit dem gleichen Methodennamen und der exakt gleichen Parameterliste (also der gleichen Signatur) besitzen. Der Name der Parametervariablen ist irrelevant. Ist der Rückgabe-



typ 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 in Abschnitt 7.4.4, »Kovariante Rückgabetypen«, genauer sehen.

### Hinweis

Wir sprechen nur von überschriebenen Methoden und nicht von überschriebenen Attributen, da Attribute nicht überschrieben, sondern nur *überdeckt*<sup>8</sup> werden. Attribute werden auch nicht dynamisch gebunden – eine Eigenschaft, die später in Abschnitt 7.5.5, »Eine letzte Spielerei mit Javas dynamischer Bindung und überdeckten 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 7.24** `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 7.25** `src/main/java/com/tutego/insel/game/ve/GameObject.java, GameObject`

```
public class GameObject {
    public String name;
}
```

Auf einem `GameObject`-Objekt liefert `toString()` eine etwas kryptische Kennung:

```
GameObject go = new GameObject();
System.out.println( go.toString() ); // com.tutego.insel.game.ve.GameObject@e48e1b
```

Es ist also eine gute Idee, `toString()` in den Unterklassen zu überschreiben. Eine String-Kennung 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:

---

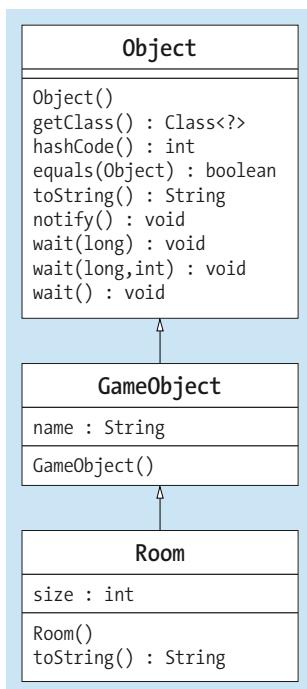
<sup>8</sup> Die JLS unterscheidet genau genommen »shadowing« und »hiding«. Interessierte Leser mögen das unter <https://docs.oracle.com/javase/specs/jls/se11/html/jls-6.html#jls-6.4> nachlesen.

**Listing 7.26** src/main/java/com/tutego/insel/game/ve/Room.java, Room

```
public class Room extends GameObject {

    public int size;

    @Override public String toString() {
        return String.format( "%s[name=%s, size=%d]",
            getClass().getSimpleName(), name, size );
    }
}
```

**Abbildung 7.7** Room ist eine Unterklasse von GameObject und hat ein eigenes `toString()`.

Und der Test sieht so aus:

**Listing 7.27** src/main/java/com/tutego/insel/game/ve/Playground.java, main()

```
Room winterfield = new Room();
winterfield.name = "Winterfield";
winterfield.size = 2040000;
System.out.println( winterfield ); // Room[name=Winterfield, size=2040000]
```

Zur Erinnerung: Ein `println(Object)` auf einem beliebigen Objekt ruft die `toString()`-Methode von diesem Objekt auf.

### Exkurs Annotation

Wir haben schon oft mit unterschiedlichen Modifizierern gearbeitet, etwa `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 auf der anderen Seite auch zum Beispiel ein besonderes Metadatum an einem Element hängen, um auszudrücken, dass es nur bestimmte Wertebereiche annehmen kann.

Java bietet eine 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 {     @WebMethod int add( int x, int y ) ...</code>	Definiert einen Webservice 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 7.2 Beispiele für Annotationen und Anwendungsfälle

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. Und wenn Annotationen an Typen gesetzt werden, bekommen sie in der Regel eine eigene Zeile.

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.

### Die Annotation @Override

Unsere Beispielklasse Room nutzt die Annotation @Override an der Methode `toString()` und macht auf diese Weise deutlich, dass die Klasse eine Methode des Obertyps 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 dies zwei Vorteile:

Zwar weiß die Laufzeitumgebung, dass eine Methode überschrieben wird, allerdings sollte Code dem Leser auch alle Informationen darüber geben, was passiert. Wird eine Methode überschrieben, ist das etwas Bedeutsames, das im Code dokumentiert werden sollte.

Außerdem überprüft der Compiler, ob wir tatsächlich eine Methode aus der Oberklasse überschreiben. Haben wir uns zum Beispiel im Methodennamen verschrieben und somit der Unterklasse unbeabsichtigt eine neue Methode hinzugefügt, so würde das der Compiler aufgrund seiner Kenntnis von @Override als Fehler melden. Einfache Schreibfehler 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.

### 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(Date)` aus Super überschreibt, obwohl es auf den ersten Blick so aussieht:

Deklaration der Basisklasse	Überladene, keine überschriebene 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(Date)` aus Sub überlädt `printDate(Date)` 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 Unterkasse 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.

### Finale Parameter in der Vererbung \*

Wird eine Methode überschrieben, dann sind die Typen der Parameterliste bestimmend, nicht die Namen. Auch spielt es keine Rolle, ob die Parametervariablen `final` sind oder nicht. Wir können es als zusätzliche Information für die jeweilige Methode betrachten. Eine Unterkasse kann demnach beliebig das `final` hinzufügen oder auch wegnehmen. Alte Bibliotheken lassen sich so leicht weiterverwenden.

#### 7.4.2 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 Unterkasse eine völlig neue Implementierung und bezieht sich dabei nicht auf die Logik der Oberkasse.

Möchte eine Unterkasse sagen: »Was meine Eltern können, ist doch gar nicht so schlecht«, kann mit der speziellen Referenz `super` auf die Eigenschaften im Namensraum der Oberkasse 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 Elternkasse ist weiterhin verfügbar.

In unserem Spiel hatte `GameObject` kein `toString()`, ändern wir dies:

**Listing 7.28** src/main/java/com/tutego/insel/game/vf/GameObject.java, `GameObject`

```
public class GameObject {

    public String name;

    @Override public String toString() {
        return String.format( "%s[name=%s]", getClass().getSimpleName(), name );
    }
}
```

Die Unterklasse Room erweitert GameObject und sollte `toString()` neu realisieren, da ein Raum ein zusätzliches Attribut hat, nämlich die Größe. Wird `toString()` allerdings in Room überschrieben, muss sich `toString()` auch um die geerbten Eigenschaften kümmern, sprich: den Namen. Das ist ungünstig, denn kommt zum Beispiel in der Oberklasse GameObject ein Attribut hinzu, müssen alle `toString()`-Methoden von allen Unterklassen geändert werden, wenn sie alle Attributbelegungen mit in die String-Kennung einbinden möchte.

Eine Lösung für das Problem ist, in `toString()` einer Unterklasse wie Room einfach auf die `toString()`-Methode der Oberklasse GameObject zuzugreifen und dann das zusätzliche Attribut mit aufzunehmen.

**Listing 7.29** src/main/java/com/tutego/insel/game/vf/Room.java, Room

```
public class Room extends GameObject {

    public int size;

    @Override public String toString() {
        return super.toString() + "[size=" + size+ "]";
    }
}
```

Stünde statt `super.toString()` nur `toString()` im Rumpf, würde der Methodenaufruf in die Endlosrekursion führen, daher funktioniert es ohne super-Referenz nicht.

Ein Test zeigt das Resultat:

**Listing 7.30** src/main/java/com/tutego/insel/game/vf/RoomToString.java, main()

```
Room enterprise = new Room();
enterprise.name = "Enterprise";
enterprise.size = 725;
System.out.println( enterprise ); // Room[name=Enterprise][size=725]
```

### 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 7.31** src/main/java/com/tutego/insel/oop/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 überdecktes 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.

### 7.4.3 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, StringBuilder`
- ▶ `Integer, Double ... (Wrapper-Klassen)`
- ▶ `Math`
- ▶ `System, Locale`
- ▶ `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(String) 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 mit einer überschriebenen setName(String)-Methode die protected-Variable name beschreiben, auf die jede UnterkLASSE Zugriff hat:

**Listing 7.32** src/main/java/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 7.33** src/main/java/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;
    }
}
```

Wir belassen es bei dem Beispiel bei einer protected-Variablen, denn Unterklassen möchten vielleicht die Variable verändern, aber eben nicht über `setName(String)`. Die Unterklassen können zum Beispiel die Variable `name` auf den Leer-String `" "` zurücksetzen, was ein Aufruf über `setName(String)` nicht vermag.

#### Hinweis

Auch private Methoden können `final` sein, aber private Methoden lassen sich ohnehin nicht überschreiben (sie werden überdeckt), sodass `final` überflüssig ist.



#### 7.4.4 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 Rückgabetyp besitzen. Das nennt sich *kovarianter Rückgabetyp* und ist sehr praktisch, da sich auf diese Weise Entwickler oft explizite Typumwandlung 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 7.34** src/main/java/com/tutego/insel/oop/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 Rückgabetyp nicht `Loudspeaker`, sondern `BigBassLoudspeaker` heißt.

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 über schriebenen Methode der Basisklasse.

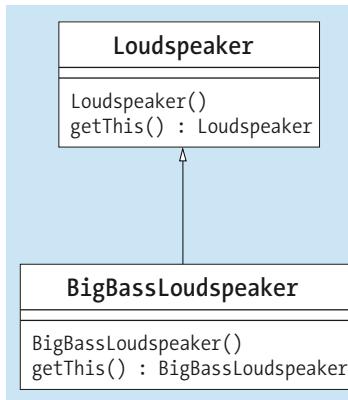


Abbildung 7.8 BigBassLoudspeaker ist ein spezieller Loudspeaker.



### Hinweis

Merkwürdig ist in diesem Zusammenhang, dass es in Java schon immer veränderte Zugriffsrechte gegeben hat. Eine Unterklasse kann die Sichtbarkeit erweitern. Auch bei Ausnahmen kann eine Unterklasse speziellere Ausnahmen bzw. ganz andere Ausnahmen als die Methode der Oberklasse erzeugen.

## 7.4.5 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 Array setzt:

**Listing 7.35** src/main/java/com/tutego/insel/oop/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 passt:

**Listing 7.36** src/main/java/com/tutego/insel/oop/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-Array speichern, da ein Object ja ein Basistyp ist. Vor einem Dilemma stehen wir dann, wenn das Array 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 com.tutego.insel.oop.ArrayCovariance.set(ArrayCovariance.java:5)
at com.tutego.insel.oop.ArrayCovariance.main(ArrayCovariance.java:17)
```

Das haben wir aber auch verdient, denn ein StringBuilder-Objekt lässt sich nicht in einem String-Array 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.

## 7.5 Drum prüfe, wer sich 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 vorgibt und eine Unterklasse diese überschreiben kann. Wir hatten das bisher bei `toString()` gesehen. 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.

### 7.5.1 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 schon vorgestellt implementiert sein:

**Listing 7.37** src/main/java/com/tutego/insel/game/vf/GameObject.java, `GameObject`

```
public class GameObject {

    public String name;

    @Override public String toString() {
        return String.format( "%s[name=%s]", getClass().getSimpleName(), name );
    }
}
```

**Listing 7.38** src/main/java/com/tutego/insel/game/vf/Room.java, `Room`

```
public class Room extends GameObject {

    public int size;

    @Override public String toString() {
        return super.toString() + "[size=" + 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 Stringrepräsentation.

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 7.39** src/main/java/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()`?

Antwort:

```

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.

Entscheidend ist, dass die Laufzeitumgebung den Objekttyp nutzt und nicht den Referenztyp – das ist das gleiche Verhalten wie bei `instanceof`. Da dem im Programmtext vereinbarten Variablentyp 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 jeweils 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 `to-`

`String()` nicht überschreiben, so fände die Laufzeitumgebung `toString()` in `Object`, aber die Methode gäbe es auf jeden Fall – entweder die Originalmethode 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 *Generics*), und die Theoretiker kennen noch viel mehr beängstigende Begriffe.

## 7.5.2 Implementierung von `System.out.println(Object)`

Werfen wir einen Blick auf ein Programm, das dynamisches Binden noch deutlicher macht. Die `printXXX(...)`-Methoden sind so überladen, dass sie jedes beliebige Objekt annehmen und dann die String-Repräsentation ausgeben:

**Listing 7.40** `java/io/PrintStream.java`, Skizze von `println()`

```
public void println( Object x ) {
    String s = String.valueOf( x );
    // String s = (obj == null) ? "null" : obj.toString();
    synchronized ( this ) {
        print( s );
        newLine();
    }
}
```

Die `println(Object)`-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(String)` 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 Typ hierarchie an, standardmäßig die Oberklassen und bekannten Unterklassen.



### 7.5.3 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 7.41** src/main/java/com/tutego/insel/oop/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 Unterkasse keinen Fehler. Für den Compiler ist es in Ordnung, wenn es eine Methode in der Unterkasse 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 Unterkasse soll von den privaten Methoden in der Oberklasse gar nichts wissen. Statt von *Überschreiben* sprechen wir hier von *Überdecken* oder *Verdecken*.

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 `Banana` 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`, `statische` und `finale` 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 überschrieben. Andernfalls könnten private Implementierungen im Nachhinein geändert werden, und Oberklassen wären nicht mehr sicher, dass nur 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 `Fruit` ist somit unnütz.

#### 7.5.4 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 Kon-

struktor. 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 Unterklassie initialisiert wurden. Wenn es die Oberklasse nun schafft, auf die Variablen der Unterklassie zuzugreifen, 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 Unterklassie zugreifen kann. Wir können aber in der Oberklasse genau jene Methode der Unterklassie aufrufen, die die Unterklassie aus der Oberklasse überschreibt. Da Methodenaufrufe dynamisch gebunden werden, kann eine Methode den Wert auslesen:

**Listing 7.42** src/main/java/com/tutego/insel/oop/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.

### 7.5.5 Eine letzte Spielerei mit Javas dynamischer Bindung und überdeckten Attributten \*

Der Kanadier »Furious Pete«<sup>9</sup> isst alles in Rekordzeit; verewigen wir seine Pizzavertilgungsleistungen in einem Java-Programm. Die Oberklasse `PizzaEater` repräsentiert die Durchschnittssesser, die für eine 12"-Pizza geschätzte 900 Sekunden benötigen. `FuriousPete` ist eine Spezialisierung und schafft es in 32 Sekunden:

---

<sup>9</sup> <http://guinnessworldrecords.com/news/2016/4/competitive-eater-challenged-to-fastest-time-to-eat-a-12%20%99%20%99-pizza-record-guinness-426366>

**Listing 7.43** src/main/java/com/tutego/insel/oop/FuriousPete.java

```

class PizzaEater {

    int consumptionTime = 900 /* Seconds */;

    void eat() {
        System.out.printf( "Ich esse in %d Sekunden eine Pizza%n", consumptionTime );
    }
}

public class FuriousPete extends PizzaEater {

    int consumptionTime = 32 /* Seconds */;

    @Override void eat() {
        System.out.println( consumptionTime );           // 32
        System.out.println( super.consumptionTime );     // 900
        System.out.println( this.consumptionTime );      // 32
        System.out.println( ((PizzaEater) this).consumptionTime ); // 900
    }

    public static void main( String[] args ) {
        new FuriousPete().eat();
    }
}

```

Die Oberklasse `PizzaEater` deklariert eine Objektvariable `consumptionTime`, und auch die Unterklasse deklariert ein Attribut mit dem gleichen Namen; es ist in Java zulässig, dass ein Attribut ein anderes gleich benanntes Attribut überdeckt.

Die Unterklasse kann mit `super.consumptionTime` eine Ebene höher kommen. `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 direkt eine Objekteigenschaft stehen, und Anweisungen wie `super.super.consumptionTime` sind somit immer ungültig.

Bei Methodenaufrufen bindet das Laufzeitsystem immer dynamisch, bei Attributzugriffen ist das nicht so; hier bestimmt der Compiler, von welcher Klasse das Attribut genommen werden soll. Unser Programm zeigt das an der Anweisung:

```
System.out.println( ((PizzaEater) this).consumptionTime ); // 3000
```

Die Ausgabe 3000 ist identisch mit `System.out.println(super.consumptionTime)`. Die `this`-Referenz hat in dem Kontext den Typ `FuriousPete`. Wenn wir den Typ aber in den Basistyp `PizzaEater` konvertieren, bekommen wir genau die Belegung von `consumptionTime` aus der Basisklasse unserer Hierarchie. Eine explizite Typumwandlung in Richtung eines Obertyps ist bei dynamisch gebundenen Methodenaufrufen auch nie nötig; die Laufzeitumgebung entscheidet selbstständig, wohin der Aufruf geht. Setzen wir in die `eat()`-Methoden vom `FuriousPete` die Zeile

```
((PizzaEater) this).eat();
```

ist das identisch mit

```
eat();
```

also eine Rekursion.

## 7.6 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*). Während `final` im Prinzip die Klasse abschließt und Unterklassen unmöglich macht, sind abstrakte Klassen das Gegenteil: Ohne Unterklassen sind abstrakte Klassen nutzlos.

Es ergeben sich daher drei Szenarien:

Klassentyp	Bedeutung
normale nichtabstrakte und nichtfinale Klasse	Eine Unterklasse kann gebildet werden, muss aber nicht.
finale Klasse	Eine Unterklasse kann nicht gebildet werden.
abstrakte Klasse	Eine Unterklasse muss gebildet werden.

Tabelle 7.3 Klassentypen im Vergleich

### 7.6.1 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.

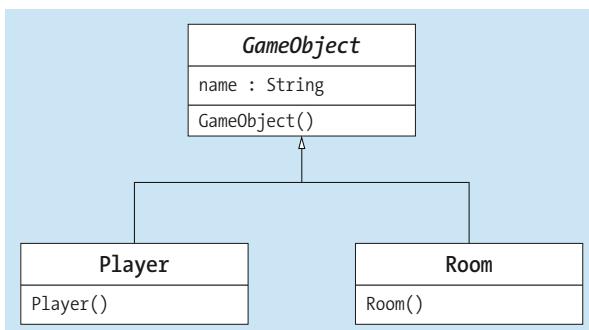
Um dies in Java auszudrücken, setzen wir den Modifizierer `abstract` an die Typdeklaration der Oberklasse. Von dieser Klasse können dann keine Exemplare gebildet werden, und der Versuch einer Objekterzeugung führt zu einem Compilerfehler. Ansonsten verhalten sich die abstrakten Klassen wie normale, enthalten die gleichen Eigenschaften und können auch selbst von anderen Klassen erben. Abstrakte Klassen sind das Gegenteil von *konkreten Klassen*.

Wir wollen die Klasse `GameObject` als Oberklasse für die Spielgegenstände abstrakt machen, da Exemplare davon nicht existieren müssen:

**Listing 7.44** src/main/java/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 und Räume. 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 7.9** 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 wird. 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 7.45** src/main/java/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 Arrays. Das ist unabhängig davon, ob die Klasse `GameObject` abstrakt ist oder nicht.

### 7.6.2 Abstrakte Methoden

Der Modifizierer `abstract` vor dem Schlüsselwort `class` leitet die Deklaration einer abstrakten Klasse ein. Doch auch eine Methode kann abstrakt sein. Sie gibt lediglich die Signatur vor, und eine Unterklasse implementiert irgendwann diese Methode. Die abstrakte Klasse ist somit für den Kopf der Methode zuständig, während die Implementierung an anderer Stelle erfolgt. Das ist eine klare Trennung vom »Was kann ich« und »Wie mache ich es«. Während die Oberklasse mit der Deklaration der abstrakten Methode ausdrückt, dass sie etwas *kann*, realisieren die Unterklassen, *wie* der Code dazu aussieht. Überspitzt gesagt: Abstrakte Methoden drücken aus, dass sie keine Ahnung von der Implementierung haben und dass sich die Unterklassen darum kümmern müssen. Das Ganze muss natürlich im Rahmen der Spezifikation geschehen.

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.

### 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*, *Tür*, *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 )
```

Ein Beispiel für die Operation `useOn(GameObject)` ist `key.useOn(door)`. Das eigene Objekt (`key` in dem Fall) wendet sich auf das an die Methode übergebene Objekt (`door` in dem Beispiel) an. Implementiert die Schlüssel-Klasse `useOn(GameObject)`, so kann sie testen, ob das übergebene `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(GameObject)` 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 7.46** src/main/java/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. Da abstrakte Methoden immer ohne Implementierung sind, 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 nichtabstrakte 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.

Kommen wir zurück zum Beispiel: Die Unterklasse `Door` soll die abstrakte Methode `useOn` (`GameObject`) 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 7.47** src/main/java/com/tutego/insel/game/vi/Door.java, Door

```
public class Door extends GameObject {

    int      id;
    boolean isOpen;

    public Door( int id ) { this.id = id; }

    @Override public boolean useOn( GameObject object ) {
        return false;
    }
}
```

Eine Tür hat für den Schlüssel eine ID, denn nicht jeder Schlüssel passt auf jedes Schloss. Außerdem hat die Tür einen Zustand: Sie kann offen oder geschlossen sein.

Die dritte Klasse, Key, speichert ebenfalls eine ID und überschreibt useOn(GameObject):

**Listing 7.48** src/main/java/com/tutego/insel/game/vi/Key.java, Key

```
public class Key extends GameObject {

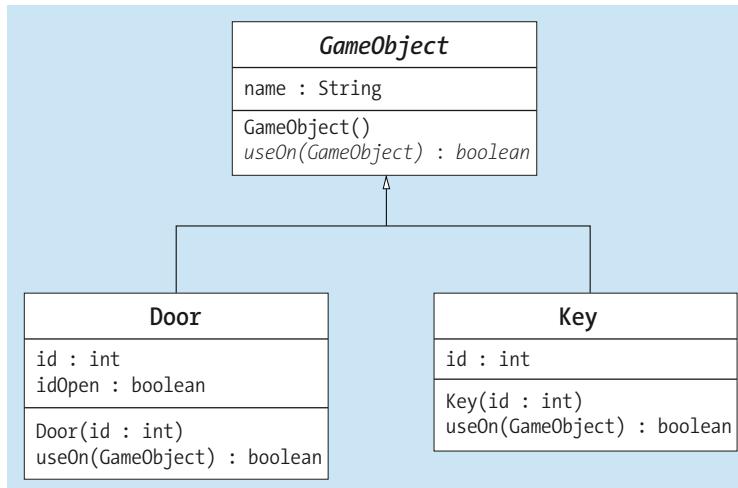
    int id;

    public Key( int id ) { this.id = id; }

    @Override public 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 übereinstimmen. Ist auch dieser Vergleich wahr, kann isOpen wahr werden, und die Methode liefert true.



**Abbildung 7.10** UML-Diagramm der Tür, des Schlüssels und der Oberklasse

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 7.49** src/main/java/com/tutego/insel/game/vi/Playground.java, main()

```
Door door1 = new Door( 1 );
GameObject key1 = new Key( 1 );
GameObject key9 = new Key( 9 );

System.out.printf( "erfolgreich=%b, isOpen=%b%n", key9.useOn(door1), door1.isOpen );
System.out.printf( "erfolgreich=%b, isOpen=%b%n", key1.useOn(door1), door1.isOpen );
System.out.printf( "erfolgreich=%b%n", door1.useOn(key1) );
```

Die Ausgaben sind:

```
erfolgreich=false, isOpen=false
erfolgreich=true, isOpen=true
erfolgreich=false
```



### Hinweis

Wenn Methoden einmal mit Rumpf existieren, so können sie nicht später abstrakt überschrieben werden und somit noch tieferen Unterklassen vorschreiben, sie zu überschreiben. Wenn eine Implementierung einmal vorhanden ist, kann sie nicht wieder versteckt werden. So existiert z. B. `toString()` in `Object` und könnte nicht in `GameObject` abstrakt überschrieben werden, um etwa für Unterklassen `Room` oder `Player` vorzuschreiben, `toString()` überschreiben zu müssen.



Implementiert eine Klasse nicht alle geerbten abstrakten Methoden, so muss die Klasse selbst wieder abstrakt sein. Ist unsere Unterklasse 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.

Das Schöne an abstrakten Methoden ist, dass sie auf jeden Fall von konkreten Exemplaren realisiert werden. Hier finden also immer polymorphe Methodenaufrufe statt.

### Zu Ende gespielt

Wir wollen das Spiel zu Ende bringen und Folgendes implementieren:

- ▶ Von der Konsole soll eine Zeile gelesen werden der Art Verb–Subjekt–Objekt, wie »stecke Schlüssel in Tür«.
- ▶ Da wir nur »nutze« kennen, wird der Verbanteil im Satz ignoriert, also aus der Eingabe gelöscht.
- ▶ Subjekt und Objekt kommen als Strings in der Eingabe vor. Wir müssen also den Namen eines Objekts auf ein `GameObject`-Exemplar übertragen. Das machen wir mit einer neuen Datenstruktur `HashMap`, einem Assoziativspeicher.

```

Listing 7.50 src/main/java/com/tutego/insel/game/vi/Game.java, main()
HashMap<String, GameObject> gameObjects = new HashMap<>();

// Spezielle Syntax, um eine Unterklasse von GameObject zu schreiben
// und dann die Methode zu überschreiben
GameObject nullGameObject = new GameObject() {
    @Override public boolean useOn( GameObject object ) { return false; }
};

gameObjects.put( "höllentor", new Door( 1 ) );
gameObjects.put( "höllenschlüssel", new Key( 1 ) );
gameObjects.put( "himmelsschlüssel", new Key( 9 ) );

while ( true ) {
    System.out.printf( "Was möchtest du tun?%n> " );
    String input = new Scanner( System.in ).nextLine().toLowerCase();
    if ( input.matches( "ende|bye|schluss|quit" ) )
        System.exit( 0 );
    String simplifiedLine =
        input.replaceAll( "benutze|stecke|nutze|mit|bei|auf|unter|in", "" );
    StringTokenizer tokenizer = new StringTokenizer( simplifiedLine );
    if ( tokenizer.countTokens() < 2 ) {
        System.out.println( "Details bitte, '" + input + "' reicht mir nicht!" );
        continue;
    }
    GameObject subject = gameObjects.getOrDefault( tokenizer.nextToken(),
                                                nullGameObject );
    GameObject object = gameObjects.getOrDefault( tokenizer.nextToken(),
                                                nullGameObject );
    System.out.println( subject.useOn( object ) ? "Ausgeführt" :
                        "Konnte '" + input + "' nicht ausführen" );
}

```

Zum Zerlegen der Eingabe greifen wir auf StringTokenizer zurück. Ein Trick in dem Programm ist das besondere nullGameObject, die Implementierung des Null-Object-Patterns. Die Idee ist, ein Objekt zu haben, das einfach bei jeder Anfrage nichts macht. Wir nutzen das, um einer NullPointerException bei get(...) aus dem Wege zu gehen, denn ist kein GameObject mit einem Namen assoziiert, liefert die Map-Methode null, und eine Kombination von null.useOn(...) ist tödlich. getOrDefault(...) liefert beim Nichtfinden das nullGameObject, und darauf können wir gültig alles anwenden, nur es passiert dann nichts. Alternativ hätten wir

abfragen können, ob null das Ergebnis von get(...) ist, und dann eine Meldung der Art »Objekt mit dem Namen ist nicht bekannt« ausgeben können.

## 7.7 Schnittstellen

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.

### 7.7.1 Schnittstellen sind neue Typen

Da Java nur Einfachvererbung kennt, ist es schwierig, Klassen mehrere Typen zu geben. Da es aber möglich sein soll, dass in der objektorientierten Modellierung eine Klasse mehrere Typen annimmt, gibt es das Konzept der *Schnittstelle* (engl. *interface*). Eine Klasse kann dann von einer Klasse erben und eine beliebige Anzahl Schnittstellen implementieren und auf diese Weise weitere Typen annehmen.

Eine Schnittstelle ist wie eine Klasse ein Typ und hat viele Gemeinsamkeiten, nur die Intention ist eine andere. Eine Schnittstelle kann enthalten:

- ▶ abstrakte Methoden
- ▶ private und öffentliche konkrete Methoden (so genannte *Default-Methoden*)
- ▶ private und öffentliche statische Methoden
- ▶ Konstanten
- ▶ geschachtelte Typen, wie Aufzählungen.

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. Auch kann sie keine Objektvariablen deklarieren.

Werden wir konkret. Vererbung ist immer linear, 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 zusätzlich den Typ Buyable annehmen soll. Denn soll eine Klasse auf einer Ebene von mehreren Typen erben, geht das durch die Einfachvererbung nicht.

### 7.7.2 Schnittstellen deklarieren

Die Deklaration einer Schnittstelle erinnert an eine abstrakte Klasse, nur steht an Stelle von `class` das Schlüsselwort `interface`:

```
interface Buyable {  
}
```

Die Schnittstelle kann nun von Klassen implementiert werden.

### 7.7.3 Abstakte Methoden in Schnittstellen

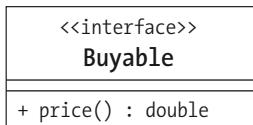
Die wichtigsten Elemente in Schnittstellen sind abstrakte Methoden. Wir kennen das schon von abstrakten Klassen: Eine abstrakte Methode hat keine Implementierung, sondern deklariert nur den Kopf einer Methode – also Modifizierer, den Rückgabetyp und die Signatur – ohne Rumpf. Deklariert wird also nur eine Vorschrift – die Implementierung einer Objektmethode übernimmt später eine Klasse.<sup>10</sup>

Sollen in einem Spiel gewisse Dinge käuflich sein, haben sie einen Preis. Eine Schnittstelle `Buyable` soll allen Klassen die Methode `price()` vorschreiben:

**Listing 7.51** src/main/java/com/tutego/insel/game/vk/Buyable.java, `Buyable`

```
interface Buyable {  
    double price();  
}
```

Da Objektmethoden in Schnittstellen standardmäßig abstrakt und öffentlich sind, können die Modifizierer `abstract` und `public` entfallen und sind redundant. Die von den Schnittstellen deklarierten Operationen sind – wie auch bei abstrakten Methoden – mit einem Semikolon abgeschlossen. Eine Implementierung ist möglich, wie wir später sehen werden.



**Abbildung 7.11** UML-Diagramm der Schnittstelle `Buyable`

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.



<sup>10</sup> Oder ein Lambda-Ausdruck, doch dazu später mehr in Kapitel 12, »Lambda-Ausdrücke und funktionale Programmierung«



### 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.

#### 7.7.4 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 7.52** src/main/java/com/tutego/insel/game/vk/Chocolate.java, Chocolate

```
public class Chocolate implements Buyable {
    @Override public double price() {
        return 0.69;
    }
}
```

Die Annotation `@Override` zeigt wieder eine überschriebene Methode (hier implementierte Methode einer Schnittstelle) an.

Während `Chocolate` nur die Schnittstelle `Buyable` implementiert, soll `Magazine` zusätzlich ein `GameObject` sein:

**Listing 7.53** src/main/java/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.

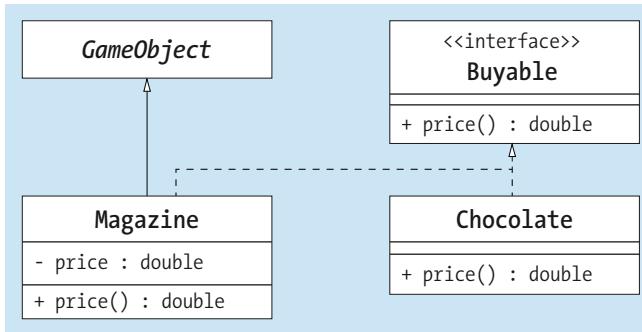


Abbildung 7.12 Die Klassen Magazine und Chocolate implementieren Buyable.

Es gelten dann folgende Typbeziehungen (die sich auch mit `instanceof` testen lassen):

- ▶ `GameObject` ist ein `GameObject`.
- ▶ `GameObject` ist ein `Object`.
- ▶ `Magazine` ist ein `Magazine`.
- ▶ `Magazine` ist ein `GameObject`.
- ▶ `Magazine` ist ein `Object`.
- ▶ `Magazine` ist ein `Buyable`.
- ▶ `Chocolate` ist ein `Chocolate`.
- ▶ `Chocolate` ist ein `Buyable`.
- ▶ `Chocolate` ist ein `Object`.

Fordert eine Methode ein Objekt eines gewissen Typs, haben wir viele Möglichkeiten:

Methode fordert Typ	Ein gültiger Argumenttyp ist
<code>Object</code>	<code>Object</code> (also beliebig), <code>Magazine</code> , <code>Chocolate</code> , <code>GameObject</code> , <code>Buyable</code>
<code>GameObject</code>	<code>GameObject</code> , <code>Magazine</code>
<code>Buyable</code>	<code>Buyable</code> , <code>Magazine</code> , <code>Chocolate</code>
<code>Magazine</code>	<code>Magazine</code>
<code>Chocolate</code>	<code>Chocolate</code>

Tabelle 7.4 Was Methoden bekommen können, wenn sie gewisse Typen fordern

Wir lesen ab: Wenn ein konkreter Typ wie `Magazine` oder `Chocolate` gefordert ist, haben wir wenig Optionen. Bei Basistypen gibt es üblicherweise immer mehrere Varianten – wer wenig will, kann eben viel bekommen.



### Hinweis

Sind die in Schnittstellen deklarierten Operationen `public`, müssen auch die implementierten Methoden in den Klassen immer öffentlich sein. `protected` ist als Sichtbarkeit nicht erlaubt. Und private Schnittstellenmethoden sind in implementierenden Klassen sowie nicht sichtbar.

Implementiert eine Klasse nicht alle Operationen aus den Schnittstellen, so erbt sie damit abstrakte Methoden und muss selbst wieder als abstrakt gekennzeichnet werden.



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, dreht sich die Ansicht um, und Obertypen stehen unten; implementierte Schnittstellen tauchen mit unter den Obertypen auf.

### 7.7.5 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. Über Schnittstellen lassen sich ganz unterschiedliche Sichten auf ein Objekt beschreiben. 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 mächtiges Objekt zum Beispiel als Argument einer Methode verwendet wird, obwohl je nach Kontext der Parametertyp einer Methode nur die kleine Schnittstelle ist.

Mit Magazine und Chocolate haben wir zwei Klassen, die Buyable implementieren. Damit existieren zwei Klassen, die einen gemeinsamen Typ und 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 7.54** src/main/java/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 7.55** src/main/java/com/tutego/insel/game/vk/PriceUtils.java, calculateSum()

```
static double calculateSum( Buyable first, Buyable... more ) {
    double result = first.price();

    for ( Buyable buyable : more )
        result += buyable.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, first.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.

## 7.7.6 Die Mehrfachvererbung bei Schnittstellen

Eine Klasse kann höchstens eine Basisklasse haben – 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 unterschiedliche Typen. Ist Unter eine solche Klasse mit der Oberklasse Ober und implementiert sie die Schnittstellen I1 und I2, so liefert für ein Exemplar u vom Typ Unter der Test u instanceof Ober ein wahres Ergebnis genauso wie u instanceof I1 und u instanceof I2.

### Begrifflichkeit

Wenn es um das Thema Mehrfachvererbung geht, dann müssen wir Folgendes unterscheiden: Geht es um *Klassenvererbung*, so genannte *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. *Typvererbung* 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 alle Unterklassen von GameObject ebenfalls vom Typ Serializable sind. Die Markierungsschnittstelle schreibt nichts vor, daher gibt es keine spezielle übergeschriebene Methode:

**Listing 7.56** src/main/java/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 definieren, dass das günsti-

gere Magazin vor einem teureren steht (eigentlich sollten mit Comparable auch equals(...) und hashCode() aus Object überschrieben werden, doch das spart das Beispiel aus<sup>11</sup>):

**Listing 7.57** src/main/java/com/tutego/insel/game/v1/Buyable.java, Buyable

```
interface Buyable {
    double price();
}
```

**Listing 7.58** src/main/java/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() );
    }

    @Override public String toString() {
        return name + " " + price;
    }
}
```

Die Implementierung nutzt Generics mit Comparable<Magazine>, was wir genauer erst in [Kapitel 11](#), »Generics<T>«, 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.

---

<sup>11</sup> 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.

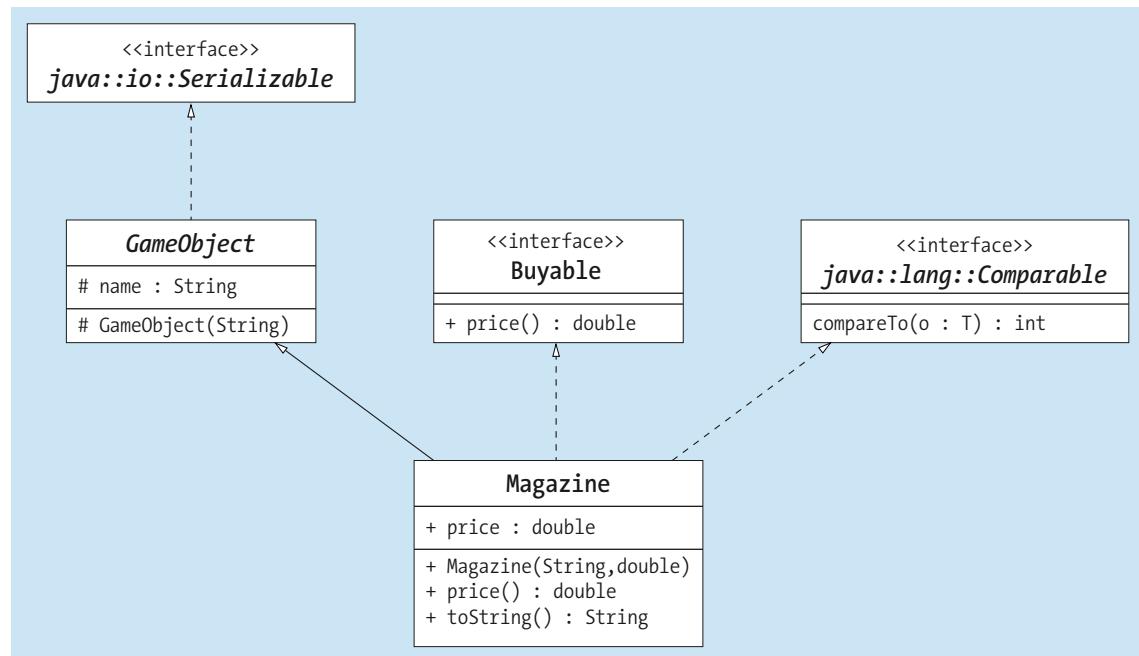


Abbildung 7.13 Die Klasse Magazine mit diversen Obertypen

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 auf `m6` 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 7.59** src/main/java/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 unsere Methode `calculateSum(...)` mit `Magazine`-Argumenten aufrufen, da jedes `Magazine` ja `Buyable` ist:

**Listing 7.60** src/main/java/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 7.61** src/main/java/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 7.62** src/main/java/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 Array, 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.

### 7.7.7 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 Unterklasse wüsste dann nicht, welche Logik sie erbt, also wäre eine spezielle Syntax in Java nötig, die das Dilemma auflösen würde. Das wollen die Sprachdesigner nicht einbauen.

Bei den Schnittstellen gibt es das Problem nicht, denn auch wenn zwei implementierende Schnittstellen die gleiche Operation vorschreiben würden, gäbe es keine zwei verschiedenen Implementierungen von Anwendungslogik. Die implementierende Klasse bekommt sozusagen zweimal die Aufforderung, die Operation zu realisieren. So wie bei folgendem Beispiel: Ein Politiker muss verschiedene Dinge vereinen – er muss sympathisch sein, aber auch durchsetzungsfähig handeln können.

**Listing 7.63** src/main/java/com/tutego/insel/oop/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.

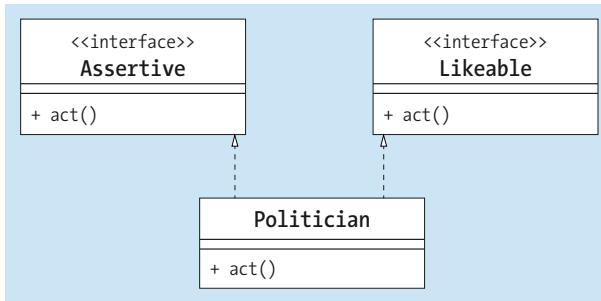


Abbildung 7.14 Eine Klasse erbt von zwei Schnittstellen die gleiche Operation.

### Hinweis

Ein Rückgabetyp gehört in Java nicht zur Signatur einer Methode. Wenn eine Klasse zwei Schnittstellen implementiert und die Signaturen der Operationen aus den Schnittstellen gleich sind, müssen auch die Rückgabetypern gleich sein. Es funktioniert bei der Implementierung nicht, wenn die Signaturen der Methoden aus den Schnittstellen gleich sind (also gleicher Methodename, gleiche Parameterliste), aber die Rückgabetypern nicht typkompatibel sind. Der Grund ist einfach: Eine Klasse kann nicht zwei Methoden mit gleicher Signatur, aber unterschiedlichen Rückgabetypern implementieren. Würde `Assertive` ein `boolean act()` besitzen, müsste `Politician` dann `void act()` und `boolean act()` gleichzeitig realisieren – das geht nicht.



### 7.7.8 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, das besonders abstoßend ist. Zusätzlich soll die Stinkquelle die Stärke der Stinkigkeit in der Einheit Olf angeben. Eine Klasse, die nun `Stinky` implementiert, muss die abstrakten 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.

**Tipp**

Eine Unterschnittsstelle kann eine Operation der Oberschnittstelle »überschreiben«. Auf den ersten Blick ist das nicht sinnvoll, erfüllt aber zwei Zwecke. Erstens: In der Unterschnittstelle kann die API-Dokumentation präzisiert werden.<sup>12</sup> Zweitens: Wegen kovarianter Rückgaben kann eine Operation in der Unterschnittstelle einen spezielleren Rückgabetyp bekommen.

### 7.7.9 Konstantendeklarationen bei Schnittstellen

Schnittstellen können keine Objektvariablen haben und folglich keinen Zustand speichern, aber sie dürfen `static final`-Variablen (benannte Konstanten) deklarieren.

**Beispiel**

Die Schnittstelle `Buyable` soll eine Konstante für einen Maximalpreis deklarieren:

```
interface Buyable {
    int MAX_PRICE = 10_000_000;
    double price();
}
```

Auch wenn die Variablen selbst nach der Initialisierung keine Änderung mehr zulassen, besteht bei mutabel referenzierten Objekten immer noch das Problem, dass eine spätere Änderung an den Objekten möglich ist. Alle Attribute einer Schnittstelle sind immer implizit `public static final`. Das verhindert, dass die Variable neu belegt wird, aber es verhindert keine Objektmanipulation.

**Beispiel und Tipp**

Die Schnittstelle `Volcano` referenziert ein veränderbares `StringBuilder`-Objekt:

```
interface Volcano {
    StringBuilder EYJAFJALLAJÖKULL = new StringBuilder( "Eyjafjallajökull" );
}
```

Da `EYJAFJALLAJÖKULL` eine öffentliche `StringBuilder`-Variable und `StringBuilder` ein veränderbarer Container ist, modifiziert eine Anweisung wie `Volcano.EYJAFJALLAJÖKULL.replace(0, Volcano.EYJAFJALLAJÖKULL.length(), "Vesuvius")`; den Inhalt, was der Idee einer Konstanten absolut widerspricht. Besser ist es, immer immutable Objekte zu referenzieren, also etwa `Strings`. Problematisch sind `Arrays`, in denen Elemente ausgetauscht werden können, sowie alle veränderbaren Objekte wie `Date`, `StringBuilder` oder mutable Datenstrukturen.

<sup>12</sup> Leser können das bei `java.util.Collection` und `java.util.Set` einmal nachschauen.

## 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. Die Basis unseres Beispiels ist die Schnittstelle BaseColors mit ein paar Deklarationen von Farben. Zwei Unterschnittstellen erweitern BaseColors, und zwar CarColors und PlaneColors, die für Farbdeklarationen für Autos und Flugzeuge stehen. Eine besondere Schnittstelle FlyingCarColors erweitert die beiden Schnittstellen CarColors und PlaneColors, denn es gibt auch fliegende Autos, die eine Farbe haben können.

**Listing 7.64** src/main/java/com/tutego/insel/oop/Colors.java

```
interface BaseColors {
    int WHITE    = 0;
    int BLACK   = 1;
    int GREY    = 2;
}
interface CarColors extends BaseColors {
    int WHITE    = 1;
    int BLACK   = 0;
}
interface PlaneColors extends BaseColors {
    int WHITE    = 0;
    int GREY    = 2;
}
interface FlyingCarColors extends CarColors, PlaneColors { }
public class Colors {
    public static void main( String[] args ) {
        System.out.println( BaseColors.GREY );      // 2
        System.out.println( CarColors.GREY );      // 2
        System.out.println( BaseColors.BLACK );     // 1
        System.out.println( CarColors.BLACK );     // 0
        System.out.println( PlaneColors.BLACK );    // 1
        System.out.println( FlyingCarColors.WHITE );
            // 💀 field FlyingCarColors.WHITE is ambiguous
        System.out.println( FlyingCarColors.GREY );
            // 💀 field FlyingCarColors.GREY is ambiguous
    }
}
```

Die erste wichtige Tatsache ist, dass unsere drei Schnittstellen ohne Fehler übersetzt werden können, aber nicht die Klasse Colors. Das Programm und der Compiler zeigen folgendes Verhalten:

- ▶ Schnittstellen vererben ihre Eigenschaften an die Unterschnittstellen. CarColors und auch PlaneColors erben die Farben WHITE, BLACK und GREY aus BaseColors.
- ▶ Konstanten dürfen überdeckt werden. CarColors vertauscht die Farbdeklarationen von WHITE und BLACK und gibt ihnen neue Werte. Wird jetzt der Wert CarColors.BLACK verlangt, liefert die Umgebung den Wert 0, während BaseColors.BLACK 1 ergibt. Auch PlaneColors überdeckt die Konstanten WHITE und GREY, obwohl die Farben mit dem gleichen Wert belegt sind.
- ▶ Erbt eine Schnittstelle von mehreren Oberschnittstellen, so ist es zulässig, dass die Oberschnittstellen jeweils ein gleichlautendes Attribut haben. So erbt etwa FlyingCarColors von CarColors und PlaneColors die Einträge WHITE, BLACK und GREY.
- ▶ Unterschnittstellen können aus zwei Oberschnittstellen die Attribute gleichen Namens übernehmen, auch wenn die Konstanten einen unterschiedlichen Wert haben. Das testet der Compiler nicht. FlyingCarColors bekommt aus CarColors ein WHITE mit 1, aber aus PlaneColors das WHITE mit 0. Daher ist in dem Beispiel Colors auch der Zugriff FlyingCarColors.WHITE nicht möglich und führt zu einem Compilerfehler. Bei der Benutzung muss ein unmissverständlich qualifizierter Name verwendet werden, der deutlich macht, welches Attribut gemeint ist, also zum Beispiel CarColors.WHITE oder PlaneColors.WHITE. Ähnliches gilt für die Farbe GREY. Obwohl Grau durch die ursprüngliche Deklaration bei BaseColors und auch bei der Überschattung in PlaneColors immer 2 ist, ist die Nutzung durch FlyingCarColors.GREY nicht zulässig. Das ist ein guter Schutz gegen Fehler, denn wenn der Compiler dies durchließe, könnte sich im Nachhinein die Belegung von GREY in BaseColors oder PlaneColors ohne Neuübersetzung aller Klassen ändern und zu Schwierigkeiten führen. Diesen Fehler – die Oberschnittstellen haben für eine Konstante unterschiedliche Werte – müsste die Laufzeitumgebung erkennen. Doch das ist nicht möglich, und in der Regel setzt der Compiler die Werte auch direkt in die Aufrufstelle ein, und ein Zugriff auf die Konstantenwerte der Schnittstelle findet nicht mehr statt.

### 7.7.10 Nachträgliches Implementieren von Schnittstellen \*

Implementiert eine Klasse eine bestimmte Schnittstelle nicht, so kann sie auch nicht am dynamischen Binden über diese Schnittstelle teilnehmen, auch wenn sie eine Methode hat, über die eine Schnittstelle abstrahiert. Besitzt zum Beispiel die nichtfinale Klasse FIFA eine öffentliche Methode price(), implementiert aber Buyable mit einer gleich benannten Methode nicht, so lässt sich zu einem Trick greifen, sodass eine Implementierung geschaffen wird, die die existierende Methode aus der Klasse und die der Schnittstelle in die Typierarchie bringt.

```
class FIFA {
    public double price() { ... }
}
```

```
interface Buyable {
    double price();
}

class FIFAisBuyable extends FIFA implements Buyable { }
```

Eine neue Unterklasse `FIFAisBuyable` erbt von der Klasse `FIFA` und implementiert die Schnittstelle `Buyable`, sodass der Compiler die existierende `price()`-Methode mit Vorgabe der Schnittstelle vereinigt. Nun lässt sich `FIFAisBuyable` als `Buyable` nutzen, und dahinter steckt die Implementierung von `FIFA`. Als Unterklasse bleiben auch alle sichtbaren Eigenschaften der Oberklasse erhalten.

### 7.7.11 Statische ausprogrammierte Methoden in Schnittstellen

In der Regel deklariert eine Schnittstelle Operationen, also abstrakte Objektmethoden, die eine Klasse später implementieren muss. Die in Klassen implementierte Schnittstellenmethode kann später wieder überschrieben werden, nimmt also ganz normal an der dynamischen Bindung teil. Einen Objektzustand kann die Schnittstelle nicht deklarieren, denn Objektvariablen sind in Schnittstellen tabu – jede deklarierte Variable ist automatisch statisch, also eine Klassenvariable.

In Schnittstellen sind statische Methoden erlaubt und lassen sich als Utility-Methoden neben Konstanten stellen. Es gibt also statische Klassenmethoden und statische Schnittstellenmethoden; beide werden nicht dynamisch gebunden.

#### Beispiel

Im vorangehenden [Kapitel 6](#), »Eigene Klassen schreiben«, hatten wir eine Schnittstelle `Buyable` deklariert. Die Idee ist, dass alles, was käuflich ist, diese Schnittstelle implementiert und einen Preis hat. Zusätzlich gibt es eine Konstante für einen Maximalpreis:

```
interface Buyable {
    int MAX_PRICE = 10_000_000;
    double price();
}
```

Hinzufügen lässt sich nun eine statische Methode `isValidPrice(double)`, die prüft, ob sich ein Kaufpreis im gültigen Rahmen bewegt:

```
interface Buyable {
    int MAX_PRICE = 10_000_000;
    static boolean isValidPrice( double price ) {
        return price >= 0 && price < MAX_PRICE;
    }
}
```

[zB]

```
    double price();
}
```

Von außen ist dann der Aufruf `Buyable.isValidPrice(123)` möglich.

Alle deklarierten Eigenschaften sind standardmäßig `public`, können aber seit Java 9 auch `privat` sein. Konstanten sind implizit immer statisch. Statische Methoden müssen den Modifizierer `static` tragen, andernfalls gelten sie als abstrakte Methode.



### Hinweis

Statische Schnittstellenmethoden erlauben eine neue Möglichkeit zur Deklaration der `main(...)`-Methode:

```
interface HelloWorldInInterfaces {
    static void main( String[] args ) {
        System.out.println( "Hallo Welt einmal anders!" );
    }
}
```

Das Schlüsselwort `interface` ist vier Zeichen länger als `class`, doch mit der Einsparung von `public` und einem Trenner ergibt sich eine Kürzung von drei Zeichen – wieder eine neue Möglichkeit zum Längefeilschen.

Der Zugriff auf eine statische Schnittstellenmethode ist ausschließlich über den Namen der Schnittstelle möglich, bzw. die Eigenschaften können statisch importiert werden. Bei statischen Methoden von Klassen ist im Prinzip auch der Zugriff über eine Referenz erlaubt (wenn auch unerwünscht), etwa wie bei `new Integer(12).MAX_VALUE`. Allerdings ist das bei statischen Methoden von Schnittstellen nicht zulässig. Implementiert etwa `Car` die Schnittstelle `Buyable`, würde `new Car().isValidPrice(123)` zu einem Compilerfehler führen. Selbst `Car.isValidPrice(123)` ist falsch, was doch ein wenig verwundert, da statische Methoden normalerweise vererbt werden.

Fassen wir die erlaubten Eigenschaften einer Schnittstelle zusammen:

	Attribut	Methode
Objekt-	nein, nicht erlaubt	ja, üblicherweise abstrakt
Statische(s)	ja, als Konstante	ja, immer mit Implementierung

Tabelle 7.5 Erlaubte Eigenschaften einer Schnittstelle

Gleich werden wir sehen, dass Schnittstellenmethoden durchaus eine Implementierung besitzen können, also nicht zwingend abstrakt sein müssen.

## Design

Eine Schnittstelle mit nur statischen Methoden ist ein Zeichen für ein Designproblem und sollte durch eine finale Klasse mit privatem Konstruktor ersetzt werden. Schnittstellen sind immer als Vorgaben zum Implementieren gedacht. Wenn nur statische Methoden in einer Schnittstelle vorkommen, erfüllt die Schnittstelle nicht ihren Zweck, Vorgaben zu machen, die unterschiedlich umgesetzt werden können.

### 7.7.12 Erweitern und Ändern von Schnittstellen

Sind Schnittstellen einmal deklariert und in einer großen Anwendung verbreitet, so sind Änderungen nur schwer möglich, da sie schnell die Kompatibilität brechen. Wird der Name einer Parametervariablen umbenannt, ist das kein Problem. Bekommt aber eine Schnittstelle eine neue Operation, führt das zu einem Compilerfehler, wenn nicht bereits alle implementierenden Klassen diese neue Methode implementieren. Framework-Entwickler müssen also sehr darauf achten, wie sie Schnittstellen modifizieren, doch sie haben es in der Hand, wie weit die Kompatibilität gebrochen wird.

## Geschichtsstunde

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 Parametertyp ändert, dann sind plötzlich alle implementierenden Klassen kaputt. Sun selbst riskierte dies bei der Schnittstelle `java.sql.Connection`. Beim Übergang von Java 5 auf Java 6 wurde die Schnittstelle erweitert, und keine Treiberimplementierung konnte mehr compiliert werden.

## Codekompatibilität und Binärkompatibilität \*

Fügen wir in einer Schnittstelle eine Konstante (`public static final`-Variable) ein oder ändern wir den Namen eines Parameters, so ist das für die implementierenden Klassen in Ordnung, und es führt zu keinem Compilerfehler. Wir sprechen in diesem Fall von Änderungen, die *codekompatibel* sind.

Fügen wir eine neue Operation in eine Schnittstelle ein, führt das sofort zu einem Compilerfehler bei allen implementierenden Klassen. Würden wir jedoch nur die Schnittstelle neu in Bytecode übersetzen, wäre dies zur Laufzeit in Ordnung, denn bekommt eine Schnittstelle eine neue Methode, so ist das für die JVM überhaupt kein Problem. Die Laufzeitumgebung arbeitet auf den Klassendateien selbst, und sie interessiert es nicht, ob eine Klasse brav alle Methoden der Schnittstelle implementiert; sie löst nur Methodenverweise auf. Wenn eine Schnittstelle plötzlich »mehr« vorschreibt, hat die JVM damit kein Problem.

Während also fast alle Änderungen an Schnittstellen zu Compilerfehlern führen, sind einige Änderungen für die JVM in Ordnung. Wir nennen das *Binär-Kompatibilität*. Wenn zum Beispiel die Schnittstelle verändert, neu übersetzt und in den Modulpfad gesetzt wird, ist Folgendes in Ordnung:

- ▶ neue Methoden in Schnittstelle hinzufügen
- ▶ Schnittstelle erbt von einer zusätzlichen Schnittstelle.
- ▶ Hinzufügen oder Löschen einer `throws`-Ausnahme
- ▶ letzten Parametertyp von `T[]` in `T...` ändern
- ▶ neue Konstanten, also statische Variablen hinzufügen

Es gibt allerdings Änderungen, die nicht binärkompatibel sind und zu einem JVM-Fehler führen:

- ▶ Ändern des Methodennamens
- ▶ Ändern der Parametertypen und Umsortieren der Parameter
- ▶ formalen Parameter hinzunehmen oder entfernen

### Strategien zum Ändern von Schnittstellen

Falls die Schnittstelle nicht weit verbreitet wurde, so lassen sich einfacher Änderungen vornehmen. Ist der Name einer Operation zum Beispiel schlecht gewählt, wird ein Refactoring in der IDE den Namen in der Schnittstelle genauso ändern wie auch alle Bezeichner in den implementierenden Klassen. Problematischer ist es, wenn externe Nutzer sich auf die Schnittstelle verlassen. Dann müssen Klienten ebenfalls Anpassungen durchführen, oder Entwickler müssen auf »Schönheitsänderungen« wie das Ändern des Methodennamens einfach verzichten.

Kommen Operationen hinzu, hat sich eine Konvention etabliert, die im Java-Universum oft anzutreffen ist: Soll eine Schnittstelle um Operationen erweitert werden, so gibt es eine neue Schnittstelle, die die alte erweitert und deren Name auf »2« endet; `java.awt.LayoutManager2` ist ein Beispiel aus dem Bereich der grafischen Oberflächen, `Attributes2`, `EntityResolver2`, `Locator2` für XML-Verarbeitung sind weitere.<sup>13</sup>

Default-Methoden sind eine weitere Möglichkeit zur späteren Erweiterung von Schnittstellen. Sie erweitern die Schnittstelle, bringen aber gleich schon eine vorgefertigte Implementierung mit, sodass Unterklassen nicht zwingend eine Implementierung anbieten müssen. Das schauen wir uns jetzt an.

---

<sup>13</sup> Ein Blick auf die API des Eclipse-Frameworks zeigt, dass bei mehr als 3.700 Typen dieses Muster mehr als sechzigmal angewendet wurde (<http://help.eclipse.org/oxygen/topic/org.eclipse.platform.doc.isv/reference/api/index.html?overview-summary.html>).

### 7.7.13 Default-Methoden

Ist eine Schnittstelle einmal verbreitet, so sollte es dennoch möglich sein, Operationen hinzuzufügen. Entwicklern sollte es erlaubt sein, neue Operationen einzuführen, ohne dass Unterklassen verpflichtet werden, diese Methoden zu implementieren. Damit das möglich ist, muss die Schnittstelle eine Standardimplementierung mitbringen. Auf diese Weise ist das Problem der »Pflicht-Implementierung« gelöst, denn wenn eine Implementierung vorhanden ist, haben die implementierenden Klassen nichts zu meckern und können bei Bedarf das Standardverhalten überschreiben. Oracle nennt diese Methoden in Schnittstellen mit vordefinierter Implementierung *Default-Methoden*<sup>14</sup>. Schnittstellen mit Default-Methoden heißen *erweiterte Schnittstellen*.

Eine Default-Methode unterscheidet sich syntaktisch in zwei Aspekten von herkömmlichen implizit abstrakten Methodendeklarationen:

- ▶ Die Deklaration einer Default-Methode beginnt mit dem Schlüsselwort `default`.<sup>15</sup>
- ▶ Statt eines Semikolons markiert bei einer Default-Methode ein Block mit der Implementierung in geschweiften Klammern das Ende der Deklaration. Die Implementierung wollen wir *Default-Code* nennen.

Sonst verhalten sich erweiterte Schnittstellen wie normale Schnittstellen. Eine Klasse, die eine Schnittstelle implementiert, erbt alle Operationen, seien es die abstrakten Methoden oder die Default-Methoden. Falls die Klasse nicht abstrakt sein soll, muss sie alle von der Schnittstelle geerbten abstrakten Methoden realisieren; sie kann die Default-Methoden überschreiben, muss das aber nicht, denn eine Vorimplementierung ist ja schon in der Default-Methode der Schnittstelle gegeben.

#### Hinweis

Erweiterte Schnittstellen bringen »Code« in eine Schnittstelle, doch das ging vorher auch schon, indem zum Beispiel eine implizite öffentliche und statische Variable auf eine Realisierung verweist:

```
interface Comparators {
    Comparator<String> TRIM_COMPARATOR = new Comparator<String>() {
```



<sup>14</sup> Der Name hat sich während der Planung für dieses Feature mehrfach gewandelt. Ganz am Anfang war der Name »defender methods« im Umlauf, dann lange Zeit »virtuelle Erweiterungsmethoden« (engl. *virtual extension methods*).

<sup>15</sup> Am Anfang sollte `default` hinter dem Methodenkopf stehen, doch die Entwickler wollten `default` so wie einen Modifizierer wirken lassen; da Modifizierer aber am Anfang stehen, rutschte auch `default` nach vorne. Eigentlich ist ein Modifizierer auch gar nicht nötig, denn wenn es eine Implementierung, also einen Codeblock, in {} gibt, ist klar, dass es eine Default-Methode wird. Doch die Entwickler wollten eine explizite Dokumentation, so wie auch `abstract` eingesetzt wird – auch dieser Modifizierer bei Methoden wäre eigentlich gar nicht nötig, denn es gibt keinen Codeblock, wenn eine Methode abstrakt ist.

```

    @Override public int compare( String s1, String s2 ) {
        return s1.trim().compareTo( s2.trim() );
    } };
}

```

Die Realisierung nutzt hier eine innere anonyme Klasse, ein Konzept, das genauer in [Kapitel 9, „Geschachtelte Typen“](#), beleuchtet wird.

### 7.7.14 Erweiterte Schnittstellen deklarieren und nutzen

Realisieren wir dies in einem Beispiel. Für Spielobjekte soll ein Lebenszyklus möglich sein; der besteht aus `start()` und `finish()`. Der Lebenszyklus ist als Schnittstelle vorgegeben, die Spielobjektklassen implementieren können. Version 1 der Schnittstelle sieht also so aus:

```

interface GameLifecycle {
    void start();
    void finish();
}

```

Klassen wie `Player`, `Room`, `Door` können die Schnittstelle erweitern, und wenn sie dies tun, müssen sie die beiden Methoden implementieren. Bei Spielobjekten, die diese Schnittstelle implementieren, kann unser Hauptprogramm, das Spiel, diese Methoden aufrufen und den Spielobjekten Rückmeldung geben, ob sie gerade in das Spiel gebracht wurden oder ob sie aus dem Spiel entfernt wurden.

Je länger Software lebt, desto mehr offenbaren sich Fehlentscheidungen beim Design. Die Umstellung einer ganzen Architektur ist eine Mammutaufgabe, einfache Änderungen wie das Umbenennen sind über ein Refactoring schnell erledigt. Nehmen wir an, dass es auch bei unserer Schnittstelle einen Änderungswunsch gibt – nur die Initialisierung und das Ende zu melden, reicht nicht. Geht das Spiel in einen Pausenmodus, soll ein Spielobjekt die Möglichkeit bekommen, im Hintergrund laufende Programme anzuhalten. Das soll durch eine zusätzliche `pause()`-Methode in der Schnittstelle realisiert werden. Hier spielen uns Default-Methoden perfekt in die Hände, denn wir können die Schnittstelle erweitern, aber eine leere Standardimplementierung mitgeben. So müssen Unterklassen die `pause()`-Methode nicht implementieren, können dies aber; Version 2 der nun erweiterten Schnittstelle `GameLifecycle`:

```

interface GameLifecycle {
    void start();
    void finish();
    default void pause() {}
}

```

Klassen, die `GameLifecycle` schon genutzt haben, bekommen von der Änderung nichts mit. Der Vorteil: Die Schnittstelle kann sich weiterentwickeln, aber alles bleibt binärkompatibel, und nichts muss neu compiliert werden. Vorhandener Code kann auf die neue Methode zurückgreifen, die automatisch mit der »leeren« Implementierung vorhanden ist. Außerdem verhalten sich Default-Methoden wie andere Methoden von Schnittstellen auch: Es bleibt bei der dynamischen Bindung, wenn implementierende Klassen die Methoden überschreiben. Wenn eine Unterklasse wie `Flower` zum Beispiel bei der Spielpause nicht mehr blühen möchte, so überschreibt sie die Methode und lässt etwa den Timer pausieren. Eine Tür dagegen hat nichts zu stoppen und kann mit dem Default-Code in `pause()` gut leben. Das Vorgehen ist ein wenig vergleichbar mit normalen nichtfinalen Methoden: Sie können, müssen aber nicht überschrieben werden.

### Hinweis

Statt des leeren Blocks könnte der Rumpf auch `throw new UnsupportedOperationException ("Not yet implemented")`; beinhalten, um anzukündigen, dass es keine Implementierung gibt. So führt eine hinzugenommene Default-Methode zwar zu keinem Compilerfehler, aber zur Laufzeit führen nicht überschriebene Methoden zu einer Ausnahme. Erreicht ist das Gegenteil vom Default-Code, weil eben keine Logik standardmäßig ausgeführt wird; das Auslösen einer Ausnahme zum Melden eines Fehlers wollen wir nicht als Logik ansehen.



### Kontext der Default-Methoden

Default-Methoden verhalten sich wie Methoden in abstrakten Klassen und können alle Methoden der Schnittstelle (inklusive der geerbten Methoden) aufrufen.<sup>16</sup> Die Methoden werden später dynamisch zur Laufzeit gebunden.

Nehmen wir eine Schnittstelle `Buyable` für käufliche Objekte:

```
interface Buyable {
    double price();
}
```

Leider schreibt die Schnittstelle nicht vor, ob Dinge überhaupt käuflich sind. Eine Methode wie `hasPrice()` wäre in `Buyable` ganz gut aufgehoben. Was kann aber die Default-Implementierung sein? Wir können auf `price()` zurückgreifen und testen, ob die Rückgabe ein gültiger Preis ist. Das soll gegeben sein, wenn der Preis echt größer 0 ist.

```
interface Buyable {
    double price();
    default boolean hasPrice() { return price() > 0; }
}
```

---

<sup>16</sup> Und damit lässt sich das bekannte Template-Design-Pattern realisieren.

Implementieren Klassen die Schnittstelle `Buyable`, müssen sie `price()` implementieren, da die Methode keine Default-Methode ist. Doch es ist ihnen freigestellt, `hasPrice()` zu überschreiben, mit eigener Logik füllen und nicht die Default-Implementierung zu verwenden. Wenn implementierende Klassen keine neue Implementierung wählen, bekommen sie den Default-Code und erben eine konkrete Methode `hasPrice()`. In dem Fall geht ein Aufruf von `hasPrice()` intern weiter an `price()` und dann genau an die Klasse, die `Buyable` und die Methode `price()` implementiert. Die Aufrufe sind dynamisch gebunden und landen bei der tatsächlichen Implementierung.



### Hinweis

Eine Schnittstelle kann die Methoden der absoluten Oberklasse `java.lang.Object` ebenfalls deklarieren, etwa um mit Javadoc eine Beschreibung hinzuzufügen. Allerdings ist es *nicht* möglich, mittels Default-Code Methoden wie `toString()` oder `hashCode()` vorzubelegen.

Neben der Möglichkeit, auf Methoden der eigenen Schnittstelle zurückzugreifen, steht auch die `this`-Referenz zur Verfügung. Das ist sehr wichtig, denn so kann der Default-Code an Utility-Methoden delegieren und einen Verweis auf sich selbst übergeben. Hätten wir zum Beispiel schon eine `hasPrice(Buyable)`-Methode in einer Utility-Klasse `PriceUtils` implementiert, so könnte der Default-Code aus einer einfachen Delegation bestehen:

```
class PriceUtils {
    public static boolean hasPrice( Buyable b ) { return b.price() > 0; }
}
interface Buyable {
    double price();
    default boolean hasPrice() { return PriceUtils.hasPrice( this ); }
}
```

Dass die Methode `PriceUtils.hasPrice(Buyable)` für den Parameter den Typ `Buyable` vorsieht und sich der Default-Code mit `this` auf genauso ein `Buyable`-Objekt bezieht, ist natürlich kein Zufall, sondern bewusst gewählt. Der Typ der `this`-Referenz zur Laufzeit entspricht dem der Klasse, die die Schnittstelle implementiert hat und deren Objektexemplar gebildet wurde.

Haben die Default-Methoden weitere Parameter, so lassen sich auch diese an die statische Methode weiterreichen:

```
class PriceUtils {
    public static boolean hasPrice( Buyable b ) { return b.price() > 0; }
    public static double priceOr( Buyable b, double defaultPrice ) {
        if ( b != null && b.price() > 0 )
            return b.price();
        return defaultPrice;
    }
}
```

```

}
interface Buyable {
    double price();
    default boolean hasPrice() { return PriceUtils.hasPrice( this ); }
    default double priceOr( double defaultPrice ) {
        return PriceUtils.defaultPrice( this, defaultPrice );
    }
}

```

Da Schnittstellen auch statische Utility-Methoden mit Implementierung enthalten können, kann der Default-Code auch hier weiterleiten. Allerdings ist zu überlegen, ob in einer Schnittstelle wirklich viel Code untergebracht werden sollte oder dieser nicht besser in eine paketsichtbare Implementierung wandern sollte. Es ist vorzuziehen, die Implementierung auszulagern, damit die Schnittstellen nicht so codelastig werden. Nutzt das JDK Default-Code, so gibt es in der Regel immer eine statische Methode in einer Utility-Klasse.

### 7.7.15 Öffentliche und private Schnittstellenmethoden

Seit Java 9 müssen die statischen und Default-Methoden nicht mehr `public` sein, sie können auch `private` sein. Das ist gut, denn das beugt Codeduplikaten vor; mit privaten Methoden können Programmteile innerhalb der Schnittstelle ausgelagert werden. Private Methoden bleiben natürlich in der Schnittstelle und werden nicht in die implementierenden Klassen vererbt.

### 7.7.16 Erweiterte Schnittstellen, Mehrfachvererbung und Mehrdeutigkeiten \*

Hintergrund zur Einführung von Default-Methoden war die Notwendigkeit, Schnittstellen im Nachhinein ohne nennenswerte Compilerfehler mit neuen Operationen ausstatten zu können. Ideal ist, wenn neue Default-Methoden hinzukommen und Standardverhalten definieren und es dadurch zu keinem Compilerfehler für implementierende Klassen kommt oder zu Fehlern bei Schnittstellen, die erweiterte Schnittstellen erweitern.

Erweiterte Schnittstellen mit Default-Code nehmen ganz normal an der objektorientierten Modellierung teil, können vererbt und überschrieben werden und werden dynamisch gebunden. Nun gibt es einige Sonderfälle, die wir uns anschauen müssen. Es kann vorkommen, dass zum Beispiel

- ▶ eine Klasse von einer Oberklasse eine Methode erbt, aber gleichzeitig von einer Schnittstelle Default-Code für die gleiche Methode, oder
- ▶ eine Klasse von zwei erweiterten Schnittstellen unterschiedliche Implementierungen angeboten bekommt.

Gehen wir verschiedene Fälle durch.

## Überschreiben von Default-Code

Eine Schnittstelle kann andere Schnittstellen erweitern und neuen Default-Code bereitzustellen. Mit anderen Worten: Default-Methoden können andere Default-Methoden aus Oberschnittstellen überschreiben und mit neuem Verhalten implementieren.

Führen wir eine Schnittstelle Priced mit einer Default-Methode ein:

```
interface Priced {
    default boolean hasPrice() { return true; }
}
```

Eine andere Schnittstelle kann die Default-Methode überschreiben:

```
interface NotPriced extends Priced {
    @Override default boolean hasPrice() { return false; }
}
public class TrueLove implements NotPriced {
    public static void main( String[] args ) {
        System.out.println( new TrueLove().hasPrice() );           // false
    }
}
```

Implementiert die Klasse TrueLove die Schnittstelle NotPriced, so ist alles in Ordnung, und es entsteht kein Konflikt. Die Vererbungsbeziehung ist linear TrueLove → NotPriced → Priced.

## Klassenimplementierung geht vor Default-Methoden

Implementiert eine Klasse eine Schnittstelle und erbt außerdem von einer Oberklasse, kann Folgendes passieren: Die Schnittstelle hat Default-Code für eine Methode, und die Oberklasse vererbt ebenfalls die gleiche Methode mit Code. Dann bekommt die Unterklass von zwei Seiten eine Implementierung. Zunächst muss der Compiler entscheiden, ob so etwas überhaupt syntaktisch korrekt ist. Ja, das ist es!

```
interface Priced {
    default boolean hasPrice() { return true; }
}
class Unsaleable {
    public boolean hasPrice() { return false; }
}
public class TrueLove extends Unsaleable implements Priced {
    public static void main( String[] args ) {
        System.out.println( new TrueLove().hasPrice() );   // false
    }
}
```

TrueLove erbt die Implementierung hasPrice() von der Oberklasse Unsaleable und auch von der erweiterten Schnittstelle Priced. Der Code compiliert und führt zu der Ausgabe false – die Klasse mit dem Code »gewinnt« also gegen den Default-Code. Merken lässt sich das ganz einfach an der Reihenfolge class ... extends ... implements ... – es steht extends am Anfang, also haben Methoden aus Implementierungen hier eine höhere Priorität als die aus erweiterten Schnittstellen.

### Default-Methoden aus speziellen Oberschnittstellen ansprechen \*

Eine Unterklasse kann eine konkrete Methode der Oberklasse überschreiben, aber dennoch auf die Implementierung der überschriebenen Methode zugreifen. Allerdings muss der Aufruf über super erfolgen, da sich sonst ein Methodenaufruf rekursiv verfährt.

Default-Methoden können andere Default-Methoden aus Oberschnittstellen ebenfalls überschreiben und mit neuem Verhalten implementieren. Doch genauso wie normale Methoden können sie mit super auf Default-Verhalten aus dem übergeordneten Typ zurückgreifen.

Nehmen wir für ein Beispiel unsere bekannte Schnittstelle Buyable und eine neue erweiterte Schnittstelle PeanutsBuyable an:

```
interface Buyable {
    double price();
    default boolean hasPrice() { return price() > 0; }
}

interface PeanutsBuyable extends Buyable {
    @Override default boolean hasPrice() {
        return Buyable.super.hasPrice() && price() < 50_000_000;
    }
}
```

In der Schnittstelle Buyable sagt der Default-Code von hasPrice() aus, dass alles einen Preis hat, was größer als 0 ist. PeanutsBuyable dagegen nutzt eine erweiterte Definition und implementiert daher das Default-Verhalten neu. Nach den berühmten kopperschen Peanuts<sup>17</sup> ist alles unter 50 Millionen problemlos käuflich und verursacht – zumindest für die Deutsche Bank – keine Schmerzen. In der Implementierung von hasPrice() greift PeanutsBuyable auf den Default-Code von Buyable zurück, um vom Obertyp eine Entscheidung über die Preis-eigenschaft zu bekommen, die aber mit der Und-Verknüpfung noch spezialisiert wird.

### Default-Code für eine Methode von mehreren Schnittstellen erben \*

Wenn eine Klasse aus zwei erweiterten Schnittstellen den gleichen Default-Code angeboten bekommt, führt das zu einem Compilerfehler. Die Klasse RockAndRoll zeigt dieses Dilemma:

---

<sup>17</sup> [https://de.wikipedia.org/wiki/Hilmar\\_Kopper#.E2.80.9EPeanuts.E2.80.9C](https://de.wikipedia.org/wiki/Hilmar_Kopper#.E2.80.9EPeanuts.E2.80.9C)

```

interface Sex {
    default boolean hasPrice() { return false; }
}
interface Drugs {
    default boolean hasPrice() { return true; }
}
public class RockAndRoll implements Sex, Drugs { } // 💀 Compilerfehler

```

Selbst wenn beide Implementierungen identisch wären, müsste der Compiler das ablehnen, denn der Code könnte sich ja jederzeit ändern.

### Mehrfachvererbungsproblem mit super lösen

Die Klasse RockAndRoll lässt sich so nicht übersetzen, weil die Klasse aus zwei Quellen Code bekommt. Das Problem kann aber einfach gelöst werden, indem in RockAndRoll die hasPrice()-Methode überschrieben und dann an eine Methode delegiert wird:

**Listing 7.65** src/main/java/com/tutego/insel/oop/RockAndRoll.java

```

interface Sex {
    default boolean hasPrice() { return false; }
}
interface Drugs {
    default boolean hasPrice() { return true; }
}
public class RockAndRoll implements Sex, Drugs {
    @Override public boolean hasPrice() { return Sex.super.hasPrice(); }
}

```

Im Rumpf der Methode hasPrice() können wir nicht einfach hasPrice() schreiben, denn dann hätten wir einen rekursiven Aufruf. Auch können wir nicht Sex.hasPrice() schreiben, da diese Syntax für den Aufruf von statischen Methoden reserviert ist. Es kommt daher super mit der neuen Schreibweise ins Spiel: Sex.super.hasPrice().

### Abstrakte überschriebene Schnittstellenoperationen nehmen Default-Methoden weg

Default-Methoden haben die interessante Eigenschaft, dass Untertypen den Status von »hat Implementierung« in »hat keine Default-Implementierung« ändern können:

```

interface Priced {
    default boolean hasPrice() { return false; }
}
interface Buyable extends Priced {
    @Override boolean hasPrice();
}

```

Die Schnittstelle Priced bietet eine Default-Methode. Buyable erweitert die Schnittstelle Priced, aber überschreibt die Methode – jedoch nicht mit Code! Dadurch wird sie in Buyable abstrakt. Eine abstrakte Methode kann also durchaus eine Default-Methode überschreiben. Klassen, die Buyable implementieren, müssen also nach wie vor eine hasPrice()-Methode implementieren, wenn sie nicht selbst abstrakt sein wollen. Es ist schon ein interessantes Java-Feature, dass die Implementierung einer Default-Methode in einem Untertyp wieder »weggenommen« werden kann. Bei der Sichtbarkeit ist das zum Beispiel nicht möglich: Ist eine Methode einmal öffentlich, kann eine Unterklasse die Sichtbarkeit nicht einschränken.

Das Verhalten des Compilers hat einen großen Vorteil: Bestimmte Veränderungen der Oberschnittstelle sind erlaubt und haben keine Auswirkungen auf die Untertypen. Nehmen wir an, hasPrice() hätte es in Priced vorher nicht gegeben, sondern nur abstrakt in Buyable. Default-Code ist ja nur eine nette Geste, und diese sollte schmerzlos in Priced integriert werden können. Anders gesagt: Entwickler können in den Basistyp so eine Default-Methode ohne Probleme aufnehmen, ohne dass es in den Untertypen zu Fehlern kommt. Obertypen lassen sich also ändern, ohne die Untertypen anzufassen. Im Nachhinein kann aber zur Dokumentation die Annotation @Override an die Unterschnittstelle gesetzt werden.

Nicht nur eine Unterschnittstelle kann die Default-Methoden »wegnehmen«, sondern auch eine abstrakte Klasse:

```
abstract class Food implements Priced {
    @Override public abstract double price();
}
```

Die Schnittstelle Priced bringt eine Default-Methode mit, doch die abstrakte Klasse Food nimmt diese wieder weg, sodass erweiternde Food-Klassen auf jeden Fall price() implementieren müssen, wenn sie nicht selbst abstract sein wollen.

### 7.7.17 Bausteine bilden mit Default-Methoden \*

Default-Methoden geben Bibliotheksdesignern ganz neue Möglichkeiten. Heute ist noch gar nicht richtig abzusehen, was Entwickler damit machen werden und welche Richtung die Java-API einschlagen wird. Auf jeden Fall wird sich die Frage stellen, ob eine Standardimplementierung als Default-Code in eine Schnittstelle wandert oder wie bisher eine Standardimplementierung als abstrakte Klasse bereitgestellt wird, von der wiederum andere Klassen ableiten. Als Beispiel sei auf die Datenstrukturen verwiesen: Eine Schnittstelle Collection schreibt Standardverhalten vor, AbstractCollection gibt eine Implementierung so weit wie möglich vor, und Unterklassen wie Listen setzen dann noch einmal auf diese Basisimplementierung auf. Erweiterte Schnittstellen können Hierarchien abbauen, denn auf eine abstrakte Basisimplementierung kann verzichtet werden. Auf der anderen Seite kann aber eine abstrakte Klasse einen Zustand über Objektvariablen einführen, was eine Schnittstelle nicht kann.

Default-Methoden können aber noch etwas ganz anderes: Sie können als Bauelemente für Klassen dienen. Eine Klasse kann mehrere Schnittstellen mit Default-Methoden implementieren und erbt im Grunde damit Basisfunktionalität von verschiedenen Stellen. In anderen Programmiersprachen ist das als *Mixin* oder *Trait* bekannt. Das ist ein Unterschied zur Mehrfachvererbung, die in Java nicht zulässig ist. Schauen wir uns diesen Unterschied jetzt einmal genauer an.

### Default-Methoden zur Entwicklung von Traits nutzen

Was ist das Kernkonzept der objektorientierten Programmierung? Wohl ohne zu zögern können wir Klassen, Kapselung und Abstraktion nennen. Klassen und Klassenbeziehungen sind das Gerüst eines jeden Java-Programms. Bei der Vererbung wissen wir, dass Unterklassen Spezialisierungen sind und das liskovsche Substitutionsprinzip (siehe [Abschnitt 7.3.2](#), »Das Substitutionsprinzip«) gilt: Falls ein Typ gefordert ist, können wir auch einen Untertyp übergeben. So sollte perfekte Vererbung aussehen: Eine Unterklasse spezialisiert das Verhalten, aber erbt nicht einfach von einer Klasse, weil diese nützliche Funktionalität hat. Aber warum eigentlich nicht? Als Erstes ist zu nennen, dass das Erben aufgrund der Nützlichkeit oft gegen die Ist-eine-Art-von-Beziehung verstößt und dass uns Java zweitens nur Einfachvererbung mit nur einer einzigen Oberklasse erlaubt. Wenn eine Klasse etwas Nützliches wie Logging anbietet und unsere Klasse davon erbt, kann sie nicht gleichzeitig von einer anderen Klasse erben, um zum Beispiel Zustände in Konfigurationsdaten festzuhalten. Eine unglückliche Vererbung verbaut also eine spätere Erweiterung. Das Problem bei der »Funktionalitätsvererbung« ist also, dass wir uns nur einmal festlegen können.

Wenn eine Klasse eine gewisse Funktionalität einfach braucht, woher soll diese denn dann kommen, wenn nicht aus der Oberklasse? Eigentlich gibt es hier nur eine naheliegende Variante: Die Klasse greift auf andere Objekte per Delegation zurück. Wenn ein Punkt mit Farbe nicht von `java.awt.Point` erben soll, kann ein Farbpunkt einfach in einer internen Variablen einen `Point` referenzieren. Das ist eine Lösung, aber dann nicht optimal, wenn eine Ist-eine-Art-von-Beziehung besteht. Und Schnittstellen wurden ja gerade eingeführt, damit eine Klasse mehrere Typen besitzt. Abstraktionen über Schnittstellen und Oberklassen sind wichtig, und Delegation hilft hier nicht. Gewünscht ist eine Technik, die einen Programmbaustein in eine Klasse setzen kann – im Grunde so etwas wie Mehrfachvererbung, aber doch anders, weil die Bausteine nicht als komplette Typen auftreten; der Baustein selbst ist nur ein Implantat und allein uninteressant. Auch ein Objekt kann von diesem Bausteintyp nicht erzeugt werden.

Am ehesten sind die Bausteine mit abstrakten Klassen vergleichbar, doch das wären Klassen, und Nutzer könnten nur einmal von diesem Baustein erben. Mit den erweiterten Schnittstellen gibt es ganz neue Möglichkeiten: Sie bilden die Bausteine, von denen Klassen Funktionalität bekommen können.<sup>18</sup> Diese Bausteine sind nützlich, denn so lässt sich ein Algorithmus

---

<sup>18</sup> Siehe etwa <http://scg.unibe.ch/archive/papers/SchaO2aTraitsPlusGlue2002.pdf>.

in eine Extra-Kompilationseinheit setzen und leichter wiederverwenden. Ein Beispiel: Nehmen wir zwei erweiterte Schnittstellen `PersistentPreference` und `Logged` an. Die erste erweiterte Schnittstelle soll mit `store()` Schlüssel-Wert-Paare in die zentrale Konfiguration schreiben, und `get()` soll sie auslesen:

```
import java.util.prefs.Preferences;
interface PersistentPreference {
    default void store( String key, String value ) {
        Preferences.userRoot().put( key, value );
    }
    default String get( String key ) {
        return Preferences.userRoot().get( key, "" );
    }
}
```

Die zweite erweiterte Schnittstelle ist `Logged` und bietet uns drei kompakte Logger-Methoden:

```
import java.util.logging.*;
interface Logged {
    default void error( String message ) {
        Logger.getLogger( getClass().getName() ).log( Level.SEVERE, message );
    }
    default void warn( String message ) {
        Logger.getLogger( getClass().getName() ).log( Level.WARNING, message );
    }
    default void info( String message ) {
        Logger.getLogger( getClass().getName() ).log( Level.INFO, message );
    }
}
```

Eine Klasse kann diese Bausteine nun einbauen:

```
class Player implements PersistentPreference, Logged {
    // ...
}
```

Die Methoden sind nun Teil vom `Player` und können auch von Unterklassen überschrieben werden. Als Aufgabe für den Leser bleibt, die Implementierung von `store()` im `Player` zu verändern, sodass der Schlüssel immer mit `player.` beginnt. Die Frage, die der Leser beantworten sollte, ist, ob `store()` von `Player` auf das `store()` von der erweiterten Schnittstelle zugreifen kann.

### Default-Methoden weitergedacht

Für diese Bausteine, also die erweiterten Schnittstellen, gibt es viele Anwendungsfälle. Da die Java-Bibliothek schon 20 Jahre alt ist, würden heute einige Typen anders aussehen. Dass sich Objekte mit `equals(...)` vergleichen lassen können, könnte heute zum Beispiel in einer erweiterten Schnittstelle stehen, etwa so:<sup>19</sup>

```
interface Equals {
    default boolean equals( Object that ) {
        return this == that;
    }
}
```

So müsste `java.lang.Object` die Methode nicht für alle vorschreiben, wobei das sicherlich kein Nachteil ist. Natürlich gilt das Gleiche für die `hashCode()`-Methode, die heutzutage aus einer erweiterten Schnittstelle `Parcelable` stammen könnte.

`java.lang.Number` ist ein weiteres Beispiel. Die abstrakte Basisklasse für Werte repräsentierende Objekte deklariert die abstrakten Methoden `doubleValue()`, `floatValue()`, `intValue()`, `longValue()` und die konkreten Methoden `byteValue()` und `shortValue()`. Bisher erben `AtomicInteger`, `AtomicLong`, `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short` von dieser Oberklasse. Auch diese Funktionalität ließe sich mit einer erweiterten Schnittstelle umsetzen.

### Zustand in den Bausteinen?

Nicht jeder wünschenswerte Baustein ist mit erweiterten Schnittstellen möglich. Ein Grund ist, dass die Schnittstellen keinen Zustand einbringen können. Nehmen wir zum Beispiel einen Container als Datenstruktur, der Elemente aufnimmt und verwaltet. Einen Baustein für einen Container können wir nicht so einfach implementieren, da ein Container Kinder verwaltet, und hierfür ist eine Objektvariable für den Zustand nötig. Schnittstellen haben nur statische Variablen, und die sind für alle sichtbar; und selbst wenn die Schnittstelle eine modifizierbare Datenstruktur referenzieren würde, wäre jeder Nutzer des Container-Bausteins von den Veränderungen betroffen. Da es keinen Zustand gibt, existieren auch für Schnittstellen keine Konstruktoren und folglich auch nicht für solche Bausteine. Denn wo es keinen Zustand gibt, gibt es auch nichts zu initialisieren. Wenn eine Default-Methode einen Zustand benötigt, muss sie selbst diesen Zustand erfragen. Hier lässt sich eine Technik einsetzen, die Oracles Java Language Architect Brian Goetz »virtual field pattern«<sup>20</sup> nennt. Wie das geht, zeigt das folgende Beispiel.

---

<sup>19</sup> Die Schnittstelle compiliert mit der jetzigen Java SE nicht, da eine Default-Methode keine Methode aus `Object` überschreiben kann.

<sup>20</sup> <http://mail.openjdk.java.net/pipermail/lambd-dev/2012-July/005171.html>

Referenziert ein Behälter eine Menge von Objekten, die sortierbar sind, können wir einen Baustein Sortable mit einer Methode `sort()` realisieren. Die Schnittstelle Comparable soll die Klasse nicht direkt implementieren, da ja nur die referenzierten Elemente sortierbar sind, nicht aber Objekte der Klasse selbst; zudem soll eine neue Methode `sort()` in Sortable hinzukommen. Damit das Sortieren gelingt, muss die Implementierung irgendwie an die Daten gelangen, und hier kommt ein Trick ins Spiel: Zwar ist `sort()` eine Default-Methode, doch die erweiterte Schnittstelle Sortable besitzt eine abstrakte Methode `getValues()`, die die Klasse implementieren muss und dem Sortierer die Daten gibt. Im Quellcode sieht das so aus:

**Listing 7.66** src/main/java/com/tutego/insel/oop/SortableDemo.java, Teil 1

```
import java.util.*;
interface Sortable<T extends Comparable<?>> {
    T[] getValues();
    void setValues( T[] values );
    default void sort() {
        T[] values = getValues();
        Arrays.sort( values );
        setValues( values );
    };
}
```

Fassen wir zusammen: Damit `sort()` an die Daten kommt, erwartet Sortable von den implementierenden Klassen eine Methode `getValues()`, und damit die Daten nach dem Sortieren wieder zurückgeschrieben werden können, eine zweite Methode `setValues(...)`. Der Clou ist, dass die spätere Implementierung von Sortable mit den beiden Methoden dem Sortierer Zugriff auf die Daten gewährt – allerdings auch jedem anderem Stück Code, da die Methoden öffentlich sind. Da bleibt ein unschönes »Geschmäckle« zurück.

Ein Nutzer von Sortable soll RandomValues sein; die Klasse erzeugt intern Zufallszahlen.

**Listing 7.67** src/main/java/com/tutego/insel/oop/SortableDemo.java, Teil 2

```
class RandomValues implements Sortable<Integer> {
    private List<Integer> values = new ArrayList<>();
    public RandomValues() {
        Random r = new Random();
        for ( int i = r.nextInt( 20 ) + 1; i > 0; i-- )
            values.add( r.nextInt(10000) );
    }
    @Override public Integer[] getValues() {
        return values.toArray( new Integer[values.size()] );
    }
}
```

```

@Override public void setValues( Integer[] values ) {
    this.values.clear();
    Collections.addAll( this.values, values );
}
}

```

Damit sind die Typen vorbereitet, und eine Demo schließt das Beispiel ab:

**Listing 7.68** src/main/java/com/tutego/insel/oop/SortableDemo.java, Teil 3

```

public class SortableDemo {
    public static void main( String[] args ) {
        RandomValues r = new RandomValues();
        System.out.println( Arrays.toString( r.getValues() ) );
        r.sort();
        System.out.println( Arrays.toString( r.getValues() ) );
    }
}

```

Aufgerufen kommt auf die Konsole zum Beispiel:

```
[2732, 4568, 4708, 4302, 4315, 5946, 2004]
[2004, 2732, 4302, 4315, 4568, 4708, 5946]
```

So interessant diese Möglichkeit auch ist, ein Problem wurde schon angesprochen: Jede Methode in einer Schnittstelle ist `public` oder `private`. Es wäre schön, wenn die Datenzugriffsmethode `protected` und somit nur sichtbar für die implementierende Klasse wäre, aber das geht nicht.



### Warnung!

Natürlich lässt sich mit Rumgetrickse ein Speicherort finden, der Exemplarzustände speichert. Es lässt sich zum Beispiel in der Schnittstelle ein Assoziativspeicher referenzieren, der eine `this`-Instanz mit einem Objekt assoziiert. Ein Container-Baustein, der mit `add()` Objekte in eine Liste setzt und sie mit `iterable()` herausgibt, könnte so aussehen:

```

interface ListContainer<T> {
    Map<Object, List<Object>> $ = new HashMap<>();
    default void add( T e ) {
        if ( ! $.containsKey( this ) )
            $.put( this, new ArrayList<Object>() );
        $.get( this ).add( e );
    }
    default public Iterable<T> iterable() {
        if ( ! $.containsKey( this ) )

```

```

        return Collections.emptyList();
    return (Iterable<T>) $.get( this );
}
}

```

Nicht nur die öffentliche Konstante \$ ist ein Problem, sondern auch, dass es ein großartiges doppeltes Speicherloch ist. Ein Exemplar der Klasse, die diese erweiterte Schnittstelle nutzt, kann nicht so einfach entfernt werden, denn in der Sammlung ist noch eine Referenz auf das Objekt, und diese Referenz verhindert eine automatische Speicherbereinigung. Selbst wenn dieses Objekt weg wäre, hätten wir noch all die referenzierten Kinder der Sammlung in der Map. Das Problem ist nicht wirklich zu lösen, und hier müsste mit schwachen Referenzen tief in die Java-Voodoo-Kiste gegriffen werden. Alles in allem, keine gute Idee, und Java-Chefentwickler Brian Goetz macht auch klar:

*»Please don't encourage techniques like this. There are a zillion ›clever‹ things you can do in Java, but shouldn't. We knew it wouldn't be long before someone suggested this, and we can't stop you. But please, use your power for good, and not for evil. Teach people to do it right, not to abuse it.«<sup>21</sup>*

Daher: Es ist eine schöne Spielerei, aber der Zustand sollte eine Aufgabe der abstrakten Basisklassen oder des Delegates sein.

## Zusammenfassung

Was wir in den letzten Beispielen zu den Bausteinen gemacht haben, war, ein Standardverhalten in Klassen einzubauen, ohne dass dabei der Zugriff auf die nur einmal existierende Basisklasse nötig war und ohne dass die Klasse an Hilfsklassen delegierte. In dieser Arbeitsweise können Unterklassen in jedem Fall die Methoden überschreiben und spezialisieren. Wir haben es also mit üblichen Klassen zu tun und mit erweiterten Schnittstellen, die nicht selbst eigenständige Entitäten bilden. In der Praxis wird es immer Fälle geben, in denen für eine Umsetzung eines Problems entweder eine abstrakte Klasse oder eine erweiterte Schnittstelle in Frage kommt. Wir sollten uns dann noch einmal an die Unterschiede erinnern: Eine abstrakte Klasse kann Objektvariablen haben und Methoden aller Sichtbarkeiten und sie auch final setzen, sodass sie nicht mehr überschrieben werden können. Eine Schnittstelle dagegen ist ohne Zustand und mit puren virtuellen und öffentlichen Methoden darauf ausgelegt, dass die Implementierung überschrieben werden kann.

### 7.7.18 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

---

21 <http://mail.openjdk.java.net/pipermail/lambda-dev/2012-July/005166.html>

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 die Datenstruktur 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, obwohl es statische Variablen und statische Methoden gibt:

```
import java.util.*;

public interface PropertyReader {

    Properties DEFAULT_PROPERTIES = new Properties();

    static { // 💀 Compilerfehler: "Interfaces can't have static initializers"
        DEFAULT_PROPERTIES.setProperty( "date", LocalDate.now().toString() );
    }

    Properties getProperties();
}
```

Zwar sind statische Initialisierungsblöcke nicht möglich, aber mit drei Tricks kann die Initialisierung erreicht werden. Wir müssen dazu etwas auf anonyme Klassen vorgreifen, ein Thema, das Kapitel 9, »Geschachtelte Typen«, genauer aufgreift.

### Konstanteninitialisierung über innere anonyme 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", LocalDate.now().toString() );
    } };

    Properties getProperties();
}
```

Ein Beispielprogramm zeigt die Nutzung:

**Listing 7.69** src/main/java/com/tutego/insel/oop/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 geschachtelte Klassen, Lösung B

Mit einem anderen Trick lassen sich auch diese Hürden nehmen. Die Idee liegt in der Einführung zweier Hilfskonstrukte:

- ▶ einer statischen geschachtelten Klasse, die wir \$\$ nennen wollen. Sie enthält einen statischen Initialisierungsblock, der auf DEFAULT\_PROPERTIES zugreift und das Properties-Objekt initialisiert.

- einer Konstanten \$ 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 statische geschachtelte Klassen und Konstanten von Schnittstellen nicht privat sein können – nur Methoden können privat sein – und so unglücklicherweise von außen zugänglich sind, geben wir ihnen die kryptischen Namen \$ und \$\$, sodass sie nicht so attraktiv erscheinen:

**Listing 7.70** src/main/java/com/tutego/insel/oop/PropertyReader.java

```
import java.util.*;

public interface PropertyReader {

    Properties DEFAULT_PROPERTIES = new Properties();

    $$ $ = new $$();

    static final class $$ {
        static {
            DEFAULT_PROPERTIES.setProperty( "date", LocalDate.now().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 geschachtelte Klasse, 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 Erzeugmethode einer eigenen internen Klasse, sodass die Initialisierung zu DEFAULT\_PROPERTIES = \$\$.\$() wird:

**Listing 7.71** src/main/java/com/tutego/insel/oop/PropertyReader2.java

```
import java.util.*;

public interface PropertyReader2 {
```

```

Properties DEFAULT_PROPERTIES = $$.$$();

static class $$ {
    static Properties $$() {
        Properties p = new Properties();
        p.setProperty( "date", LocalDate.now().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.



### 7.7.19 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 ein Objekt einen gewollten Typ einnimmt.

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`: Diesen Typ implementieren viele Hörer in der Java-Bibliothek.
- ▶ `java.io.Serializable`: Zustände eines Objekts lassen sich in einen Datenstrom schreiben – mehr dazu folgt in [Kapitel 19](#), »Einführung in Dateien und Datenströme«.

#### Hinweis

Seit es das Sprachmittel der Annotationen gibt, sind Markierungsschnittstellen bei neuen Bibliotheken nicht mehr anzutreffen.



### 7.7.20 (Abstrakte) Klassen und Schnittstellen im Vergleich

Eine abstrakte Klasse und eine Schnittstelle mit abstrakten Methoden sind sich sehr ähnlich: Beide schreiben den Unterklassen bzw. den implementierten Klassen Operationen vor, die implementiert werden 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 Designphase an, wenn Gemeinsamkeiten in eine Oberklasse ausgelagert werden sollen. Abstrakte Klassen können zudem Objektzustände enthalten, was Schnittstellen nicht können.

Im Design ist weiterhin der Grundgedanke bei Schnittstellen: Wenn es Vorschriften für Verhalten ist, dann ist eine Schnittstelle goldrichtig. Bei Basisimplementierungen kommen dann abstrakte Klassen ins Spiel, die in der Java-Bibliothek oft auf `Abstract` enden.

#### Wie wo was dynamisch binden

Es gibt bei Methoden von konkreten Klassen, abstrakten Klassen und Schnittstellen Unterschiede, wo der Aufruf letztendlich landet. Nehmen wir folgende Methode an:

```
void fun( T t ) {
    t.m();
}
```

Fordert die Methode ein Argument vom Typ `T` und ruft auf der Parametervariablen `t` die Methode `m()` auf, so können wir Folgendes festhalten:

- ▶ Ist `T` eine finale Klasse, so wird immer die Methode `m()` von `T` aufgerufen, da es keine Unterklassen geben kann, die `m()` überschreiben.
- ▶ Ist `T` eine nichtfinale Klasse und `m()` eine finale Methode, wird genau `m()` aufgerufen, weil keine Unterklasse `m()` überschreiben kann.
- ▶ Ist `T` eine nichtfinale Klasse und `m()` keine finale Methode, so könnten Unterklassen von `T` `m()` überschreiben, und `t.m()` würde dann dynamisch die überschriebene Methode aufrufen.
- ▶ Ist `T` eine abstrakte Klasse und `m()` eine abstrakte Methode, so wird in jedem Fall eine Realisierung von `m()` in einer Unterklasse aufgerufen.
- ▶ Ist `T` eine Schnittstelle und `m()` keine Default-Implementierung, so wird in jedem Fall eine Implementierung `m()` einer implementierenden Klasse aufgerufen.
- ▶ Ist `T` eine Schnittstelle und `m()` eine Default-Implementierung, so kann `t.m()` bei der Default-Implementierung landen oder bei einer überschriebenen Version einer implementierenden Klasse.

## 7.8 SOLIDe Modellierung

Wer gute objektorientierte Software schreiben möchte, sollte sich an einige Designprinzipien halten. Es sind Best-Practice-Methoden, die natürlich nicht zwingend sind, aber in der Regel das Design verbessern.

### 7.8.1 DRY, KISS und YAGNI

Die ersten drei Regeln sind:

- ▶ **DRY (Don't Repeat Yourself):** »Wiederhole dich nicht.« Codeduplizierung sollte vermieden und doppelter Code in Methoden ausgelagert werden. Das heißt auch, dass bestehender Code – aus etwa eigenen Bibliotheken, der Java SE oder quelloffenen Bibliotheken – verwendet werden soll.
- ▶ **KISS (Keep It Simple, Stupid):** »Halte es einfach und idiotensicher«. Ein Problem soll einfach und leicht verständlich gelöst werden. Für Entwickler bedeutet es: einfacher Code, wenige Zeilen Code, auf den ersten Blick verständlich.
- ▶ **YAGNI (You Ain't Gonna Need It):** Das Prinzip »Du wirst es nicht brauchen« soll uns daran erinnern, einfachen Code zu schreiben und nur das zu programmieren, was im Moment in der Anforderung auch erwartet wird. Es ist ein zentraler Punkt von Extreme Programming (XP) und der Idee »Implementiere immer die einfachste mögliche Lösung, die funktioniert«, denn wenn etwas programmiert wird, was später nie produktiv wird, ist es Zeit- und Geldverschwendungen, und der Code muss dennoch dokumentiert, gewartet und getestet werden.

### 7.8.2 SOLID

Michael Feathers hat die Abkürzung *SOLID* eingeführt und fünf Punkte benannt, die weiterhin guten objektorientierten Entwurf ausmachen. Die einzelnen Kriterien selbst stammen von unterschiedlichen Autoren.

#### S: Single Responsibility Principle (SRP)

Etwas flapsig ausgedrückt steht das Prinzip für: »Mache genau eine Sache, die aber richtig.« Ein Typ sollte genau eine Verantwortung (engl. *responsibility*) haben, sodass bei Änderungen im besten Fall auch nur eine Stelle angepasst werden muss, nicht viele. Das Gegenteil sind so genannte Gott-Klassen, die alles können – ein Anti-Pattern. Robert Martin, der das SRP in seinem Buch »Agile Software Development: Principles, Patterns, and Practices« beschreibt, sagt auch: »Es sollte nie mehr als einen Grund geben, eine Klasse zu ändern.« Was heißt das nun praktisch?

Nehmen wir an, eine Person-Klasse speichert Namen, PLZ und Alter. An PLZ und Alter gibt es Anforderungen: Eine deutsche PLZ besteht nur aus Ziffern, ist 5 Stellen lang, und ein Alter ist

sicherlich nicht negativ und nach oben beschränkt. Allerdings sind diese beiden Validierungen zwei unterschiedliche Dinge, also übernimmt die Person-Klasse Verantwortlichkeiten, die an sich mit einer Person nichts zu tun haben. Demnach gibt es zwei Gründe, warum die Klasse bei einer Änderung der Validierung angepasst werden muss, und zwei Gründe sind mehr als ein Grund und folglich ein Bruch des SRP.

Treibt die Modellierung das SRP ins Extrem, entstehen sehr viele kleine Typen. Damit ist auch dem Codeversteher nicht geholfen, wenn Verantwortlichkeiten wegen Unübersichtlichkeit nicht mehr zu verstehen sind.

### O: Open/closed principle

Bertrand Meyer formuliert 1988 in seinem Buch »Object-Oriented Software Construction«, dass Module sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein müssen. Unter dem Begriff »Modul« müssen sich Java-Entwickler einen Typ vorstellen. Eine herkömmliche Klasse ist insbesondere mit privaten Zuständen geschlossen für Modifikationen, aber eine Unterklasse erlaubt ohne Codeänderungen der Oberklasse die Erweiterung um neue Zustände oder durch das Überschreiben von Methoden eine Anpassung einer Implementierung. Eine Unterklasse darf Methoden allerdings keine andere Semantik geben, sonst würde das die Geschlossenheit brechen.

### L: Liskov Substitution Principle (LSP)

Barbara Liskov hielt 1987 den Vortrag »Data abstraction and hierarchy«, in dem es um die Tatsache ging, dass es möglich sein sollte, Objekte in Programmen mit Objekten eines Untertyps zu ersetzen, ohne dass die Korrektheit leidet. Damit der Austausch funktioniert, muss natürlich der Untertyp wissen, was »korrekt« ist, damit Methoden nicht eine falsche Implementierung realisieren, die das Verhalten brechen. Flapsig ausgedrückt: Kinder müssen das Verhalten der Eltern erben und respektieren. In Java ist das nicht einfach, denn syntaktische Konstrukte wie Preconditions, Postconditions und Invarianten gibt es nicht; Java-Entwickler müssen also rein aus dem Javadoc, der textuellen Information also, das herausziehen, was ein korrektes Verhalten ist.

### I: Interface Segregation Principle (ISP)

Das ISP wird Robert Cecil Martin zugeschrieben, als er für Xerox am Druckersystem arbeitete. Die zentrale Aussage ist: »Viele Client-spezifische Schnittstellen sind besser als eine allgemeine Schnittstelle.« Der Client ist der Nutzer eines Java-Typs, unter Schnittstelle ist verallgemeinert das Angebot an Methoden gemeint. Praktisch heißt das Folgendes: Es gibt Typen, die sehr viele Methoden haben und dann ein »allgemeiner« Typ wären. Werden solche Objekte herumgereicht, dann bekommen die Programmstellen immer das gesamte Objekt mit allen Methoden. Das komplette Angebot an Methoden ist aber nicht immer nötig und viel-

leicht sogar gefährlich. Besser ist es, die API klein zu halten und damit nur den verschiedenen Stellen zu ermöglichen, was auch benötigt wird.

#### D: Dependency Inversion Principle (DIP)

»Hänge nur von Abstraktionen ab, nicht von Spezialisierungen.« So hat es Robert Cecil Martin formuliert, ursprünglich etwas länger.<sup>22</sup> Gut zu erkennen ist das Prinzip in der Schichtenarchitektur: Eine obere Schicht greift auf Dienste einer tieferen Schicht zurück. Die obere Schicht sollte sich aber nicht an konkrete Typen klammern, sondern nur von Basistypen wie Java-Schnittstellen abhängen. In diesem Zusammenhang passt gut das Prinzip »Programmieren gegen Schnittstellen«.

#### Das große Ganze

Wie in einem Quentin-Tarantino-Film hängt alles irgendwie in einem großen Designuniversum zusammen. Jedoch haben die Entwurfspraktiken Schwerpunkte: Das SRP nimmt sich Typen vor und die Architektur im Großen. Das Open-Closed-Prinzip handelt von Typen und ihren Erweiterungen. LSP handelt von Vererbung und Untertypen. Und das ISP handelt von Geschäftslogik und Abhängigkeiten der Typen.

### 7.8.3 Sei nicht STUPID

Jedem »tue« in SOLID steht ein »lass es« in *STUPID* gegenüber. Das Akronym steht für die die dunkle Seite:

- ▶ **Singleton:** Ein Singleton ist ein Objekt, das es im System nur einmal geben kann. Solche Objekte gibt es immer wieder, und sie sind an sich nichts Schlimmes. Problematisch ist jedoch, dass viele Entwickler das Singleton selbst als Klasse schreiben, und dann entsteht schnell eine Implementierung, die sich durch den globalen Zustand schlecht testen lässt. Besser ist es, Frameworks zu nutzen, die für uns dann ein Exemplar bereitstellen.
- ▶ **Tight Coupling (enge Kopplung):** Das Ziel guten Entwurfs ist die Reduktion von Abhängigkeiten; auf je weniger Module/Pakete/Typen ein Stück Code zurückgreift, desto besser. Einfaches Indiz ist die Menge der import-Deklarationen am Anfang eines Typs.
- ▶ **Untestability (Nicht-Testbarkeit):** Wird erst nach dem Design und der Programmierung über das Testen nachgedacht, ist es oft schon zu spät – schnell entsteht schwer zu testender Code, besonders wenn die Kopplung zu eng ist. Besser ist der Ansatz der testgetriebenen Entwicklung, bei der die Testbarkeit das Design beeinflusst. Am besten überlegen sich Designer und Entwickler im Vorfeld, wie eine bestimmte Klasse und Funktionalität getestet werden kann, bevor es an die intensive Implementierung geht.

---

<sup>22</sup> Die erste Fassung lautet: »A. High-level modules should not depend on low-level modules. Both should depend on abstractions. B. Abstractions should not depend on details. Details should depend on abstractions.«

- ▶ **Premature Optimization (voreilige Optimierung)**: Entwickler meinen, ein Gefühl dafür zu haben, welche Programmteile Performance verschlingen und welche Teile schnell sind. Oft irren sie sich, verschenken aber viel Zeit zur Optimierung dieser vermeindlich langsamen Stellen. Der beste Ansatz ist, nach KISS eine einfache Lösung zu realisieren und dann über einen Profiler sich genau die Stellen aufzeigen zu lassen, die Nacharbeit erfordern.
- ▶ **Indescriptive Naming (nichtbeschreibende Benennung)**: Variablennamen wie one, z, l, myvariable, var1, val10, theInt, aDouble, \_1bool, button123 sind wenig sprechend und müssen vermieden werden; der Programmleser sollte sofort wissen, worum es sich bei der Variable handelt.
- ▶ **Duplikationen**: Code, der mit kleinen Änderungen 1:1 kopiert wurde, ist zu vermeiden. Codeduplikate lassen sich mit Werkzeugen und IDE-Plugins relativ gut finden.

## 7.9 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-Standardbibliothek zu machen, damit die Zusammenhänge klarer werden.

# Kapitel 8

## Ausnahmen müssen sein

»*Wir sind in Sicherheit! Er kann uns nicht erreichen!*«  
›Sicher?‹  
›Ganz sicher! Bären haben Angst vor Treibsand!‹«  
– Hägar, Dik Browne (1917–1989)

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.

### 8.1 Problembereiche einzäunen

In den frühen Programmiersprachen gab es für Routinen keine andere Möglichkeit, als über den Rückgabewert einen Fehlschlag anzuzeigen – in der Programmiersprache C ist das auch heute noch der Fall. 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

Die Java-Bibliothek geht bei den Methoden `delete()`, `mkdir(...)`, `mkdirs(...)` und `renameTo(...)` der Klasse `File` nicht mit gutem Beispiel voran. Anstatt über eine Ausnahme anzuzeigen, dass die Operation nicht gegückt 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.

[zB]

### 8.1.1 Exceptions in Java mit try und catch

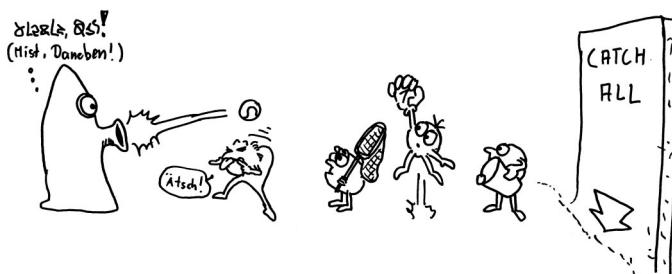
Bei der Verwendung von Exceptions wird der Programmcode nicht durch Abfrage des Rückgabestatus unterbrochen. Ein besonders ausgezeichnetes Programmstück überwacht die ordentliche Ausführung und ruft im Fehlerfall speziellen Programmcode zur Behandlung auf.

Den überwachten Programmreich (Block) leitet das Schlüsselwort `try` ein. Dem `try`-Block folgt in der Regel<sup>1</sup> ein `catch`-Block, in dem Programmcode steht, der die Ausnahme 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
```

Fehler führen zu Ausnahmen, und diese Ausnahmen behandelt ein `catch`-Block. Hinter `catch` folgt also der Programmblock, der beim Auftreten einer Ausnahme ausgeführt wird, um den Fehler abzufangen (daher der Ausdruck `catch`). Der Fehler kann auf der Kommandozeile gemeldet oder etwa in einen Logger geschrieben werden. Ob es zur Laufzeit wirklich zu einem Fehler kommt, ist nicht bekannt, aber wenn, dann ist eine Behandlung vorhanden.

Es ist nach der Fehlerbehandlung nicht mehr so einfach möglich, an der Stelle fortzufahren, an der der Fehler auftrat. Auch lässt sich im Nachhinein nicht wirklich feststellen, an welcher Stelle genau der Fehler aufgetreten ist, wenn es in einem großen `try`-Block mehrere ausnahmenauslösende Stellen gibt. Andere Programmiersprachen erlauben das durchaus.



### 8.1.2 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:

---

<sup>1</sup> In manchen Fällen auch ein `finally`-Block, sodass es dann ein `try-finally` wird.

```
int vatRate = Integer.parseInt( "19" );
```

In dem Beispiel ist eine Konvertierung möglich, und die Methode führt die Umwandlung ohne Ausnahme aus. Anders sieht das aus, wenn der String keine Zahl repräsentiert:

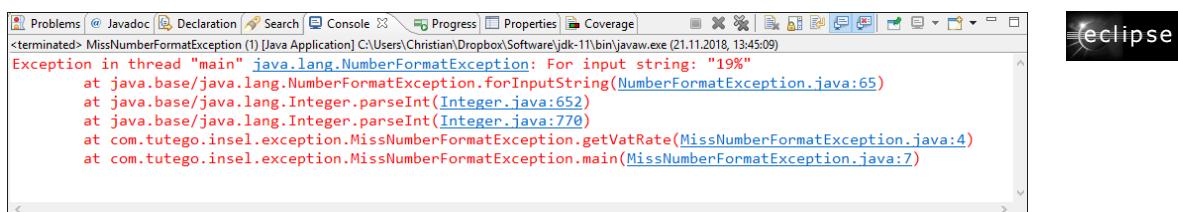
**Listing 8.1** src/main/java/com/tutego/insel/exception/MissNumberFormatException.java

```
package com.tutego.insel.exception;          /* 1 */
public class MissNumberFormatException {      /* 2 */
    public static int getVatRate() {           /* 3 */
        return Integer.parseInt( "19%" );       /* 4 */
    }                                         /* 5 */
    public static void main( String[] args ) { /* 6 */
        System.out.println( getVatRate() );      /* 7 */
    }                                         /* 8 */
}
```

Die Ausführung des Programms bricht mit einer Ausnahme ab, und die virtuelle Maschine gibt uns automatisch eine Meldung aus:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "19%"
  at java.base/java.lang.NumberFormatException.forInputString(
    NumberFormatException.java:65)
  at java.base/java.lang.Integer.parseInt(Integer.java:652)
  at java.base/java.lang.Integer.parseInt(Integer.java:770)
  at c.t.i.e.MissNumberFormatException.getVatRate(MissNumberFormatException.java:4)
  at c.t.i.e.MissNumberFormatException.main(MissNumberFormatException.java:7)
```

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 der Ausnahme führte (Fehlerausgaben wie diese haben wir schon im Abschnitt »Auf null geht nix, nur die NullPointerException« in Abschnitt 3.7.1 beobachtet).



**Abbildung 8.1** Eclipse zeigt eine Exception im Ausgabefenster rot. Praktischerweise verhalten sich die Fehlermeldungen wie Hyperlinks: Ein Klick, und Eclipse zeigt die Zeile, die die Exception auslöst.



```

Exception in thread "main" java.lang.NumberFormatException: For input string: "19%"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at MissNumberFormatException.getVatRate(MissNumberFormatException.java:3)
    at MissNumberFormatException.main(MissNumberFormatException.java:6)

Process finished with exit code 1

```

Abbildung 8.2 Auch bei IntelliJ führt ein Klick auf die Fehlerstelle und in den Quellcode.

### Dokumentierte Ausnahmen

Eine Ausnahme kommt nicht wirklich überraschend, und Entwickler müssen sich darauf vorbereiten, dass, wenn sie etwas Falsches an Methoden oder Konstruktoren übergeben, diese schimpfen. Im besten Fall erklärt die API-Dokumentation, welche Eingaben gültig sind 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:

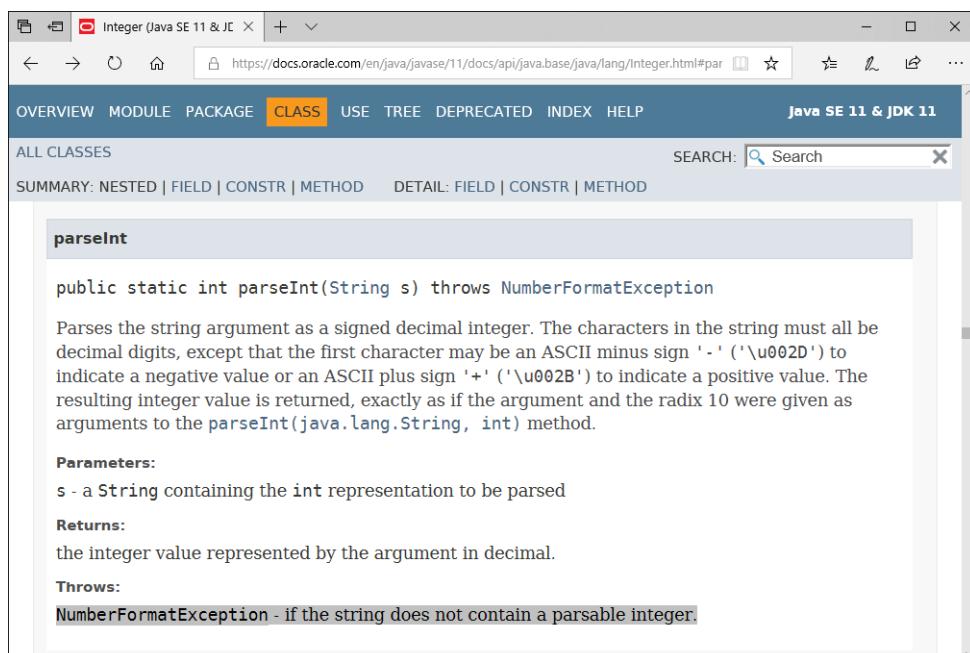


Abbildung 8.3 Javadoc dokumentiert die Ausnahme.

### 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(...)` die Ausnahme ausgelöst hat und nicht irgendeine andere Methode.

### Eine NumberFormatException auffangen

Dass ein Programm einfach so abbricht und die JVM endet, ist üblicherweise keine Lösung. Ausnahmen sollten aufgefangen und gemeldet werden. Um Ausnahmen 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 diese Ausnahme auch, und weil ohne die aufgefangene Ausnahme das Programm abbricht, soll nun die `NumberFormatException` aufgefangen werden. Dabei kommt die try-catch-Konstruktion zum Einsatz:

**Listing 8.2** src/main/java/com/tutego/insel/exception/CatchNumberFormatException.java, main()

```
String stringToConvert = "19%";
double vat = 19;
try {
    vat = Integer.parseInt( stringToConvert );
}
catch ( NumberFormatException e ) {
    System.err.printf( "'%s' kann man nicht in eine Zahl konvertieren!%n",
        stringToConvert );
}
System.out.printf( "Weiter geht's mit MwSt=%g%n", vat );
```

Die gesamte Ausgabe ist:

```
'19%' kann man nicht in eine Zahl konvertieren!
Weiter geht's mit MwSt=19,0000
```

Der try-Anweisung folgt ein Block, genannt *try-Block*. Wir nutzen ihn in Kombination mit einer catch-Klausel. Der Code `catch(NumberFormatException e)` deklariert einen *Exception-Handler* – er fängt alles auf, was vom Ausnahmetyp `NumberFormatException` ist. Die Variable `e` ist ein *Exception-Parameter*. Die Nutzung von `var` ist nicht erlaubt. `Integer.parseInt("19")`

führt, da der String keine Zahl ist, zu einer `NumberFormatException`, die wir behandeln. Da Ausnahmen Objekte sind, referenziert die Variable `e` dieses Ausnahmeobjekt.

Nach dem Auffangen ist der Fehler wie weggeblasen, und mit der Konsolenausgabe geht es ganz normal weiter.

### 8.1.3 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. Sie können eine sinnvolle Behandlung beinhalten oder auch einfach leer sein. Ein leerer `catch`-Block ist in der Regel wenig sinnvoll, weil dann die Ausnahme klammheimlich unterdrückt wird (das wäre genauso wie ignorierte Statusrückgabewerte von C-Funktionen). Das Mindeste ist eine minimale Fehlerausgabe via `System.out.println(e)` oder das informativere `e.printStackTrace(...)` für eine `Exception e` oder das Loggen dieser Ausnahme. Noch besser ist das aktive Reagieren, denn die Ausgabe selbst behandelt diese Ausnahme nicht! Im `catch`-Block ist es durchaus legitim, wiederum andere Ausnahmen auszulösen und somit die Ausnahme 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.

### 8.1.4 Wiederholung abgebrochener Bereiche \*

Es gibt in Java bei Ausnahmen bisher keine von der Sprache unterstützte Möglichkeit, an den Punkt zurückzukehren, der die Ausnahme 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 8.3** src/main/java/com/tutego/insel/exception/ContinueInput.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 Ausnahme durch, so beenden wir die Schleife mit `break`. Kommt es zu einer `NumberFormatException`, dann wird `break` nicht ausgeführt, und der Programmfluss führt wieder in die Endlosschleife.

### 8.1.5 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 5.9.2](#), »Yes we can, yes we scan – 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.

Ausnahmen können von allen Codeblöcken ausgelöst werden, in Methoden genauso wie in Konstruktoren. `Integer.parseInt(String)` ist zum Beispiel eine Methode, die eine Ausnahme auslöst, wenn der String keine Zahl ist, und das wird in der API-Dokumentation erklärt. Genauso kündigt die API-Dokumentation vom Konstruktor der Klasse `URL` an, dass eine Ausnahme ausgelöst wird, wenn die URL falsch formuliert wird (etwa als "telefon://0123-123123"). Die Methode löst eine `IOException` aus, wenn es keinen Zugriff auf die Webseite gibt. Ausnahmen von Konstruktoren sind ein wichtiges Werkzeug, da sie effektiv verhindern, dass ein Objekt in einen falschen Startzustand kommt. Nicht nur der Konstruktor von `URL` löst im Fehlerfall eine `IOException` aus, sondern auch die `URL`-Methode `openStream()`.

Beide Ausnahmen sind so genannte *geprüfte Ausnahmen* – Details in [Abschnitt 8.3](#) –, 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. Dabei spielt es keine Rolle, ob die Ausnahmen vielleicht nie auftreten können – sie müssen behandelt werden.

**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 `URL` connection.

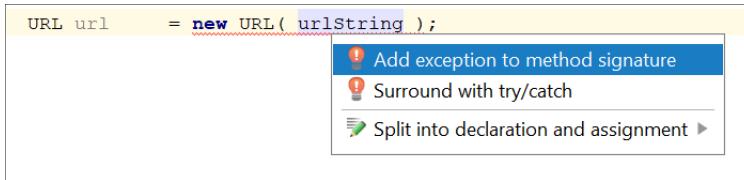
**Throws:**  
`IOException` - if an I/O exception occurs.

**See Also:**  
`openConnection()`, `URLConnection.getInputStream()`

Abbildung 8.4 Die API-Dokumentation zeigt die Ausnahmen.



Abbildung 8.5 Eine nicht behandelte Ausnahme wird als Fehler angezeigt.



IJ

**Abbildung 8.6** IntelliJ zeigt Fehler an, und mit der Tastenkombination **Alt**+**←** können sie direkt behoben werden.

Wir müssen uns diesen potenziellen Ausnahmen also stellen und daher die Problemzonen in einen `try`-Block schreiben:

**Listing 8.4** src/main/java/com/tutego/insel/exception/FindAllEmailAddresses.java

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Scanner;
import java.util.regex.Pattern;

public class FindAllEmailAddresses {

    public static void main( String[] args ) {
        printAllEMailAddresses( "https://www.rheinwerk-verlag.de/hilfe/service-
            und-kontakt_8" );
    }

    static void printAllEMailAddresses( String urlString ) {
        try {
            URL url      = new URL( urlString );
            Scanner scanner = new Scanner( url.openStream() );
            Pattern pattern = Pattern.compile( "[\\w-]+@[\\w[\\w-]*\\.\\w{2,3}" );
            for ( String email; (email = scanner.findWithinHorizon( pattern, 0 )) != null; )
                System.out.println( email );
        }
        catch ( MalformedURLException e ) {
            System.err.println( "URL ist falsch aufgebaut!" );
        }
        catch ( IOException e ) {
            System.err.println( "URL konnte nicht geöffnet werden!" );
        }
    }
}
```

Tritt beim Erzeugen des URL-Objekts oder bei der Verbindung eine Ausnahme auf, fängt der try-Block diese ab, und der catch-Teil bearbeitet sie. Einem try-Block können mehrere catch-Klauseln zugeordnet sein, die verschiedene Typen von Ausnahmen auffangen. Dass die Resource Scanner am Ende nicht geschlossen wird, ist in dem Beispiel in Ordnung, weil das Programm direkt beendet wird und so alle Ressourcen freigegeben werden.



Abbildung 8.7 Einen try-catch-Block kann Eclipse mit der Tastenkombination [Strg]+[1] selbst anlegen. Auch bietet Eclipse an, die Ausnahme an den Aufrufer weiterzuleiten.

### 8.1.6 Ablauf einer Ausnahmesituation

Das Laufzeitsystem erzeugt ein Ausnahmeeobjekt, 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 gleich sehen werden). Wenn die erste catch-Klausel nicht zur Art der augetretenen Ausnahme 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 jede `IOException` passt, und `MalformedURLException` ist eine Unterklasse von `IOException`. Alle anderen Anweisungen in den catch-Blöcken würden dann nicht ausgeführt; der Compiler erkennt dieses Problem und gibt einen Fehler aus.

### 8.1.7 throws im Methodenkopf angeben

Neben dem »Einzäunen« von problematischen Blöcken durch einen try-Block mit angehängtem catch-Block zur Behandlung 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 die Ausnahme kümmern.

Wir können unsere Methode `printAllEmailAddresses(String)` so umschreiben, dass sie die Ausnahmen nicht mehr selbst abfängt, sondern nach oben weiterleitet:

**Listing 8.5** src/main/java/com/tutego/insel/exception/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|-]*\\.\\.[a-z]{2,3}" );

    for ( String email; (email = scanner.findWithinHorizon( pattern, 0 )) != null; )
        System.out.println( email );
}
```

Nun ist `main(...)` am Zug und muss sich mit `MalformedURLException` und `IOException` herumärgern:

**Listing 8.6** src/main/java/com/tutego/insel/exception/FindAllEmailAddresses2.java, main()

```
public static void main( String[] args ) {
    try {
        printAllEmailAddresses( "https://www.rheinwerk-verlag.de/hilfe/service-
und-kontakt_8" );
    }
    catch ( MalformedURLException e ) {
        System.err.println( "URL ist falsch aufgebaut!" );
    }
    catch ( IOException e ) {
        System.err.println( "URL konnte nicht geöffnet werden!" );
    }
}
```

Dadurch steigt die Ausnahme 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 in [Abschnitt 8.2](#), »Die Klassenhierarchie der Ausnahmen«, später mehr.

### 8.1.8 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', 'I', '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 8.7** src/main/java/com/tutego/insel/exception/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. (Das Programm ist klein, und die JVM gibt nach dem Programmende alle nativen Betriebssystemressourcen wieder frei. Da unser Java-Programm aber länger laufen kann, ist es guter Stil,

nach dem Abschluss der Dateioperationen Ressourcen zu schließen.) 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 die Ausnahme ordentlich abfangen und dokumentieren:

**Listing 8.8** src/main/java/com/tutego/insel/exception/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 einer Ausnahme. 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 Quellcode-Duplizierung, 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 eine Ausnahme gab oder ob es keine Ausnahme gab und die Routine glatt durchlief. Das ist genau, was wir hier bei der Ressourcenfreigabe brauchen. Da `finally` immer ausgeführt wird, wird die Datei geschlossen (und der interne File-Handle freigegeben), wenn alles gut ging – und ebenso im Fall einer Ausnahme:

**Listing 8.9** src/main/java/com/tutego/insel/exception/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 lokaler Variablen einen größeren Radius – größer, als er eigentlich sein sollte. Mit einem Extrablock lässt sich

das lösen, sieht aber nicht so hübsch aus. Das spezielle Sprachkonstrukt *try mit Ressourcen* löst das elegant; Informationen dazu folgen in Abschnitt 8.6.1, »try mit Ressourcen«.

### Hinweis

In einem try-Block deklarierte lokale Variablen sind nicht im angehängten catch- oder finally-Block sichtbar.



## Zusammenfassung

Nach einem catch (oder mehreren) kann optional ein finally-Block folgen. Die Laufzeitumgebung führt die Anweisungen im finally-Block immer aus, egal, ob eine Ausnahme auftrat oder die Anweisungen im try-Block optimal durchliefen. Das heißt, der Block wird auf jeden Fall ausgeführt – lassen wir `System.exit(int)` 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 angehängter 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 keine Ausnahme behandelt werden soll, aber unabhängig von möglichen Ausnahmen immer Programmcode abgearbeitet werden soll – ein typisches Beispiel ist die Ressourcenfreigabe.

Kommen wir zu unserem Programm zurück, das die Größe eines GIF-Bildes ermittelt. Wenn beim IO-Fehler eben nichts zu retten ist, geben wir die Ausnahme an den Aufrufer weiter, ohne es jedoch zu versäumen, die in der Methode angeforderten Ressourcen wieder freizugeben:

**Listing 8.10** src/main/java/com/tutego/insel/exception/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 einer Ausnahme 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 eine mögliche Ausnahme beim Schließen nach oben geleitet, bei `ReadGifSize.java` jedoch nicht, denn dort ist 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.

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 8.6](#), »Automatisches Ressourcen-Management (try mit Ressourcen)«, vorstellt.

## 8.2 Die Klassenhierarchie der Ausnahmen

Eine Ausnahme ist ein Objekt vom Typ einer Klasse. Die Exception-Klassen selbst sind von anderen Ausnahmeklassen abgeleitet.

### 8.2.1 Eigenschaften des Exception-Objekts

Das Exception-Objekt, das uns in der `catch`-Klausel 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 8.11** src/main/java/com/tutego/insel/exception/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%"
java.lang.NumberFormatException: For input string: "19%"
java.lang.NumberFormatException: For input string: "19%"
at java.base/java.lang.NumberFormatException.forInputString(
    NumberFormatException.java:65)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
at java.base/java.lang.Integer.parseInt(Integer.java:770)
at c.t.i.e.NumberFormatExceptionElements.main(NumberFormatExceptionElements.java:7)

```

Die Ausgabe besteht aus dem Klassennamen der Exception, der Meldung und dem StackTrace. `printStackTrace(...)` ist parametrisiert und kann auch in einen Ausgabekanal geschickt werden.

### 8.2.2 Basistyp `Throwable`

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 nichtabstrakte Klasse).

Von dort aus verzweigt sich die Hierarchie der Ausnahmearten 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 Ausnahmen, dass sie zur Beendigung des Programms führen und vom Programmierer nicht weiter beachtet werden müssen und

sollten. Throwable vererbt eine Reihe von nützlichen Methoden, die in der folgenden Grafik sichtbar sind. Sie fasst gleichzeitig die Vererbungsbeziehungen noch einmal zusammen.

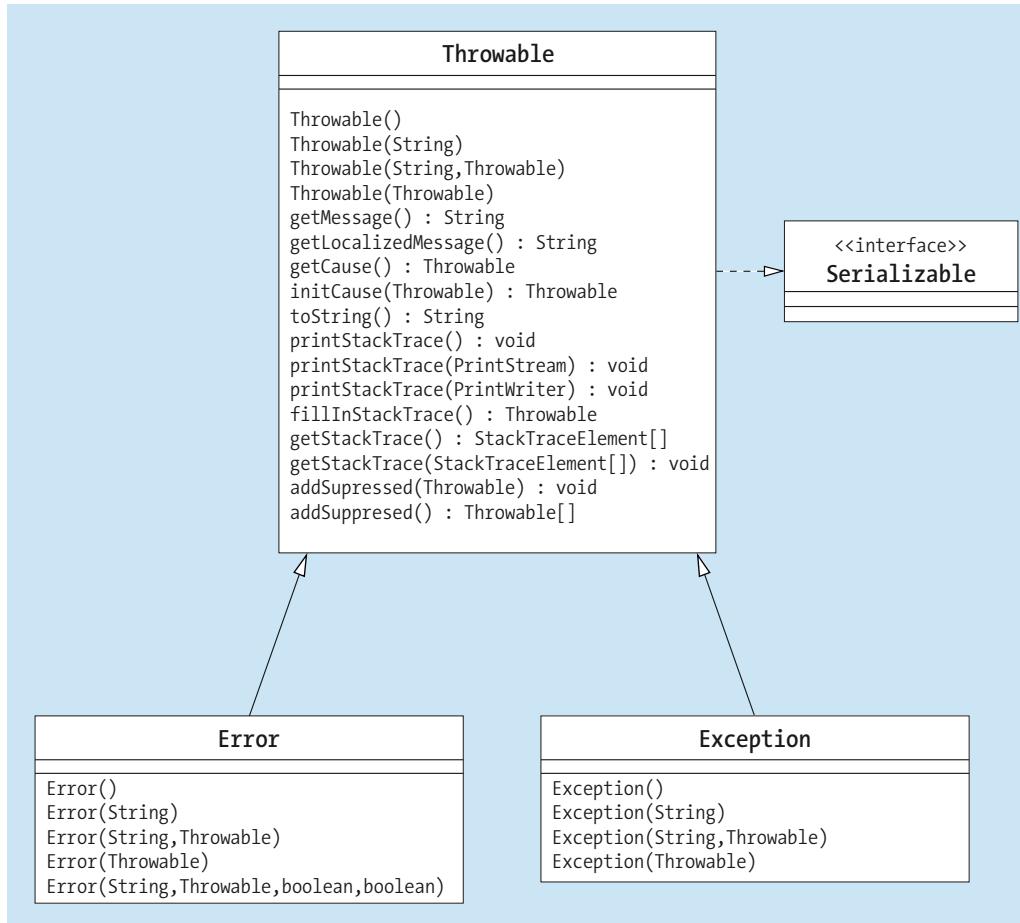


Abbildung 8.8 UML-Diagramm der wichtigen Oberklasse Throwable

### 8.2.3 Die Exception-Hierarchie

Jede Benutzerausnahme 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 bzw. Unterhierarchien auf. [Abbildung 8.9](#) zeigt einige Unterklassen der Klasse `Exception`.

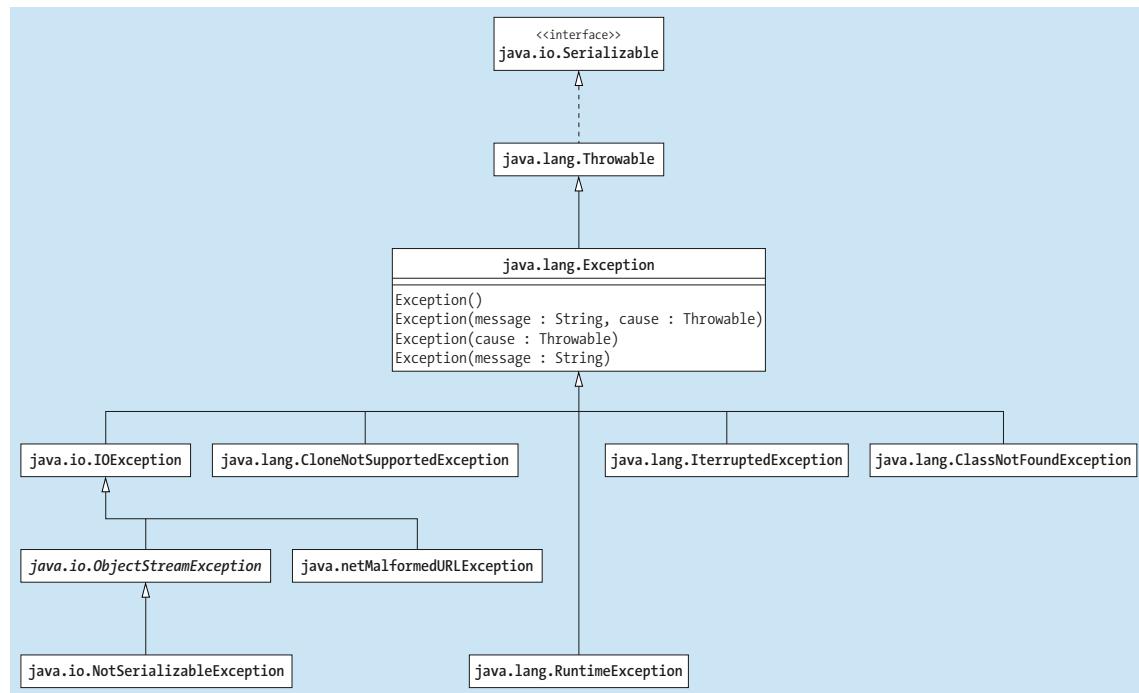


Abbildung 8.9 Ausgewählte Unterklassen von Exception

#### 8.2.4 Oberausnahmen auffangen

Eine Konsequenz der Hierarchien besteht darin, dass es ausreicht, eine Ausnahme 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, das eine `FileNotFoundException` und eine `IOException` einzeln behandelt. Ist die Behandlung aber die gleiche, lässt sie sich wie folgt zusammenfassen:

Listing 8.12 src/main/java/com/tutego/insel/exception/ReadGifSizeShort.java, main()

```
RandomAccessFile f = null;
```

```
try {
    f = new RandomAccessFile( "duke.gif", "r" );
    f.seek( 6 );
```

```

        System.out.printf( "%s x %s Pixel%", 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 ) { }
    }
}

```

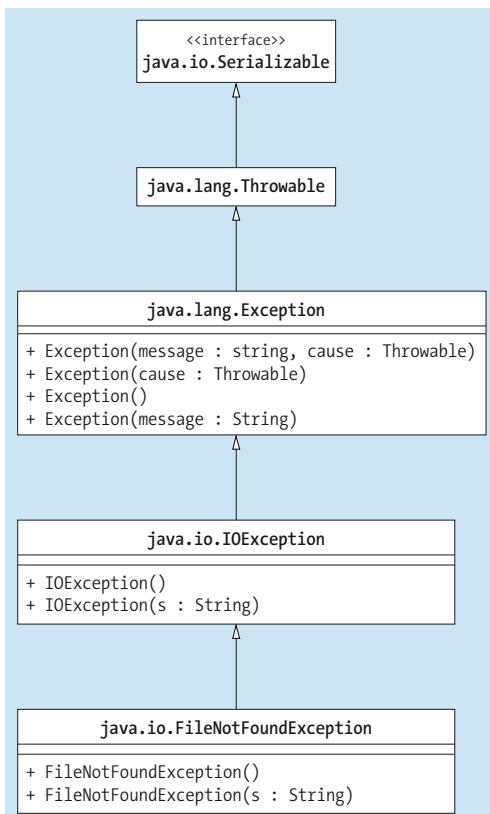


Abbildung 8.10 IOException im Klassendiagramm

Angst davor, dass wir den Typ der Ausnahme später nicht mehr unterscheiden können, brauchen wir nicht zu haben, denn die an die `catch`-Klausel gebundenen Variablen können wir mit `instanceof` weiter verfeinern. Aus Gründen der Übersichtlichkeit sollte diese Technik jedoch sparsam angewendet werden. Ausnahmen, die unterschiedlich behandelt werden müssen, verdienen immer getrennte `catch`-Klauseln. Das trifft z. B. auf `FileNotFoundException` und `IOException` zu.

### 8.2.5 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)` Ausnahmen vom Typ `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 8.1 Die Reihenfolge der catch-Blöcke spielt eine Rolle.

### 8.2.6 Alles geht als Exception durch

Löst ein Programmblöck etwa eine `IOException`, `MalformedURLException` und eine `FileNotFoundException` aus, soll die Ausnahme aber gleich behandelt werden, so fängt ein `catch (IOException e)` die beiden Typen `FileNotFoundException` und `MalformedURLException` gleich mit ab, da beide Unterklassen von `IOException` sind. So behandelt ein Block alle drei Ausnahmetypen. Das ist praktisch.

Nun gibt es jedoch auch Ausnahmen, die in der Vererbungsbeziehung nebeneinanderliegen, etwa `SQLException` und `IOException`. Was ist, wenn die Ausnahmebehandlung gleich sein soll? Die naheliegende Idee ist, in der 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 die Ausnahme 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 Ausnahmebehandlungen 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 Ausnahmetypen 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 `ArithmetricException` sollen nicht mitbehandelt werden. Das zentrale Problem ist hier, dass diese Ausnahmen ungeprüfte Ausnahmen vom Typ

RuntimeException sind und RuntimeException eine Unterklasse von Exception ist. Fangen wir alle Exception-Typen, so wird alles mitgefangen – und RuntimeException eben auch. Es ist nicht möglich, alle Nicht-Laufzeitausnahmen 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(String[]) alles weiterleitet

Ist die Ausnahmebehandlung in einem Hauptprogramm ganz egal, so können wir alle Ausnahmen auch an die Laufzeitumgebung weiterleiten, die dann das Programm – genau genommen den Thread – im Fehlerfall abbricht:

**Listing 8.13** src/main/java/com/tutego/insel/exception/IDontCare.java, main()

```
public static void main( String[] args ) throws Exception {
    Scanner in = new Scanner( Paths.get( "lyrics.txt" ) );
    System.out.println( in.nextLine() );
}
```

Das funktioniert, da alle Ausnahmen von der Klasse Exception<sup>2</sup> abgeleitet sind. Wird die Ausnahme 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 Ausnahmen sollten in jedem Fall behandelt werden.

### 8.2.7 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 Ausnahmen 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 Ausnahmen gleich aussieht. Um Quellcode-Duplizierung zu vermeiden, sollte der Programmcode zusammengefasst werden. Nehmen wir an, die Behandlung der Ausnahmen SQLException und IOException soll gleich sein. Das mit einem catch(Exception e) zu lösen, ist keine gute Idee und sollte nie im Pro-

---

<sup>2</sup> Genauer gesagt, sind alle Ausnahmen in Java von der Exception-Oberklasse Throwable abgeleitet.

grammcode vorkommen, denn dann würden auch andere Ausnahmen mitgefangen, die mit der eigentlichen Ausnahme nichts zu tun hatten. Zum Glück gibt es eine elegante Lösung.

### multi-catch

Eine spezielle Schreibweise für catch-Klauseln erlaubt es, 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 senkrechte Strich 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 ) { ... }
```

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 8.14** basiscolors.txt

```
0 000000
1 ff0000
8 00ff00
9 ffffff
```

Eine eigene Methode `readColorTable(String)` soll die Datei einlesen und ein `int`-Array 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 Ladefehler, 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 Arrays sind diverse Ausnahmen möglich:

- ▶ `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). Die Ausnahme kommt vom Scanner.
- ▶ `ArrayIndexOutOfBoundsException`: Der Index liegt nicht im Bereich von 0 bis 255.

Während die erste Ausnahme beim Dateisystem zu suchen ist, sind die zwei unteren Ausnahmen – 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 8.15** src/main/java/com/tutego/insel/exception/ReadColorTable.java, ReadColorTable

```
public class ReadColorTable {

    private static int[] readColorTable( String filename ) {
        Scanner input = null;
        int[] colors = new int[ 256 ];
        try {
            input = new Scanner( Paths.get( filename ),
                StandardCharsets.ISO_8859_1.name() );
            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 {
            if ( input != null ) input.close();
        }
        return null;
    }

    public static void main( String[] args ) {
        readColorTable( "basiscolors.txt" );
    }
}
```

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%n", e );
}
```

*Multi-catch*-Blöcke sind also 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 Standardtests 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 ) { }
```

Während allerdings eine Umsortierung der Zeilen die Fehler korrigiert – wie in [Abschnitt 8.2.5](#), »Schon gefangen?«, erwähnt –, spielt die Reihenfolge bei *multi-catch* keine Rolle.

## 8.3 RuntimeException muss nicht aufgefangen werden

Einige Arten von Ausnahmen können potenziell an vielen Programmstellen auftreten, etwa eine ganzzahlige Division durch null<sup>3</sup> oder ungültige Indexwerte beim Zugriff auf Array-Elemente. Treten solche Ausnahmen beim Programmlauf auf, liegt dem in der Regel ein Denkfehler des Programmierers zugrunde, und das Programm sollte normalerweise nicht versu-

---

<sup>3</sup> Fließkommadivisionen durch 0,0 ergeben entweder  $\pm$  unendlich oder NaN.

chen, die ausgelöste Ausnahme aufzufangen und zu behandeln. Daher gibt es in der Java-API mit der Klasse `RuntimeException` eine Unterklasse von `Exception`, die Programmierfehler 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. Doch drückt es aus, dass der Compiler sich für diese Ausnahmen nicht interessiert, sondern erst die JVM zur Laufzeit.)



### Hinweis

Es funktioniert gut, eine `RuntimeException` als Selbst-schuld-Fehler zu sehen. Durch sachgemäße Prüfung z. B. der Wertebereiche würden viele `RuntimeExceptions` nicht entstehen.

#### 8.3.1 Beispiele für `RuntimeException`-Klassen

Die Java-API bietet insgesamt eine große Anzahl von `RuntimeException`-Klassen, und es werden immer mehr. Tabelle 8.2 listet einige bekannte Ausnahmetypen auf und zeigt, welche Operationen die Ausnahmen auslösen. Wir greifen hier schon auf spezielle APIs zurück, die erst später im Buch vorgestellt werden.

Unterklasse 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>	Typumwandlung ist zur Laufzeit nicht möglich. So löst <code>String s = (String) new Object();</code> eine <code>ClassCastException</code> mit der Meldung »java.lang.Object cannot be cast to java.lang.String« aus.
<code>EmptyStackException</code>	Der Stapelspeicher ist leer. <code>new java.util.Stack().pop()</code> provoziert den Fehler.
<code>IllegalArgumentExeption</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.

Tabelle 8.2 `RuntimeException`-Klassen

Unterklasse von RuntimeException	Was den Fehler auslöst
IllegalMonitorStateException	Ein Thread möchte warten, hat aber den Monitor nicht. Ein Beispiel: new String().wait();
NullPointerException	Meldet einen der häufigsten Programmierfehler, beispielsweise durch ((String) null).length();
UnsupportedOperationException	Operationen sind nicht gestattet, etwa durch java.util.Arrays.asList(args).add("jv").

Tabelle 8.2 RuntimeException-Klassen (Forts.)

### 8.3.2 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 Abfangen besteht, heißen die aus RuntimeException hervorgegangenen Ausnahmen auch *ungeprüfte Ausnahmen* bzw. *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.

#### Style

Eine RuntimeException wird nicht im Methodenkopf angegeben, kann aber im Javadoc dokumentiert werden.

## 8.4 Harte Fehler – Error \*

Klassen, die von java.lang.Error abgeleitet sind, repräsentieren harte Fehler, die mit der JVM in Verbindung stehen. Anders reagieren dagegen die von Exception abgeleiteten Klassen – sie stehen für allgemeine Programmfehler. Beispiele für konkrete Error-Klassen sind:

- ▶ AnnotationFormatError
- ▶ AssertionError
- ▶ AWTError
- ▶ CoderMalfunctionError<sup>4</sup>

<sup>4</sup> Die lustigste Fehlerklasse, wie ich finde. Sie könnte bei einigen Entwicklern bei jeder Methode ausgelöst werden.

- ▶ FactoryConfigurationError (XML-Fehler)
- ▶ IOError
- ▶ 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. 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.

## 8.5 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.

### 8.5.1 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 8.16** com/tutego/insel/exception/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 Alter-Übergaben oder solche mit Null nicht gestattet sind und zu einer Ausnahme führen. Im ersten Schritt baut dazu `new` 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 8.17** src/main/java/com/tutego/insel/exception/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:

```
Exception in thread "main" java.lang.IllegalArgumentException: Kein Alter <= 0 erlaubt!
at com.tutego.insel.exception.v1.Player.<init>(Player.java:9)
at com.tutego.insel.exception.v1.Player.main(Player.java:28)
```



### Hinweis

Löst ein Konstruktor eine Ausnahme aus, ist eine Nutzung wie die folgende problematisch:

```
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 Ausnahmen 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 Ausnahmeeobjekt mit einer detaillierten Fehlerangabe.

### 8.5.2 Vorhandene Runtime-Ausnahmetypen 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 Standardfälle, wie falsche Argumente oder falscher Programmstatus, decken Standard-Exception-Klassen ab.



### Hinweis

Entwickler sollten nie `throw new Exception()` oder sogar `throw new Throwable()` schreiben, sondern sich immer konkreter Unterklassen bedienen.

Einige Standard-Runtime-Exception-Unterklassen des `java.lang`-Pakets folgen nun in der Übersicht.

### IllegalArgumentException

Die `IllegalArgumentException` zeigt an, dass ein Parameter nicht korrekt angegeben ist. Dieser Ausnahmetyp lässt sich somit nur bei Konstruktoren oder Methoden ausmachen, denen fehlerhafte Argumente übergeben wurden. Oft ist der Grund die Missachtung des Wertebereichs. Wenn die Werte grundsätzlich korrekt sind, darf dieser Ausnahmetyp nicht ausgelöst werden. Dazu folgen gleich noch ein paar mehr Details.

### IllegalStateException

Objekte haben in der Regel Zustände. Gilt es, Operationen auszuführen, sind aber die Zustände 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.<sup>5</sup>

### 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 Aufrüfer glauben zu lassen, die Methode führt etwas aus, sollten diese Methoden eine `UnsupportedOperationException` auslösen. Die API-Dokumentation kennzeichnet abstrakte Methoden, die Unterklassen vielleicht nicht realisieren wollen, als *optionale Operationen*.

Unglücklicherweise gibt es auch eine `javax.naming.OperationNotSupportedException`. Doch diese sollte nicht verwendet werden. Sie ist speziell für Namensdienste vorgesehen und auch keine `RuntimeException`.

<sup>5</sup> Im .NET-Framework gibt es eine vergleichbare Ausnahme, die `System.InvalidOperationException`. In Java trifft der Name allerdings das Problem etwas besser.

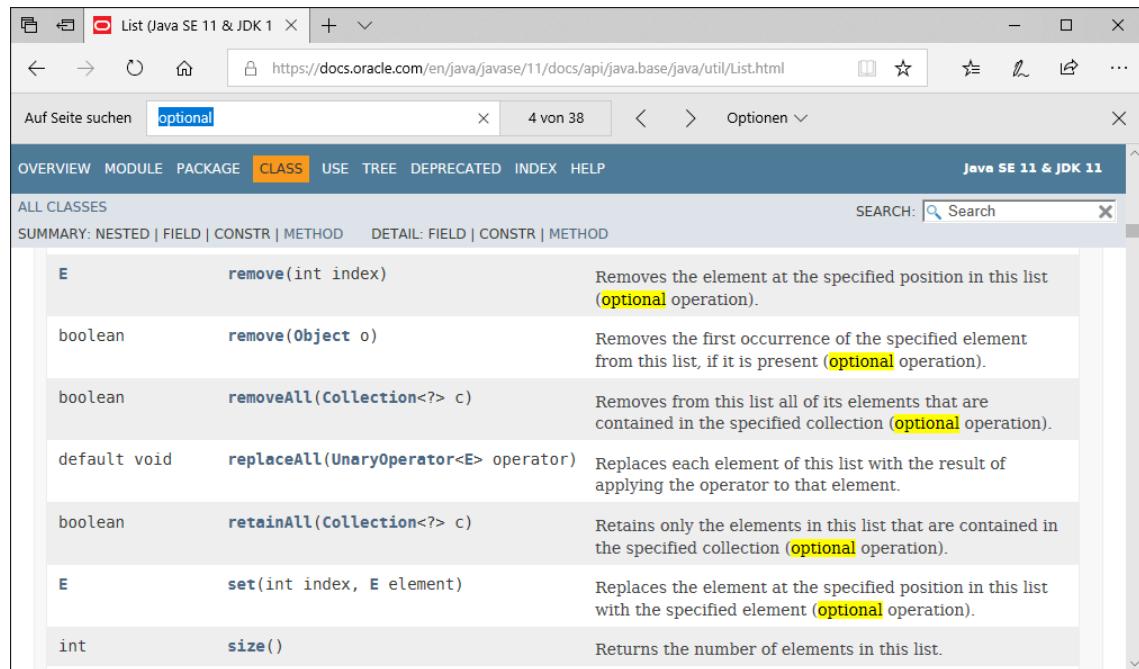


Abbildung 8.11 Optionale Operationen in der Schnittstelle java.util.List

### IndexOutOfBoundsException

Eine `IndexOutOfBoundsException` löst die JVM automatisch aus, wenn zum Beispiel ein Programm die Grenzen eines Arrays missachtet. Wir können diesen Ausnahmetyp selbst immer dann nutzen, wenn wir Indexzugriffe 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`. In der Regel nutzen Programmierer diese Typen aber nicht. Inkonsistenzen gibt es beim Einsatz von `IllegalArgumentException` und `IndexOutOfBoundsException`. Ist etwa der Index falsch, so entscheiden sich einige Autoren für den ersten Ausnahmetyp, andere für den zweiten. Beides ist prinzipiell gültig. Die `IndexOutOfBoundsException` ist aber konkreter und zeigt eher ein Implementierungsdetail an. Der parametrisierte Konstruktor `IndexOutOfBoundsException(int)` nimmt im Ganzzahl-Parameter den falschen Index an; der sollte immer gemeldet werden, um die Fehlersuche zu vereinfachen.

Listing 8.18 java/lang/IndexOutOfBoundsException.java, `IndexOutOfBoundsException(int)`

```
public IndexOutOfBoundsException(int index) {
    super("Index out of range: " + index);
}
```

### Eigene NullPointerException auslösen?

Eine `NullPointerException` gehört mit zu den häufigsten Ausnahmen. Die JVM löst diese Ausnahme 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, diese Ausnahmen abzufragen – der Programmierfehler muss behoben werden. Ein Programmierer löst eine `NullPointerException` selten selbst aus, sondern das macht die JVM automatisch. Sie kann jedoch vom Entwickler bewusst ausgelöst werden, wenn eine zusätzliche Klarheit verschaffen soll, oder früh beim Prüfen von Parametern am Kopf einer Methode.

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 eine `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` ohne zusätzliche Fehlermeldung selten ist, ist dieses Idiom nicht wirklich nützlich.



### 8.5.3 Parameter testen und gute Fehlermeldungen

Eine `IllegalArgumentException` ist eine wertvolle Ausnahme, die eine interne Ausnahme anzeigen: dass nämlich eine Methode mit falschen Argumenten aufgerufen wurde. Eine Methode sollte im Idealfall alle Parameter auf ihren korrekten Wertebereich hin prüfen und nach dem *Fail-fast-Verfahren* arbeiten, also so schnell wie möglich eine Ausnahme melden, anstatt Fehler zu ignorieren oder zu verschleppen. Wenn etwa das Alter einer Person bei `setAge(int)` 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 Teil-Strings falsch oder ist der Index für einen Array-Zugriff zu groß, hagelt es eine `StringIndexOutOfBoundsException` bzw. `ArrayIndexOutOfBoundsException`:

```
System.out.println( "Orakel-Paul".substring( 0, 20 ) ); // ☠
```

liefert:

```
java.lang.StringIndexOutOfBoundsException: begin 0, end 20, length 11
```

Und

```
System.out.println( "Orakel-Paul".toCharArray()[20] ); // ☠
```

liefert:

```
java.lang.ArrayIndexOutOfBoundsException: Index 20 out of bounds for length 11
```

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.<sup>6</sup> Die Java Standardbibliothek bietet drei `checkXXX(...)`-Methoden in der Klasse `Objects`, siehe [Kapitel 10, »Besondere Typen der Java SE«](#).

### null-Prüfungen

Für null-Prüfungen bietet sich die Methode `Objects.requireNonNull(reference)` an, die immer dann eine `NullPointerException` auslöst, wenn `reference` gleich `null` ist. Optional als zweites Argument lässt sich die Fehlermeldung angeben.

### Tool-Unterstützung

Ein anderer Ansatz sind Prüfungen durch externe Codeprüfungsprogramme. Google zum Beispiel setzt in seinen vielen Java-Bibliotheken auf Parameter-Annotationen wie `@Nonnull` oder `@Nullable`.<sup>7</sup> 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.

---

<sup>6</sup> 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 (<https://commons.apache.org/proper/commons-lang>) oder mit `com.google.common.base.Preconditions` von Google Guava (<https://github.com/google/guava>) Vergleichbares bietet; in jedem Fall ist eine gute Parameterprüfung bei öffentlichen Methoden von Bibliotheken ein Muss.

<sup>7</sup> Sie wurden in JSR-305, »Annotations for Software Defect Detection«, definiert. Java 7 sollte dies ursprünglich unterstützen, doch das wurde gestrichen.

#### 8.5.4 Neue Exception-Klassen deklarieren

Eigene Ausnahmen 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 parameterlosen Konstruktor und einen mit einem String parametrisierten Konstruktor, um eine Fehlermeldung anzunehmen und zu speichern.

Um für die Klasse `Player` im letzten Beispiel einen neuen Ausnahmetyp zu deklarieren, erweitern wir `RuntimeException` zur `PlayerException`:

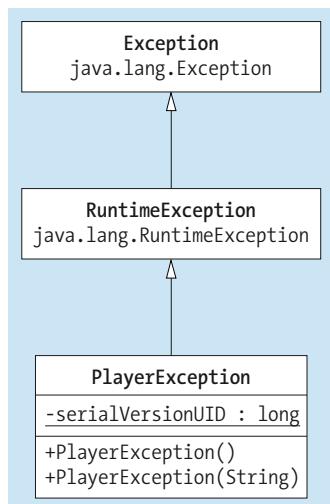


Abbildung 8.12 UML-Diagramm für `PlayerException`

**Listing 8.19** src/main/java/com/tutego/insel/exception/v2/PlayerException.java

```

package com.tutego.insel.exception.v2;

public class PlayerException extends RuntimeException {

    public PlayerException() { }

    public PlayerException( String s ) {
        super( s );
    }
}
  
```

Nehmen wir uns die Initialisierung mit dem Alter noch einmal vor. Statt der `IllegalArgumentException` löst der Konstruktor im Fehlerfall unsere speziellere `PlayerException` aus:

**Listing 8.20** src/main/java/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" c.t.i.e.v2.PlayerException: Kein Alter <= 0 erlaubt!
at com.tutego.insel.exception.v2.Player.<init>(Player.java:9)
at com.tutego.insel.exception.v2.Player.main(Player.java:16)
```

**Tipp**

Es ist immer eine gute Idee, Unterklassen von `Exception` zu bauen. Würden wir keine Unterklassen anlegen, sondern direkt mit `throw new Exception()` eine Ausnahme anzeigen, so könnten wir diese Ausnahme später nicht mehr von anderen Ausnahmen unterscheiden. Mit der Hierarchiebildung werden nämlich die Spezialisierung bei mehreren `catch`-Klauseln sowie eine Unterscheidung mit `instanceof` unterstützt. Sind die Ausnahmen exakt vom Typ `Exception`, müssten wir unsere Ausnahmen immer mit `catch(Exception e)` auffangen und bekämen so alle anderen Ausnahmen mit aufgefangen, die dann nicht mehr unterschieden werden könnten. Allerdings sollten Entwickler nicht zu inflationär mit den Ausnahmenhierarchien umgehen; in vielen Fällen reicht eine Standardausnahme aus.

### 8.5.5 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 Ausnahmen dieser Art Programmierfehler sind und behoben werden müssen. Ein `catch` wäre unnötig, da diese Ausnahme ja im korrekten Code gar nicht auftreten kann. 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 Ausnahmen von sich aus auslöst und nicht eine Methode.

- ▶ Soll sich die Anwendung von der Ausnahme »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 *Java EE* oder *Spring* fast ausschließlich auf `RuntimeException` setzen: Wenn es einen Fehler gibt, dann lässt sich schwer etwas behandeln und einfach korrigieren. Zeigt etwa ein internes Modul beim Datenbankzugriff eine Ausnahme an, muss die ganze Operation abgebrochen werden, und nichts ist zu retten. Hier gilt im Großen, was auch bei der `NullPointerException` im Kleinen passiert: Die Ausnahme 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.<sup>8</sup> Hätte das Programm mit der `Files`-Methode `exists(...)` vorher nach der Existenz der Datei gefragt, wäre uns diese Ausnahme erspart geblieben. (Der Sonderfall, dass die Datei beim Test noch da war, dann aber im Hintergrund gelöscht wird, ist auch nicht zu vernachlässigen.) 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 Voraussetzungen 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 Methode `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

---

<sup>8</sup> Eine Reaktion wie »File not found. Should I fake it? (Y/N)« ist auch nicht so clever.

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`- oder `throws`-Klauseln abgeändert werden müssen. `RuntimeExceptions` sind hier flexibler. Ändert sich bei einer agilen Programmierung der Typ von geprüften Ausnahmen im Lebenslauf einer Software öfters, führt das zu vielen Änderungen, 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 Schlafendelenken 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.<sup>9</sup>

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. Die Ausnahme 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, so dass heute eine große Unsicherheit über die richtige `Exception`-Basisklasse besteht. Zwar zwingt uns der Compiler, eine geprüfte Ausnahme 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 Beobachter und ist nur noch eine Konvention. Mit dieser Alles-ist-ungeprüft-Version würde dann Java gleichauf mit C#, C++, Python, Groovy ... liegen.<sup>10</sup>

---

<sup>9</sup> Luxuriös wäre eine Prüfung zur Compilezeit, denn wenn etwa `new URL("http://de.tutego")` 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.

<sup>10</sup> Doch eines ist sicher: Java-Vater James Gosling ist dagegen: <http://www.artima.com/intv/solid.html>

### Besondere Rückgaben oder Ausnahmen?

Nicht immer ist eine Ausnahme nötig, doch wann es eine Rückgabe wie `null` oder »–« gibt und wann eine Ausnahme ausgelöst werden soll, ist nicht immer einfach zu beantworten und hängt vom Kontext ab. Ein Beispiel: Eine Methode liest eine Datei ein und führt eine Suche durch. Wenn eine bestimmte Teilzeichenkette nicht vorhanden ist, soll die Methode dann eine Ausnahme werfen oder nicht? Hier kommt es darauf an:

1. Wenn das Dokument in der ersten Zeile eine Kennung tragen muss und der Test auf diese Kennung prüft, dann liegt ein Protokollfehler vor, wenn diese Kennung nicht vorhanden ist.
2. Im Dokument gibt es eine einfache Textsuche. Ein Suchwort kann enthalten sein, muss aber nicht.

Im ersten Fall passt eine Ausnahme gut, da ein interner Fehler vorliegt. Muss die Kennung in der Datei sein, ist sie es aber nicht, darf dieser Fehler nicht untergehen, und eine Ausnahme zeigt das perfekt an. Ob geprüft oder ungeprüft, steht auf einem anderen Blatt. Im zweiten Fall ist eine Ausnahme unangebracht, da es kein Fehler ist, wenn der Such-String nicht im Dokument ist; das kann vorkommen. Das ist das Gleiche wie bei `indexOf(...)` oder `matches(...)` von `String` – die Methoden würden ja auch keine Ausnahmen werfen, wenn es keine Übereinstimmung gibt.

#### 8.5.6 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(String)` setzt vor einen Hostnamen "http://" und liefert das Ergebnis als `URI`-Objekt zurück. `createUriFromHost("tutego.de")` liefert somit einen `URI` mit `http://tutego.de`. Ist der Hostname aber falsch, löst der Konstruktor der `URI`-Klasse eine Ausnahme aus:

**Listing 8.21** src/main/java/com/tutego/insel/exception/Rethrow.java, Ausschnitt

```
public class Rethrow {

    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 diese Ausnahme provozieren wir, indem wir dem Konstruktor ein "http://%" übergeben, was ein offensichtlich falscher `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.base/java.net.URI$Parser.fail(URI.java:2915)
...
    at java.base/java.net.URI.<init>(URI.java:600)
    at com.tutego.insel.exception.Rethrow.createUriFromHost(Rethrow.java:9)
    at com.tutego.insel.exception.Rethrow.main(Rethrow.java:20)
```

## 8.5.7 Aufruf-Stack 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 Aufruf-Stack weitergibt. Aus dem vorangehenden Beispiel:

```
Hilfe! Malformed escape pair at index 7: http://%
java.net.URISyntaxException: Malformed escape pair at index 7: http://%
at java.base/java.net.URI$Parser.fail(URI.java:2915)
```

```

at java.base/java.net.URI.<init>(URI.java:600)
at com.tutego.insel.exception.Rethrow.createUriFromHost(Rethrow.java:9)
at com.tutego.insel.exception.Rethrow.main(Rethrow.java:20)

```

Die `main(...)`-Methode fängt die Ausnahme von `createUriFromHost(...)` ab, aber diese Methode steht nicht ganz oben im Aufruf-Stack. 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 Aufruf-Stack neu füllen lässt. Unsere bekannte Methode `createUriFromHost(...)` soll auf `fillInStackTrace()` zurückgreifen:

**Listing 8.22** src/main/java/com/tutego/insel/exception/RethrowFillInStackTrace.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 Aufruf-Stack mit der `fail(...)`-Methode ganz oben gespeichert, allerdings löscht `fillInStackTrace()` zunächst den ganzen Stack-Trace 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 com....RethrowFillInStackTrace.createUriFromHost(RethrowFillInStackTrace.java:12)
at com....RethrowFillInStackTrace.main(RethrowFillInStackTrace.java:20)

```

### 8.5.8 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 Ausnahmen eines gewissen Typs aufzufangen. Wir können auch so weit in der Ausnahmehierarchie nach oben laufen, um alle Ausnahmen 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 nicht immer klar, was als Ausnahme 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 einer Ausnahme, 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 sie Ausnahmen an den Aufrufer weitergibt:

**Listing 8.23** src/main/java/com/tutego/insel/exception/RethrowTypes.java, saveScreenshot()

```
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 ) {
        Files.delete( Paths.get( filename ) );
        throw e;
    }
}
```

Mit den beiden `catch`-Blöcken sind wir genau auf die Ausnahmen eingegangen, die `createScreenCapture(Rectangle)` 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 ) {
    // Datei löschen
    throw e;
}
```

schreiben wir:

```
catch ( Throwable e ) {
    // Datei löschen
    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:

**Listing 8.24** src/main/java/com/tutego/insel/exception/RethrowTypes2.java, Ausschnitt

```
public static void saveScreenshot( String filename )
    throws AWTException /*1*/, IOException /*2*/ {
    ...
    catch ( AWTException /*1*/ e ) {
        throw e;
    }
    catch ( Throwable /*2*/ e ) {
        Files.delete( Paths.get( filename ) );
        throw e;
    }
}
```

Die erste `catch`-Klausel fängt die `AWTException` und leitet sie weiter. Damit wird `saveScreenshot(String)` zum möglichen Auslöser von `AWTException`, und die Ausnahme muss mit `throws` an die Signatur. Wenn nun ein `catch`-Block jedes `Throwable` auffängt und dieses `Throwable` 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 von einer unpräziseren hin zur präziseren Typanalyse.

### Präzisere Typprüfung

Der Java-Compiler führt bei Ausnahmen eine präzise Typanalyse durch: 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 des gefangenen Typs 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 nichtfinalen `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 ) {
    // Datei löschen
    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. Es ist daher 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 Code<sup>11</sup> von unterschiedlichen Projekten keine Probleme gefunden. Prinzipiell könnte der Compiler jetzt unerreichbaren Code finden, der vorher versteckt blieb. Ein kleines Beispiel, das vor Java 7 compiliert, 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. Vor Java 7 war das etwas anders, denn

---

<sup>11</sup> Die Zahl stammt aus der FOSDEM-Präsentation 2011 »Project Coin: Language Evolution in the Open«.

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.)

### 8.5.9 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 geht das 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:

```
class java.lang.Throwable
    implements Serializable
```

- `Throwable(Throwable cause)`
- `Throwable(String message, Throwable cause)`

Den Grund der Ausnahme erfragt die Methode `Throwable.getCause()`.

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:

```
class java.lang.Exception
    extends Throwable
```

- `Exception()`
- `Exception(String message)`
- `Exception(String message, Throwable cause)`
- `Exception(Throwable cause)`

Einige der tiefer liegenden Unterklassen haben dann auch diese Konstruktortypen mit `Throwable`-Parameter, wie `IOException`, `SQLException` oder `ClassNotFoundException`, andere wiederum nicht, wie `PrinterException`. Eigene Unterklassen können auch mit `initCause`(`Throwable`) 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 Webseite 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 (`MalformedURLException` ist eine spezielle `IOException`, das verkürzt das Programm etwas). 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 8.25** src/main/java/com/tutego/insel/exception/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 ( IOException e ) {
        throw new RuntimeException( e );
    }
}

public static void main( String[] args ) {
    System.out.println( isAvailable( "http://laber.rhabar.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:

```
Exception in thread "main" java.lang.RuntimeException: java.net.UnknownHostException: laber.rabar.ber
  at com.tutego.insel.exception.NestedException.isAvailable(NestedException.java:15)
  at com.tutego.insel.exception.NestedException.main(NestedException.java:23)
Caused by: java.net.UnknownHostException: laber.rabar.ber
  at java.base/java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:220)
...
  at java.base/java.net.HttpURLConnection.getResponseCode(HttpURLConnection.java:527)
  at com.tutego.insel.exception.NestedException.isAvailable(NestedException.java:12)
  ... 1 more
```

In der Praxis ist es bei großen Stack-Traces – und einem Szenario, bei dem abgefangen und neu 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.

### Hinweis

Statt den parametrisierten Konstruktor `new RuntimeException(e)` zu verwenden, hätten wir auch `initCause(...)` verwenden können:

```
catch ( IOException e ) {
    RuntimeException e2 = new RuntimeException();
    e2.initCause( e );
    throw e2;
}
```

Allerdings gibt es einen wichtigen Unterschied: Ohne den parametrisierten Konstruktor und mit `initCause(...)` gibt es keinen automatischen Aufruf von `fillInStackTrace()`, daher sieht der Stack-Trace unterschiedlich aus.



### UncheckedIOException

Ausnahmen bei Ein-/Ausgabe-Operationen werden in Java traditionell über eine geprüfte Ausnahme vom Typ `IOException` gemeldet. Bei Frameworks ist das zum Teil etwas lästig, so dass es im Paket `java.io` eine Art Wrapper-Klasse gibt, die eine geprüfte `IOException` in einer ungeprüften `UncheckedIOException` mantelt.

```
class java.io.UncheckedIOException
extends RuntimeException
```

- `UncheckedIOException(IOException cause)`  
Ummantelt `cause`.
- `UncheckedIOException(String message, IOException cause)`  
Ummantelt `cause` mit einer zusätzlichen Meldung.

Bisher macht die Java-Bibliothek nur an einer Stelle von diesem Ausnahmetyp Gebrauch, und das ist bei `lines()` der Klasse `BufferedReader`, damit bei der Stream-API die geprüften Ausnahmen nicht im Weg stehen.

## 8.6 Automatisches Ressourcen-Management (try mit Ressourcen)

Java hat eine automatische Speicherbereinigung, die nicht mehr referenzierte Objekte erkennt und ihren Speicher automatisch freigibt. Nun bezieht sich der Garbage-Collector aber ausschließlich auf Speicher, doch es gibt viele weitere Ressourcen:

- ▶ Dateisystemressourcen von Dateien
- ▶ Netzwerkressourcen wie Socket-Verbindungen
- ▶ Datenbankverbindungen
- ▶ nativ gebundene Ressourcen des Grafiksubsystems
- ▶ 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. Soll eine Variable in `finally` zugänglich sein, muss sie außerhalb des `try`-Blocks deklariert werden, was ihr eine längere 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.

### 8.6.1 try mit Ressourcen

Um das Schließen von Ressourcen zu vereinfachen, gibt es eine besondere Form der `try`-Anweisung, die *try mit Ressourcen* heißt. Mit diesem Sprachkonstrukt lassen sich *Ressourcen-*

*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 von Tabelle 8.3 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" );</pre> <pre>try ( Scanner res = new Scanner( in ) ) {</pre> <pre>    System.out.println( res.nextLine() );</pre> <pre>}</pre>	<pre>{</pre> <pre>    Scanner res = new Scanner( in );</pre> <pre>    try {</pre> <pre>        System.out.println( res.nextLine() );</pre> <pre>    }</pre> <pre>    finally {</pre> <pre>        res.close();</pre> <pre>    }</pre> <pre>}</pre>

**Tabelle 8.3** Vereinfache Umsetzung von `try` mit Ressourcen

Bisher haben wir nach dem Schlüsselwort `try` direkt einen Block gesetzt, doch `try` mit Ressourcen nutzt eine eigene erweiterte Syntax:

1. Nach dem `try` folgt statt des direkten `{}`-Blocks eine Ressourcenspezifikation in runden Klammern und dann erst der `{}`-Block, also `try (...) {...}` statt `try {...}`.
2. Die Ressourcenspezifikation besteht aus einem Satz runder Klammern und einer Liste von Ressourcen. Die Ressourcen sind finale Variablen vom Typ `AutoCloseable`, und genau diese werden später automatisch geschlossen. Die Variablen können mit einem Gleichheitszeichen direkt mit einem Ausdruck initialisiert werden, etwa einem Konstruktor- oder Methodenaufruf; vor Java 9 war die Initialisierung zwingend. Der Einsatz von `var` ist erlaubt.

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 einer Ausnahme kam. Der Compiler fügt alle nötigen Prüfungen ein.



### Hinweis

Wird auf einer Variablen, die vom Typ AutoCloseable ist, nicht `close()` aufgerufen oder wird sie nicht in einem `try` mit Ressourcen eingesetzt, gibt der Java-Compiler eine Warnung. Eclipse etwa meldet: »Resource leak: '<unassigned Closeable value>' is never closed«. Die Warnung verschwindet mit `@SuppressWarnings("resource")`.

### 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()`, das `try` mit Ressourcen automatisch aufruft. Anders ist es, wenn die Resource ein klassischer Datenstrom (`InputStream/OutputStream/Reader/Writer`) ist, denn dort deklariert die `close()`-Methode eine `IOException`. Die muss daher auch behandelt werden, wie es das folgende Beispiel zeigt:

**Listing 8.26** TryWithResourcesReadsLine, readFirstLine()

```
static String readFirstLine( String file ) {
    try ( BufferedReader br = Files.newBufferedReader(
        Paths.get( file ), StandardCharsets.ISO_8859_1 ) ) {
        return br.readLine();
    }
    catch ( IOException e ) { e.printStackTrace(); return null; }
}
```

Wenn `try` mit Ressourcen verwendet wird, bleibt die deklarierte Ausnahme bei `close()` bestehen; es zaubert die in der Regel ausgelöste `IOException` nicht weg, und entweder muss ein fangendes `catch` her oder die Ausnahme weitergeleitet werden. Daraus ergibt sich ein interessanter Nebeneffekt: Hat ein Aufruf von einem `IOException`-auslösenden `close()` im Code gefehlt und wird Programmcode in der `try`-mit-Ressourcen-Syntax umgeschrieben, so führt der vom Compiler automatisch eingesetzte `close()`-Aufruf zu einem Compilerfehler, wenn die geprüfte `IOException` nicht behandelt wird.



### 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 die nicht behandelte Ausnahme von close() führt zu »exception thrown from implicit call to close() on resource variable 'r'«.

## 8.6.2 Die Schnittstelle AutoCloseable

Das try mit Ressourcen schließt Ressourcen vom Typ AutoCloseable. Daher wird es Zeit, sich diese Schnittstelle etwas genauer anzuschauen:

**Listing 8.27** java/lang/AutoCloseable.java

```
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
java.io.Scanner	close() // ohne Ausnahme
javax.sound.sampled.Line	close() // ohne Ausnahme
java.io.Writer	close() throws IOException
java.sql.Connection	close() throws SQLException

**Tabelle 8.4** Einige Typen, die AutoCloseable implementieren bzw. erweitern

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 Ressourcenfreigabe unpassend ist, wenn etwa ein Grafikobjekt bei der Freigabe eine IOException auslöst. Vielmehr ist der Weg

andersherum: 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 freigegeben werden könnte, oder Grafikanwendungen, bei denen der Grafikkontext wieder freigegeben werden muss, gehören nicht dazu.



### Hinweis

Es gibt Ströme, die müssen offen bleiben. Das gilt etwa für System.in, einen vom System bereitgestellten InputStream. Auch wenn dieser in einem Scanner verpackt wird, wird ein close() auf dem Scanner zu einem close() auf dem InputStream, und ein erneutes Lesen aus System.in wird mit einem »java.io.IOException: Stream closed« quittiert. Ein try ( Scanner in = new Scanner(System.in) ) { ... } ist also keine gute Idee.



### Tipp \*

Es ist mit einem Trick möglich, auch Exemplare in einem try mit Ressourcen zu nutzen, die nicht vom Typ AutoCloseable sind. Ein Lambda-Ausdruck oder eine Methodenreferenz lassen sich nutzen, um eine beliebige Methode als close()-Methode einzusetzen. Ein ReentrantLock zum Beispiel ist eine Implementierung eines Lock, um bei nebenläufigen Zugriffen einen Bereich abzuschließen. lock() beginnt den Bereich, unlock() gibt ihn wieder frei. Das unlock() lässt sich über einen Lambda-Ausdruck als close()-Methode verkaufen.

```
ReentrantLock lock = new ReentrantLock();
try ( AutoCloseable unlock = lock::unlock ) { // oder () -> {lock.unlock();}
    lock.lock();
}
System.out.println( lock.isLocked() ); // false
```

Ob dieser »Trick« sinnvoll ist oder nicht, ist eine andere Frage. Das try mit Ressourcen setzt auf jeden Fall das unlock() in einen internen finally-Block, der über die Konstruktion eingespart wird. Allerdings wird üblicherweise die Ressource im try-mit-Ressourcen-Block auch erst deklariert, was hier vorher gemacht werden muss, außerdem ist die Variable unlock unnütz. Daher ist die Relevanz eher niedrig.

### 8.6.3 Mehrere Ressourcen nutzen

Unsere beiden Beispiele zeigen die Nutzung eines Ressourcentyps. Es sind aber auch mehrere Typen möglich, die ein Semikolon trennt:

```
try ( InputStream in = Files.newInputStream( srcPath );
      OutputStream out = Files.newOutputStream( destPath ) ) {
    ...
}
```

#### Hinweis

Die Trennung erledigt ein Semikolon, und jedes Segment kann einen unterschiedlichen Typ deklarieren, etwa InputStream/OutputStream. Die Ressourcentypen 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 einer Ausnahme 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 Array-Initialisierungen zum Schluss ein Komma stehen kann:

```
int[] array = { 1, 2, };
//           ^ Komma optional
try ( InputStream in = Files.newInputStream( path ); ) { ... }
//           ^ Semikolon optional
```

```
try ( InputStream in = Files.newInputStream( src );
      OutputStream out = Files.newOutputStream( dest ); ) { ... }
// ^ Semikolon optional
```

Ob das stilvoll ist, muss jeder selbst entscheiden; in der Insel steht kein unnützes Zeichen.

#### 8.6.4 try mit Ressourcen auf null-Ressourcen

Dass immer zum Abschluss eines try-mit-Ressourcen-Blocks ein `close()` aufgerufen wird, ist nicht ganz korrekt; es gibt nur dann einen Schließversuch, wenn die Ressource ungleich `null` ist.



#### Beispiel

Der Codebaustein compiliert und führt zu einer Konsolenausgabe:

```
try ( Scanner scanner1 = null; Scanner scanner2 = null ) {
    System.out.println( "Ok" );
}
```

Bei Konstruktoren ist ein Objekt ja immer gegeben, aber es gibt auch Fabriaufrufe, bei denen vielleicht `null` herauskommen kann, und für diese Fälle ist es ganz praktisch, dass `try` mit Ressourcen dann nichts macht, um eine `NullPointerException` beim `close()` zu vermeiden.

#### 8.6.5 Unterdrückte Ausnahmen \*

Aufmerksame Leser haben bestimmt schon ein Detail wahrgenommen: Im Text steht »ver einfachte 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 allein 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 gleich 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.<sup>12</sup> Wie also beide Ausnahmen melden? Hier gibt es einen Trick: Da die Ausnahme im try-Block wichtiger ist, ist sie die »Hauptausnahme«, und die close()-Ausnahme kommt Huckepack als Extra-Information mit obendrauf.

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 8.28** src/main/java/com/tutego/insel/exception/NotCloseable.java, Ausschnitt

```
public class NotCloseable implements AutoCloseable {
    @Override public void close() {
        throw new UnsupportedOperationException( "close() mag ich nicht" ); // ☠
    }
}
```

Zum Beispiel selbst:

**Listing 8.29** src/main/java/com/tutego/insel/exception/SuppressedClosed.java, Ausschnitt

```
public class SuppressedClosed {
    public static void main( String[] args ) {
        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 com.tutego.insel.exception.SuppressedClosed.main(SuppressedClosed.java:6)
Suppressed: java.lang.UnsupportedOperationException: close() mag ich nicht
at com.tutego.insel.exception.NotCloseable.close(NotCloseable.java:4)
at com.tutego.insel.exception.SuppressedClosed.main(SuppressedClosed.java:7)
```

---

<sup>12</sup> In einem frühen Prototyp war dies tatsächlich der Fall – die Ausnahme wurde komplett geschluckt.

Die Hauptausnahme ist die `NullPointerException`. Die interessante Zeile beginnt mit `Suppressed:`, denn dort ist die `close()`-Ausnahme referenziert. An den Aufrufer kommt die spannende Ausnahme des misslungenen `try`-Blocks 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 com.tutego.insel.exception.NotCloseable.close(NotCloseable.java:4)
at com.tutego.insel.exception.SupportedClosed.main(SupportedClosed.java:7)
```

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:

**Listing 8.30** src/main/java/com/tutego/insel/exception/SupportedClosed2.java, Ausschnitt

```
try ( NotCloseable res1 = new NotCloseable();
      NotCloseable res2 = new NotCloseable() ) {
    throw new NullPointerException();
}
```

Aufgerufen führt dies zu:

```
Exception in thread "main" java.lang.NullPointerException
at com.tutego.insel.exception.SupportedClosed2.main(SupportedClosed2.java:7)
Suppressed: java.lang.UnsupportedOperationException: close() mag ich nicht
at com.tutego.insel.exception.NotCloseable.close(NotCloseable.java:4)
at com.tutego.insel.exception.SupportedClosed2.main(SupportedClosed2.java:8)
Suppressed: java.lang.UnsupportedOperationException: close() mag ich nicht
at com.tutego.insel.exception.NotCloseable.close(NotCloseable.java:4)
at com.tutego.insel.exception.SupportedClosed2.main(SupportedClosed2.java:8)
```

Jede unterdrückte `close()`-Ausnahme taucht auf.

### Umsetzung

In [Abschnitt 8.1.8](#), »Abschlussbehandlung mit 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, gibt es in der Basisklasse `Throwable` zwei besondere Methoden:

```
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 geschluckt werden. Wenn, dann zeigt sie auch `printStackTrace(...)` nicht mehr an.

#### Blick über den Tellerrand

In C++ gibt es Destruktoren, 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 Blocks `__enter__()` aufruft und beim Verlassen die Methode `__exit__()`. Das ist insofern interessant, als hier zwei Methoden zur Verfügung stehen. Bei Java ist es nur `close()` beim Verlassen des Blocks, aber es gibt keine Methode zum Betreten eines Blocks; so etwas muss beim Anlegen der Ressource erledigt werden.

## 8.7 Besonderheiten bei der Ausnahmebehandlung \*

Bei der Ausnahmebehandlung gibt es ein paar Überraschungen, die vier Abschnitte gesondert vorstellen.

### 8.7.1 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:

```
class Windows10KeyGenerator {
    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. Kann der Compiler sehen, dass eine Methode eine Ausnahme auslöst und die return-Anweisung nicht erreichbar ist, dann ist alles hinter dem throw nicht erreichbar, und es dürfen keine weiteren Anweisungen folgen.

generateKey() muss diese Exception nicht mit throws ankündigen, da UnsupportedOperationException eine RuntimeException ist.

### 8.7.2 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 hillaryVsDonald() {
    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.



### 8.7.3 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 8.31** src/main/java/com/tutego/insel/exception/SubRandomAccessFile.java, Ausschnitt

```

public class SubRandomAccessFile extends RandomAccessFile {
    public SubRandomAccessFile( 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();
        }
    }
    @Override
    public void close() {
    }
}

```

Die Methoden `length()`, `write(..)` und `close()` lösen in `RandomAccessFile` eine `IOException` aus. Unsere Unterklasse `SubRandomAccessFile` überschreibt `length()` und lässt die Ausnahme in der Signatur weg. Das hat in der Nutzung einige Folgen, denn wenn wir die Klasse als `SubRandomAccessFile` der Art

```

SubRandomAccessFile raf = ...
raf.length();

```

verwenden, muss bei `length()` keine Ausnahme mehr 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 wegzulassen.

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`.

### Design

Wenn demnach eine überschriebene Methode der Unterklasse keine geprüften Ausnahmen hinzufügen kann, muss das Design der Basistypen so entworfen sein, dass Unterklassen notwendige Ausnahmen melden können.

### Hinweis

Implementiert eine Unterklasse einen eigenen Konstruktor und ruft dieser `super(...)` für einen Konstruktor auf, der eine Ausnahme auslöst, so muss auch der Konstruktor der Unterklasse diese Ausnahme melden, denn der neue Konstruktor kann die Ausnahme nicht auffangen. In unserem Beispiel wäre also illegal:

```
public SubRandomAccessFile( File file, String mode ) {
    try {
        super( file, mode );
    } catch ( Exception e ) { }
}
```

Der Grund ist ganz einfach: Wenn der Konstruktor der Oberklasse eine Ausnahme auslöst, ist das Objekt nicht vollständig initialisiert. Und wenn der Konstruktor der Unterklasse dann die Ausnahme abfängt, würde ja die Unterklasse vielleicht nicht vollständig initialisierte Eigenschaften der Oberklasse erben, also ein halbgares Objekt. Das ist unerwünscht.

#### 8.7.4 Nicht erreichbare catch-Klauseln

Löst in einem `try`-Block eine Anweisung eine Ausnahme aus und gibt es dafür eine `catch`-Klausel, so heißt die `catch`-Klausel *erreichbar*. Zusätzlich darf vor dieser `catch`-Klausel natür-

lich kein anderes catch stehen, das diese Ausnahme mit abfängt. Wenn wir zum Beispiel catch(Exception e) als erstes Auffangbecken bereitstellen, behandelt das natürlich alle Ausnahmen. Die Konsequenz daraus: catch-Klauseln müssen immer von den speziellen zu den allgemeinen Ausnahmearten sortiert werden (alles andere würde der Compiler auch verhindern).

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 Ausnahmen in ihrer throws-Klausel angibt:

**Listing 8.32** src/main/java/com/tutego/insel/exception/TooManyExceptions.java, openFile()

```
void openFile() throws FileNotFoundException,
                      IOException,
                      InterruptedException {
    try ( RandomAccessFile r = 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 nicht 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 8.33** src/main/java/com/tutego/insel/exception/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.

## 8.8 Assertions \*

Die Übersetzung des englischen Wortes *assertion* lässt vermuten, worum es geht: um *Zusicherungen*. Assertions formulieren Aussagen, die beim korrekten Ablauf des Codes immer wahr sein müssen. Ist eine Bedingung nicht erfüllt, folgt eine Ausnahme, die darauf hinweist, dass im Programm etwas falsch gelaufen sein muss. Der Einsatz von Assertions im Code fördert die Dokumentation für den gültigen Programmzustand. Wir unterscheiden:

- ▶ Precondition: Zustand, der vor einer Operation immer wahr sein muss
- ▶ Postcondition: Zustand, der nach einer Operation immer wahr sein muss

Die Formulierung korrekter Zustände ist ein wesentliches Element von *Design by Contract*, einer Entwicklungsmethode, bei der es darum geht, einen »Vertrag« (engl. *contract*) aufzu stellen über das, was ein Programm leisten muss. Bertrand Meyer, auch Erfinder der Programmiersprache Eiffel, prägte diesen Begriff.

### 8.8.1 Assertions in eigenen Programmen nutzen

Java-Programme nutzen für Assertions im Quellcode die `assert`-Anweisung. Es gibt zwei Varianten, eine mit und eine ohne Meldung:

```
assert AssertConditionExpression;
assert AssertConditionExpression : MessageExpression;
```

`AssertConditionExpression` steht für ein Prädikat, das zur Laufzeit ausgewertet wird. Der Ausdruck wird nicht automatisch geklammert, da er sich nicht wie ein Methodenaufruf liest.

#### Java-Geschichte

Neue Schlüsselwörter wurden immer wieder eingeführt: in Java 1.3 das Schlüsselwort `strictfp`, in Java 1.4 das Schlüsselwort `assert` für die »Behauptungen«, und in Java 5 Aufzählungstypen mit dem neuen Schlüsselwort `enum`.

### 8.8.2 Assertions aktivieren und Laufzeit-Errors

Assertions stehen immer in der Klassendatei, da sie der Compiler immer in Bytecode abbildet. Jedoch werden Assertions zur Laufzeit 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.

Sind Assertions aktiviert und wertet die JVM das Ergebnis der assert-Anweisungen zu `true` aus, führt die Laufzeitumgebung die Abarbeitung normal weiter; ergibt die Auswertung `false`, wird das Programm mit einem `java.lang.AssertionError` beendet. Wir haben gesehen, dass es zwei Varianten gibt:

```
assert AssertConditionExpression;
assert AssertConditionExpression : MessageExpression;
```

Der optionale zweite Parameter, `MessageExpression`, ist ein Text, der beim Stack-Trace als Nachricht in der Fehlermeldung erscheint. Die in Java ausgelösten Ausnahmen sind vom Typ »Error« und nicht vom Typ »Exception« und sollten daher auch nicht aufgefangen werden, da eine nicht erfüllte Bedingung ein Programmierfehler ist.



### Hinweis

Die JVM ignoriert Assertions standardmäßig bei der Ausführung, und eine Aktivierung erfolgt nur auf Befehl; ein Ablauf ohne Bedingungstests ist eher der Normalfall. Daraus folgt, dass Ausdrücke in den assert-Anweisungen ohne Nebeneffekte sein müssen. So etwas wie

```
assert counter-- == 0;
```

ist keine gute Idee, denn das Vermindern der Variablen ist ein Nebeneffekt, der nur dann stattfindet, wenn die JVM auch Assertions aktiviert hat. Allerdings lässt sich das auch für einen Trick nutzen, Assertions bei der Ausführung zu erzwingen. Im statischen Initialisierer einer Klasse können wir setzen:

```
boolean assertEnabled = false;
assert assertEnabled = true;
if ( ! assertEnabled )
    throw new RuntimeException( "Assertions müssen aktiviert werden" );
```

### 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 ist das Ergebnis ein `NaN`. Sollte irgendein Programmteil nun die Methode `subAndSqrt(double, double)` 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 8.34** src/main/java/com/tutego/insel/assertion/AssertKeyword.java, Ausschnitt

```
public class AssertKeyword {

    public static double subAndSqrt( double a, double b ) {
        double result = Math.sqrt( a - b );

        assert ! Double.isNaN( result ) : "Berechnungsergebnis ist NaN!";

        return result;
    }

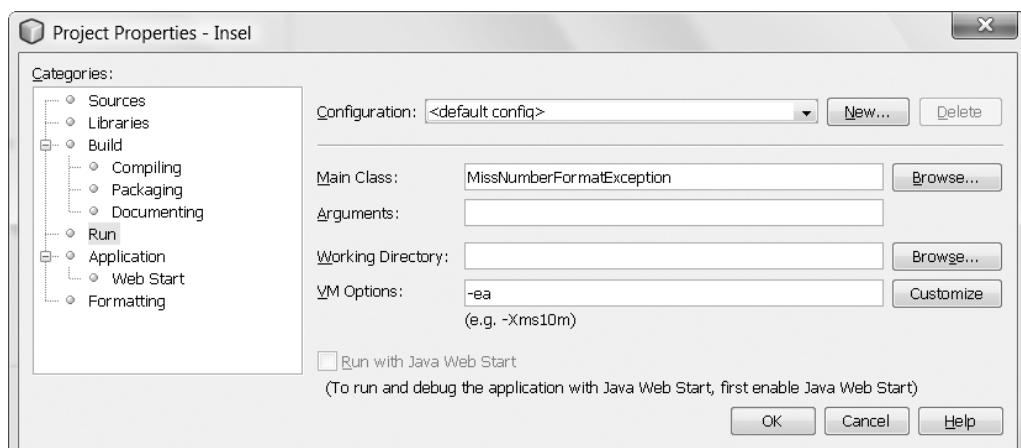
    public static void main( String[] args ) {
        System.out.println( "Sqrt(10-2)=" + subAndSqrt(10, 2) );
        System.out.println( "Sqrt(2-10)=" + subAndSqrt(2, 10) );
    }
}
```

Rufen wir das Programm mit dem Schalter -ea auf:

```
$ java -ea com.tutego.insel.assertion.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:8)
at com.tutego.insel.assertion.AssertKeyword.main(AssertKeyword.java:15)
```



**Abbildung 8.13** Unter »File • Project Properties« kann der VM-Schalter für die Assertions gesetzt werden.

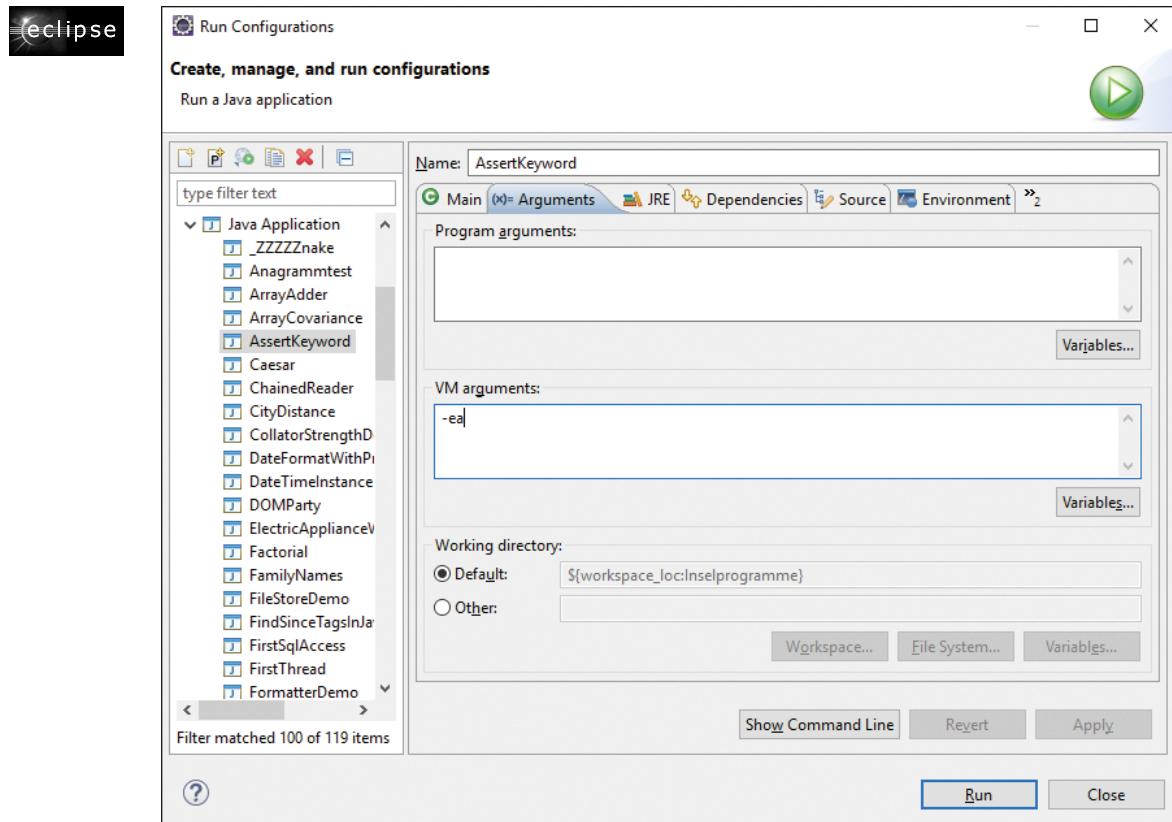


Abbildung 8.14 Wurde das Programm in Eclipse schon gestartet, kann im Menü »Run • Run Configurations ...« unterhalb des Reiters »Arguments« bei den »VM arguments« der Schalter »-ea« gesetzt werden.

### 8.8.3 Assertions feiner aktivieren oder deaktivieren

Assertions müssen nicht global für das ganze Programm gesetzt werden, sondern können auch feiner deklariert werden, etwa für eine Klasse oder ein Paket. Mit geschickter Variation von `-ea` (Assertions aktivieren) und `-da` (Assertions deaktivieren) lässt sich sehr gut steuern, was die Laufzeitumgebung prüfen soll.



#### Beispiel

Aktiviere Assertions für die Klasse `com.tutego.App`:

```
$ java -ea:com.tutego.App ClassWithMain
```

Aktiviere Assertions für das Default-Paket (dafür stehen die drei Punkte):

```
$ java -ea:... ClassWithMain
```

Aktiviere Assertions für das Paket com.tutego inklusive aller Unterpakete (auch dafür stehen drei Punkte):

```
$ java -ea:com.tutego... ClassWithMain
```

Aktiviere Assertions für das Paket com.tutego inklusive aller Unterpakete, aber deaktiviere sie für die Klasse App in dem Paket com.tutego:

```
$ java -ea:com.tutego... -da:com.tutego.App ClassWithMain
```

## 8.9 Zum Weiterlesen

Unterschiedliche Programmarchitekten 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 »Logging und Monitoring« aus »Java SE 9 Standard-Bibliothek« 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 erwartenden Ausnahmen mit getestet werden.



# Kapitel 9

## Geschachtelte.Typen

»Der Nutzen ist ein Teil der Schönheit.«

– Albrecht Dürer (1471–1528)

### 9.1 Geschachtelte 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 Kompilationseinheit, deklariert wurden. Es gibt darüber hinaus die Möglichkeit, eine Typdeklaration in eine andere Typdeklaration 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. Das ist das gleiche Prinzip wie bei lokalen Variablen: Diese sind nur sichtbar für die Methode, nicht für die gesamte Klasse. Geschachtelte Typen zeigen eine enge Abhängigkeit auf: Der geschachtelte Typ ist nur sinnvoll einzusetzen in Zusammenhang mit dem äußeren Typ.

Für eine Klasse In, die in eine Klasse Out gesetzt wird, sieht das im Quellcode so aus:

```
class Out {  
    class In {  
    }  
}
```

Wir sprechen in diesem Fall von einem *geschachteltem Typ*, engl. *nested type*, in diesem Fall einer geschachtelten Klasse, aber auch eine Schnittstellendeklaration oder Aufzählungstyp kann geschachtelt sein.

Geschachtelte Typen können statisch oder nichtstatisch sein. Die nichtstatischen geschachtelten Typen heißen *innere Typen*. Das Besondere bei den inneren Typen ist, dass sie immer eine Referenz auf ihren äußeren Typ haben.

Die Java-Spezifikation beschreibt vier Arten von geschachtelten Typen:

Art der Schachtelung	Beispiel mit geschachtelten Klassen
Statischer geschachtelter Typ	<pre>class Out {     static class In {}</pre>
Innerer Typ (nichtstatischer geschachtelter Typ)	<pre>class Out {     class In {}}</pre>
Lokaler (innerer) Typ	<pre>class Out {     Out() {         class In {}}}</pre>
Anonyme innere Klasse	<pre>class Out {     Out() {         Runnable r = new Runnable() {             public void run() {}}}</pre>

Tabelle 9.1 Die vier Arten von geschachtelten Typen

Eine geschachtelte (nicht-)statische Klasse heißt auch *Mitgliedsklasse* (engl. *member class*).



#### Hinweis

Das Gegenteil von geschachtelten Typen, also das, womit wir uns bisher die ganze Zeit beschäftigt haben, sind *Top-Level-Typen*. Die Laufzeitumgebung kennt nur Top-Level-Typen, und geschachtelte Typen werden letztendlich zu ganz »normalen« Klassendateien.

## 9.2 Statische geschachtelte Typen

Die einfachste Variante einer geschachtelten Klasse oder Schnittstelle wird wie eine statische Eigenschaft in die Klasse eingesetzt und heißt *statischer geschachtelter Typ*. Diese Typen können das Gleiche wie »normale« Typen, nur bilden sie quasi ein kleines Unterpackage mit einem Namensraum. Insbesondere sind zur Erzeugung von Exemplaren von statischen geschachtelten Klassen nach diesem Muster keine Objekte der äußeren Klasse nötig. (Die

weiteren inneren Typen, die wir kennenlernen wollen, sind alle nichtstatisch und benötigen einen Verweis auf das äußere Objekt.)

Deklarieren wir `Lamp` als äußere Klasse und `Bulb` als eine statische geschachtelte Klasse:

**Listing 9.1** src/main/java/com/tutego/insel/nested/Lamp.java, Lamp

```
public class Lamp {

    static String name = "Latifa";
    int watt = 1985;

    static class Bulb {
        void output() {
            System.out.println( name );
            System.out.println( watt ); // ☠
            // Cannot make a static reference to the non-static field watt
        }
    }

    public static void main( String[] args ) {
        Bulb bulp = new Lamp.Bulb(); // oder Lamp.Bulb bulp = ...
        bulp.output();
    }
}
```

Die statische geschachtelte Klasse `Bulb` besitzt Zugriff auf alle anderen statischen Eigenschaften der äußeren Klasse `Lamp`, in unserem Fall auf die Variable `name`. Ein Zugriff auf Objektvariablen ist aus der statischen geschachtelten Klasse heraus nicht möglich, da sie als gesonderte Klasse gezählt wird, die im gleichen Paket liegt. Der Zugriff von außen auf statische geschachtelte Klassen gelingt mit der Schreibweise ÄußererTyp.GeschachtelterTyp; 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 statische geschachtelte 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; geschachtelte Klassen dürfen ebenfalls `public` oder `paketsichtbar`, alternativ aber auch `protected` oder `private` sein. Eine private statische geschachtelte 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 geschachtelten Typen ermög-

licht für den Compiler einen etwas effizienteren Bytecode, ist aber ansonsten nicht in Gebrauch.

### Umsetzung von statischen geschachtelten Typen \*

Der Compiler generiert aus den geschachtelten Typen normale Klassendateien, die jedoch mit einigen so genannten *synthetischen Methoden* ausgestattet sind. Für die geschachtelten Typen generiert der Compiler neue Namen nach dem Muster: ÄußererTyp\$GeschachtelterTyp, das heißt, ein Dollar-Zeichen trennt die Namen von äußerem und geschachteltem Typ. Genauso heißt die entsprechende .class-Datei auf der Festplatte. Das wird auch der *binäre Name* des geschachtelten Typs genannt und ist zum Beispiel beim manuellen Laden wichtig.

## 9.3 Nichtstatische geschachtelte Typen

Ein nichtstatischer geschachtelter Typ ist ein innerer Typ vergleichbar mit einer Objekt-eigenschaft. Deklarieren wir eine innere Klasse Room in House:

**Listing 9.2** src/main/java/com/tutego/insel/nested/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 Klassen 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«.

### 9.3.1 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 geschachtelten Klassen aus

Abschnitt 9.2. »Statische geschachtelte Typen«; statische geschachtelte Typen existieren auch ohne Objekt der äußeren Klasse.

In einem Konstruktor oder in einer Objektmethode der äußeren Klasse kann einfach mit dem Schlüsselwort `new` 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 nichtstatischen geschachtelten Klassen 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(String[])`-Methode des Hauses ein `Room`-Objekt aufzubauen, schreiben wir:

**Listing 9.3** src/main/java/com/tutego/insel/nested/House.java, main()

```
House h = new House();
Room r = h.new Room();
```

Oder auch in einer Zeile:

```
Room r = new House().new Room();
```

### 9.3.2 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 9.4** src/main/java/com/tutego/insel/nested/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

Nichtstatische geschachtelte Klassen können beliebig geschachtelt sein, und da der Name eindeutig ist, gelangen wir mit `Klassenname.this` immer an die jeweilige Eigenschaft.

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.Chair c = r.new Chair();   // Exemplar von Chair in r
c.out();                                         // Methode von Chair

```

Die Qualifizierung mit dem Punkt bei `FurnishedHouse.Room.Chair` bedeutet nicht automatisch, dass `FurnishedHouse` ein Paket mit dem Unterpaket `Room` ist, in dem die Klasse `Chair` existiert. Die Doppelbelegung des Punktes 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.

### 9.3.3 Vom Compiler generierte Klassendateien \*

Für das Beispiel `House` und `Room` erzeugt der Compiler die Dateien `House.class` und `House$Room.class`. Damit die innere Klasse an die Attribute der äußeren gelangt, generiert der Compiler automatisch in jedem Exemplar der inneren Klasse eine Referenz auf das zugehörige Objekt der äußeren Klasse. Damit kann die innere Klasse auch auf nichtstatische Attribute der äußeren Klasse zugreifen. Für die innere Klasse ergibt sich folgendes Bild in `House$Room.class`:

```

class House$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.

### 9.3.4 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. Tabelle 9.2 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 9.2 Erlaubte Modifizierer

## 9.4 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.

### 9.4.1 Beispiel mit eigener Klassendeklaration

Im folgenden Beispiel deklariert die `main(...)`-Methode eine innere Klasse `Snowden` mit einem Konstruktor, der auf die finale Variable `PRISM` zugreift:

**Listing 9.5** src/main/java/com/tutego/insel/nested/NSA.java, NSA

```
public class NSA {

    public static void main( String[] args ) {
        final int PRISM = 1;
        int tempora = 2;
        tempora++;                                // (*)

        class Snowden {
            Snowden() {
                System.out.println( PRISM );
//                System.out.println( tempora ); // ☠ Auskommentiert ein Compilerfehler
            }
        }
        new Snowden();
    }
}
```

Die Deklaration der lokalen Klasse `Snowden` wird hier wie eine Anweisung eingesetzt. Ein Sichtbarkeitsmodifizierer ist bei 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. Zusätzlich kann sie auf lokale Variablen und Parameter zugreifen, allerdings nur dann, wenn die Variablen `final` sind. Dabei müssen finale Variablen nicht zwingend mit dem Modifizierer `final` gekennzeichnet werden, um `final` zu sein. Gibt es keinen Schreibzugriff auf Variablen, sind sie *effektiv final*. Die Variable `PRISM` ist explizit mit dem Modifizierer `final` markiert, also kann die innere Klasse darauf zugreifen. `tempora` ist nicht `final` (`tempora++` ist ein Schreibzugriff), und daher führt ein Lesezugriff in der inneren Klasse bei `println(tempora)` zu einem Compilerfehler – Eclipse meldet: »Local variable `tempora` defined in an enclosing scope must be final or effectively final«. Im Beispiel kann das einfach getestet werden: Wird die Zeile (\*) mit `tempora++;` auskommentiert, so ist `tempora` effektiv `final`, und `Snowden` kann auf `tempora` zugreifen.

Liegt die innere Klasse in einer statischen Methode, kann sie keine Objektmethoden der äußeren Klasse aufrufen.

### 9.4.2 Lokale Klasse für einen Timer nutzen

Damit die Beispiele etwas praxisnäher werden, wollen wir uns anschauen, wie ein Timer wiederholende Aufgaben ausführen kann. Die Java-Bibliothek bringt hier schon alles mit: Es gilt, ein Exemplar von `java.util.Timer()` zu bilden und der Objektmethode `scheduleAtFixedRate(...)` ein Exemplar vom Typ `TimerTask` zu übergeben. Die Klasse `TimerTask` schreibt eine abstrakte Methode `run()` vor, in die der nebenläufige und regelmäßig abzuarbeitende Programmcode gesetzt wird.

Nutzen wir das für ein Programm, das uns sofort und regelmäßig daran erinnert, wie wichtig doch Sport ist:

**Listing 9.6** src/main/java/com/tutego/insel/nested/SportReminder.java, SportReminder

```
public class SportReminder {
    public static void main( String[] args ) {
        class SportReminderTask extends TimerTask {
            @Override public void run() {
                System.out.println( "Los, beweg dich, du faule Wurst!" );
            }
        }
        new Timer().scheduleAtFixedRate( new SportReminderTask(),
            0 /* ms delay */,
            1000 /* ms period */ );
    }
}
```

Unsere Klasse `SportReminderTask`, die `TimerTask` erweitert, ist direkt in `main(...)` deklariert. Das erzeugte Exemplar kommt später in `scheduleAtFixedRate(...)`, und los rennt der Timer, um uns jede Sekunde an die Wichtigkeit von Bewegung zu erinnern. Der Vorteil der lokalen Klassendeklaration ist, dass sie bis auf `main(...)` kein anderer sehen kann.

## 9.5 Anonyme innere Klassen

Anonyme Klassen gehen noch einen Schritt weiter als lokale Klassen: Auch sie sind innere Klassen, haben aber keinen Namen und erzeugen immer automatisch ein Objekt; Klassendeklaration und Objekterzeugung 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 Klassenname(Optionale Argumente) { ... }`. Steht hinter new ein Klassentyp, dann ist die anonyme Klasse eine Unterklasse von Klassenname. Es lassen sich mögliche Argumente für den Konstruktor der Basisklasse angeben (das ist zum Beispiel dann nötig, wenn die Oberklasse keinen parameterlosen Konstruktor deklariert).
- ▶ `new Schnittstellename() { ... }`. Steht hinter new der Name einer Schnittstelle, dann erbt die anonyme Klasse von Object und implementiert die Schnittstelle Schnittstellename. 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 – wohl aber Exemplarinitialisierer – und nur Objektmethoden und finale statische Variablen erlaubt.

### 9.5.1 Nutzung einer anonymen inneren Klasse für den Timer

Eben gerade haben wir für den Timer extra eine neue lokale Klasse deklariert, aber genau genommen haben wir diese nur einmal nutzen müssen, nämlich um ein Exemplar zu bilden und scheduleAtFixedRate(...) übergeben zu können. Das ist ein perfektes Szenario für anonymous innere Klassen. Aus

```
class SportReminderTask extends TimerTask {
    @Override public void run() { ... }
}
new Timer().scheduleAtFixedRate( new SportReminderTask(), ... );
```

wird

**Listing 9.7** src/main/java/com/tutego/insel/nested/ShorterSportReminder.java, main()

```
new Timer().scheduleAtFixedRate( new TimerTask() {
    @Override public void run() {
        System.out.println( "Los, ..." );
    }
},
0 /* ms delay */,
1000 /* ms period */);
```

Im Kern ist es also eine Umwandlung von new SportReminderTask() in new TimerTask() { ... }. Von dem Klassennamen SportReminderTask ist nichts mehr zu sehen, das Objekt ist anonym.

### Hinweis

Eine anonyme Klasse kann Methoden der Oberklasse überschreiben, Operationen aus Schnittstellen implementieren und sogar neue Eigenschaften anbieten:

**Listing 9.8** src/main/java/com/tutego/insel/nested/ObjectWithQuote.java, main()

```
String s = new Object() {
    String quote( String s ) {
        return String.format( "%s", s );
    }
}.quote( "Cora" );
System.out.println( s ); // 'Cora'
```

Der neu deklarierte anonyme Typ hat eine Methode quote(String), 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) bzw. der Schnittstelle bekannt. (Wir lassen die Tatsache außen vor, dass eine Anwendung mit Reflection auf die Methoden zugreifen kann.)



### 9.5.2 Umsetzung innerer alterntiver Klassen \*

Auch für innere anonyme Klassen erzeugt der Compiler eine normale Klassendatei. Wir haben gesehen, dass der Java-Compiler bei einer »normalen« geschachtelten 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 usw.

### Ausnahmen in inneren anonymen Klassen

In einem Stack-Trace taucht der generierte Klassename auf, wenn es eine Ausnahme gab. Ist etwa die Deklaration eingebettet in eine main(...)-Methode der Klasse T:

```
new Object() { String nuro() { throw new IllegalStateException(); } }.nuro();
```

So folgt bei der Ausführung:

```
Exception in thread "main" java.lang.IllegalStateException
at T$1.nuro(T.java:6)
at T.main(T.java:6)
```

### 9.5.3 Konstruktoren innerer anonymer 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.

Wir wollen eine innere Klasse schreiben, die Unterkelas von `java.awt.Point` ist. Sie soll die `toString()`-Methode überschreiben:

**Listing 9.9** src/main/java/com/tutego/insel/nested/InnerToStringPoint.java, main()

```
Point p = new Point( 10, 12 ) {
    @Override public String toString() {
        return "(" + x + "," + y + ")";
    }
};

System.out.println( p ); // (10,12)
```

Die anonyme Unterkasse wird also durch den normalen Konstruktor von `Point` initialisiert.

### Exemplarinitialisierungsblöcke bei inneren anonymen Klassen

Da aber anonyme Klassen keinen Namen haben, muss für Konstrukturen ein anderer Weg gefunden werden. Hier helfen *Exemplarinitialisierungsblöcke*, also Blöcke in geschweiften Klammern direkt innerhalb einer Klasse, die wir schon in [Kapitel 6](#), »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 Unterkasse von `Point` und initialisiert im Konstruktor einen Punkt mit Zufallskoordinaten. Aus diesem speziellen Punkt-Objekt lesen wir dann die Koordinaten wieder aus:

**Listing 9.10** src/main/java/com/tutego/insel/nested/AnonymousAndInside.java, main()

```
java.awt.Point p = new java.awt.Point() {
{
    x = (int)(Math.random() * 1000); y = (int)(Math.random() * 1000);
}
};

System.out.println( p.getLocation() ); // java.awt.Point[...
```

```
System.out.println( new java.awt.Point( -1, 0 ) {{
    y = (int)(Math.random() * 1000);
}.getLocation() );                                // java.awt.Point[x=-1,y=...]
```

### Sprachlichkeit

Wegen der beiden geschweiften Klammern heißt diese Variante auch *Doppelklammer-Initialisierung* (engl. *double brace initialization*).

Die Doppelklammer-Initialisierung ist kompakt, wenn etwa Datenstrukturen oder hierarchische Objekte initialisiert werden sollen.

### Beispiel \*

Im folgenden Beispiel erwartet `appendText(...)` ein Objekt vom Typ `HashMap`, das durch den Trick direkt initialisiert wird:

```
String s = new DateTimeFormatterBuilder()
.appendText( ChronoField.AMPM_OF_DAY,
    new HashMap<Long, String>() {{ put(0L, "früh");put(1L,"spät" ); }} )
.toFormatter().format( LocalTime.now() );
System.out.println( s );
```

Im nächsten Beispiel bauen wir eine geschachtelte Map, das ist ein Assoziativspeicher. Der enthält an einem Punkt wieder einen anderen Assoziativspeicher:

```
Map<String, Object> map = new HashMap<String, Object>() {{
    put( "name", "Chris" );
    put( "address", new HashMap<String, Object>() {{
        put( "street", "Feenallee 1" );
        put( "city", "Elefenberg" );
    } );
}};
```



### Warnung



Die Doppelklammerinitialisierung ist nicht ganz »billig«, da eine Unterklasse aufgebaut wird, also neuer Bytecode generiert wird. Zudem hält die innere Klasse eine Referenz auf die äußere Klasse fest. Des Weiteren kann es Probleme mit `equals(...)` geben, da wir mit der Doppelklammerinitialisierung eine Unterklasse schaffen, die vielleicht mit `equals(...)` nicht mehr gültig verglichen werden kann, denn die Class-Objekte sind jetzt nicht mehr identisch. Das spricht in der Summe eher gegen diese Konstruktion. Der Typ `Map` bietet mit den statischen `of(...)`/`entry(...)`-Methoden eine bessere Möglichkeit.

**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 folgende Beispiel:

```
System.out.println( new java.awt.Point( -1, 0 ) {{
    y = (int)(Math.random() * 1000);
}}.getLocation() );                                // java.awt.Point[x=-1,y=...]
```

**Beispiel**

Wir initialisieren ein Objekt `BigDecimal`, das beliebig große Ganzahlen 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() );
}}
```

**9.6 Zugriff auf lokale Variablen aus lokalen und anonymen Klassen \***

Lokale und anonyme Klassen können auf die lokalen Variablen und 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 Arrays 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 9.11** src/main/java/com/tutego/insel/nested/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<>();
```

```

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.

## 9.7 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 9.3.2](#), »Die this-Referenz«, gelernt. 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 Oberklasse ist. Das eigentliche Problem besteht darin, dass hier zweidimensionale Namensräume hierarchisch kombiniert werden müssen. Die eine Dimension sind die Variablen und 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 9.12** src/main/java/com/tutego/insel/nested/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.

### 9.7.1 Geschachtelte Klassen greifen auf private Eigenschaften zu

Die äußere umschließende Klasse kann auf private Eigenschaften der geschachtelten Klasse zugreifen. Das folgende Beispiel soll das illustrieren:

**Listing 9.13** src/main/java/com/tutego/insel/nested/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 Kompilationseinheit (also Datei) definiert wird, kann schon nicht mehr auf `NotSoPrivate.Family` zugreifen, und natürlich hat auch keine Klasse einer anderen Kompilationseinheit Zugriff.

### Zugriffsrechte \*

Eine geschachtelte Klasse kann auf alle Attribute der äußeren Klasse zugreifen. Da eine geschachtelte Klasse in eine ganz normale Klassendatei 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 geschachtelte. Und da eine geschachtelte Klasse als normale Klassendatei im gleichen Paket sitzt, kann sie ebenfalls ohne Verrenkungen auf paketsichtbare und `protected`-Eigenschaften der äußeren Klasse zugreifen. Eine geschachtelte 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$0( House house ) {
        return house.owner;
    }
}

```

Die statische Methode `access$0(...)` ist der Helpershelfer, der für ein gegebenes `House` das `private` Attribut nach außen gibt. Da die geschachtelte 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 geschachtelten Klasse genutzte `private` Attribut erzeugt der Compiler eine solche Methode. Wenn wir eine weitere `private` Variable `int size` hinzunähmen, 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 paketsichtbaren 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 mit dem Jar-Werkzeug abgeschlossen werden, was bei Java *Sealing* heißt.

## 9.8 Nester

Geschachtelte Klassen haben eine Besonderheit bei privaten Eigenschaften, nämlich dass sowohl der äußere Typ auf private Eigenschaften der geschachtelten Klasse zugreifen kann als auch die geschachtelte Klasse auf private Eigenschaften der äußeren Klasse. Das führt zu einer besonderen Form der Umsetzung in Java-Bytecode, die die privaten Eigenschaften für andere Klassen freigeben muss, denn auf der Ebene der virtuellen Maschine gibt es nur Top-Level-Klassen.

Ab Java 11 gibt es eine Neuerung, so genannte *Nester*, beschrieben im JEP 181: »Nest-Based Access Control«.<sup>1</sup> Der Bytecode bildet damit viel besser ab, dass ein Typ in einem Typ geschachtelt ist. In der Bibliothek gibt es drei neue Objektmethoden bei Class: getNestHost(), isNestmateOf(Class) und getNestMembers().

**Listing 9.14** src/main/java/com/tutego/insel/nested/OuterNest.java

```
package com.tutego.insel.nested;

public class OuterNest {

    public static class In1 {
        private void intern1() {
            new OuterNest().new In2().intern2();
        }
    }

    public class In2 {
        private void intern2() {
```

---

<sup>1</sup> <http://openjdk.java.net/jeps/181>

```
    new In1().intern1();
}
}

public static void main( String[] args ) {
    for ( Class<?> clazz : OuterNest.class.getNestMembers() ) {
        System.out.printf( "%s %b%n", clazz, clazz.isNestmateOf( OuterNest.class ) );
    }
}
}
```

Unter Java 11 kommt auf die Konsole:

```
class com.tutego.insel.nested.OuterNest true
class com.tutego.insel.nested.OuterNest$In1 true
class com.tutego.insel.nested.OuterNest$In2 true
```

Unter Java 10 erscheint lediglich die erste Zeile!

## 9.9 Zum Weiterlesen

Geschachtelte Typen, insbesondere innere anonyme Klassen, zur Implementierung von funktionalen Schnittstellen haben mit der Einführung von Lambda-Ausdrücken an Bedeutung verloren. [Kapitel 12, »Lambda-Ausdrücke und funktionale Programmierung«](#), geht ins Detail.



# Kapitel 10

## Besondere Typen der Java SE

»Einen Rat befolgen heißt, die Verantwortung verschieben.«

– Johannes Urzidil (1896–1970)

Programmieren wir mit Java, nutzen wir oftmals unbewusst Typen aus der Standardbibliothek. Das fällt oft gar nicht auf, da zum einen das Paket `java.lang` automatisch importiert wird – und damit Typen wie `String`, `Object` immer eingebunden sind – und weil zum anderen einiges hinter den Kulissen passiert. Sechs Beispiele:

- ▶ Erweitert eine Oberklasse keine eigene Klasse, so erbt sie automatisch von `java.lang.Object`.
- ▶ Ist ein primitiver Datentyp gegeben, aber ein Objekttyp gewünscht, konvertiert der Compiler den einfachen Datentyp in ein Wrapper-Objekt. Das nennt sich *Boxing*.
- ▶ Hängen wir Strings mit `+` zusammen, erzeugt der Compiler – zumindest bis Java 8 – automatisch einen `java.lang.StringBuilder`, hängt die Segmente mit `append(...)` zusammen und liefert dann mit `toString()` einen neuen String. Bei keinem anderen Referenztyp erlaubt der Compiler die »Addition«, sondern nur Vergleiche mit `==` oder `!=`.
- ▶ Beim erweiterten `for` erwartet der Compiler entweder ein Array oder etwas vom Typ `Iterable`: Von diesen Objekten erfragt er den `Iterator` und läuft selbstständig durch die Sammlung.
- ▶ Damit `try` mit Ressourcen verwendet werden kann, erwartet der Compiler ein `AutoCloseable` und ruft auf diesen Objekten im `finally`-Block die `close()`-Methode auf.
- ▶ Bei der Deklaration eines Aufzählungstyps mit `enum` generiert der Compiler eine von `java.lang.Enum` abgeleitete Klasse; selbst darf ein Programmierer keine Unterklassen von `Enum` bilden, das verbietet der Compiler.

Dieses Kapitel stellt unterschiedliche Typen vor, die in irgendeiner Weise bevorzugt werden oder eine Sonderstellung in Java einnehmen, weil sie allgegenwärtig sind. Dazu zählen:

- ▶ die Basisklasse `Object`
- ▶ Vergleichsobjekte
- ▶ Wrapper-Klassen
- ▶ Aufzählungen und die Schnittstellen `Iterable` und `Iterator`

- ▶ Aufzählungstypen, enum und die Sonderklasse Enum
- ▶ Nicht zufällig liegen einige Typen im Paket java.lang.

## 10.1 Object ist die Mutter aller Klassen

java.lang.Object ist die oberste aller Elternklassen. Somit spielt diese Klasse eine ganz besondere Rolle, da alle anderen Klassen automatisch Unterklassen sind und die Methoden erben bzw. überschreiben.

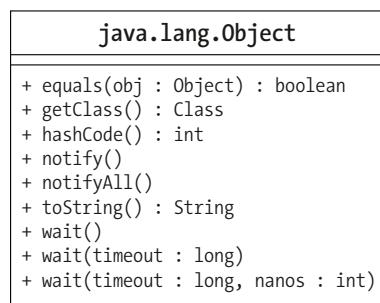


Abbildung 10.1 UML-Diagramm der absoluten Basisklasse Object

### 10.1.1 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 ein Aufruf von `x.getClass()` von unterschiedlichen Exemplaren `x` vom Typ `X` immer dasselbe `Class<X>`-Objekt liefern. Die `Class`-Exemplare lassen sich also sicher mit `==` prüfen.

[zB]

#### Beispiel

Die Objektmethode `getName()` eines `Class`-Objekts liefert den Namen der Klasse:

```
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 eine Klasse, eine Schnittstelle, ein Array oder ein primitiver Typ ist. Beispiele sind:

- ▶ `String.class`
- ▶ `Integer.class`
- ▶ `int.class` (das nicht mit `Integer.class` identisch ist)

Der Ausdruck ist immer vom Typ `Class`. Bei primitiven Typen liefert die Schreibweise `primitiveType.class` das gleiche Ergebnis wie `WrapperType.TYPE`; es ist also `Integer.TYPE` identisch mit `int.class`. `Class`-Objekte spielen insbesondere bei dynamischen Abfragen über die so genannte Reflection eine Rolle. Zur Laufzeit können so beliebige Klassen geladen, Objekte erzeugt und Methoden aufgerufen werden.

### 10.1.2 Objektidentifikation mit `toString()`

Jedes Objekt sollte sich durch die Methode `toString()` mit einer Zeichenkette identifizieren und den Inhalt der interessanten Attribute als Zeichenkette liefern.

#### Beispiel

[zB]

Die Klasse `Point` implementiert `toString()` so, dass der Rückgabe-String 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 `printXXX(...)` mit einer Objektreferenz als Argument aufgerufen werden. Ähnliches gilt für den Zeichenkettenoperator `+` mit einer Objektreferenz als Operand:

**Listing 10.1** src/main/java/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 10.2** src/main/java/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 Unterklasse 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.



### Warnung

Einige kreative Programmierer nutzen die `toString()`-Repräsentation für Objektvergleiche, etwa so: Wenn wir zwei `Point`-Objekte `p` und `q` haben und `p.toString().equals(q.toString())` ist, dann sind beide Punkte eben gleich. Doch ist es hochgradig gefährlich, sich auf die Rückgabe von `toString()` zu verlassen, aus mehreren Gründen: Offensichtlich ist, dass `toString()` nicht unbedingt überschrieben sein muss. Zweitens muss `toString()` nicht unbedingt alle Elemente repräsentieren, und die Ausgabe könnte abgekürzt sein. Drittens können natürlich Objekte `equals`-gleich sein, auch wenn ihre String-Repräsentation nicht gleich ist, was etwa bei URL-Objekten der Fall ist. Der einzige erlaubte Fall für so eine Konstruktion wäre `String/StringBuilder/StringBuffer/CharSequence`, wo es ausdrücklich um Zeichenketten geht. Neben dem fehlerhaften Verhalten gibt es in der Regel ein massives Performance-Problem. `equals(...)` nimmt ja in der Regel Abkürzungen, sodass zum Beispiel `obj.equals(obj)` sofort `true` liefert. Oder dass bei Datenstrukturen erst einmal auf die identische Länge getestet wird, bevor es zum Elementvergleich kommt.

### 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 Hashwert 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 Hashwert.

Zwar sagt der Hashwert 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.

### Beispiel

[zB]

Ein Objekt der class A {} wird gebildet, und `toString()` liefert die ID:

```
class A {}
A a = new A();
System.out.println( "1. " + a + ", 2. " + a + ", 3. " + new A() );
```

Die Ausgabe kann sein:

1. Main\$1A@4554617c, 2. Main\$1A@4554617c, 3. Main\$1A@74a14482

Bei mehrfachen Aufrufen von `toString()` auf einem Exemplar – Ausgabe 1 und 2 – bleibt die Rückgabe konstant. Wird das Programm neu gestartet, kann der Hashwert anders aussehen.

### 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 IntelliJ 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://commons.apache.org/proper/commons-lang>) auf den richtigen Weg.

### 10.1.3 Objektgleichheit mit `equals(...)` und Identität

Ob zwei Referenzen dasselbe Objekt repräsentieren, stellt der Vergleichsoperator `==` fest, `!=` das Gegenteil. Die Operatoren testen die Identität, wissen aber nichts von einer möglichen

inhaltlichen 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 Zeichen um Zeichen vergleicht:

```
String firstname = "Christian";
if ( "Christian".equals( firstname ) )
    ...
...
```

```
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:

**Listing 10.3** `java/lang/Object.java, equals()`

```
public boolean equals( Object obj ) {
    return this == obj;
}
```

Überschreibt eine Klasse `equals(Object)` 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(Object)`-Methode, sodass die Laufzeitumgebung unter Umständen ungewollt bei `Object` und seinem Referenzvergleich landet. Dies hat ungeahnte Folgen, und diese Fehleinschätzung kommt leider bei Exemplaren der Klassen `StringBuilder` und `StringBuffer` vor, die kein eigenes `equals(...)` implementieren. Wir haben dies bereits in Kapitel 5, »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 Sig-

natur 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(Object)` angeheftet sein.

Die `equals(Object)`-Methode stellt einige Anforderungen:

1. Heißt der Vergleich `equals(null)`, so ist das Ergebnis immer `false`.
2. Kommt ein `this` hinein, lässt sich eine Abkürzung nehmen und `true` zurückliefern.
3. Das Argument ist zwar vom Typ `Object`, aber dennoch vergleichen wir immer konkrete Typen. Eine `equals(Object)`-Methode einer Klasse `X` wird sich daher nur mit Objekten vom Typ `X` vergleichen lassen. Eine spannende Frage ist, ob `equals(Object)` auch Unterklassen von `X` beachten soll.
4. Eine Implementierung von `equals(Object)` 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(Object)`-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 Parametertypen, wohl aber 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.



### Grundlegender Aufbau der `equals(...)`-Methode

Die Punkte 1 bis 4 sind nur die Vorbereitung bis zum eigentlichen Vergleich. Dann geht es darum, Attribut für Attribut vom eigenen Objekt mit dem Zustand eines anderen Objektes zu vergleichen. Wir unterscheiden Tests von primitiven Werten, Tests von Arrays und Tests von Referenztypen.

- Für `boolean` und alle ganzzahligen primitiven Werte ist ein einfacher `==`-Vergleich möglich. Beim `==`-Vergleich von Fließkommazahlen kommt es wegen des Sonderwertes `NaN` zu einem Problem; das umgeht die Konvertierung in ein Ganzzahl-Bit-Muster. Grundsätz-

lich könnte auch ein Test mit den statischen `compareTo(...)`-Methoden in den Wrapper-Klassen helfen, in dem mit `compareTo(...) == 0` auf die Gleichheit geprüft wird.

- ▶ Für den Vergleich von Arrays bietet sich `Arrays.equals(array1, array2)` an.
- ▶ Für den Vergleich von Referenzen wird der Vergleich an die Objekte weitergegeben, wobei darauf zu achten ist, ob die Referenzvariablen `null` sind. Die praktische Hilfsmethode `Objects.equals(obj1, obj2)` kümmert sich darum.



### Tipp

Natürlich ließe sich auf die Objektmethode `equals(...)` der numerischen Wrapper-Klassen zurückgreifen, doch das würde bedeuten, für jeden primitiven Vergleich immer neue Objekte aufzubauen. Das kostet unnötig Zeit, denn `equals(...)` und auch `hashCode()`-Methoden müssen schnell sein, da sie bei Operationen in Datenstrukturen oft aufgerufen werden.

### Beispiel einer eigenen `equals(...)`-Methode

Die beiden ersten Punkte sind leicht erfüllbar, und ein Beispiel für einen `Club` mit den Attributnen `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 ist das Ziel erreicht?



### Hinweis

Die `equals(...)`-Methode gibt bei nicht passenden Typen immer `false` zurück und löst keine Ausnahme aus.

### 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 Unterklasse von `Club` ist. Diese Unterklasse wird wie die Oberklasse 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 Unterklasse 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()` und einem einfachen Referenzvergleich.<sup>1</sup> Korrekt ist daher Folgendes:

**Listing 10.4** src/main/java/com/tutego/insel/object/Club.java, Club

```
public class Club {

    int numberOfPersons;
    int sm;

    @Override
    public boolean equals( Object o ) {
        if ( o == null )
            return false;
```

---

<sup>1</sup> Class-Objekte müssen nicht über `equals(...)` verglichen werden, denn Class-Objekte haben kein eigenes `equals(...)`, sondern erben nur die Implementierung von `Object`, und dort wird auch nur ein Referenzvergleich vorgenommen.

```

if ( o == this )
    return true;

if ( o.getClass() != 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 10.1.5](#), »Hashwerte 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(...)<sup>2</sup> ist der folgende: Wenn das Objekt nicht verändert wird, muss das Ergebnis 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.com/` und `http://www.tutego.com/` aber letztendlich `http://www.tutego.de/` steckt, liefert equals(...) die Rückgabe true:

---

2 Eine korrekte Implementierung der Methode equals(...) bildet eine Äquivalenzrelation. Lassen wir die null-Referenz außen vor, ist sie reflexiv, symmetrisch und transitiv.

**Listing 10.5** src/main/java/com/tutego/insel/object/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.

**10.1.4 Klonen eines Objekts mit clone() \***

Zum Replizieren eines Objekts gibt es oft zwei Möglichkeiten:

- ▶ einen Konstruktor (auch Copy-Konstruktor 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.

**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 kovarianter Rückgabetypen anpassen. Die `clone()`-Methode bei `java.awt.Point` bleibt allerdings bei `Object`.

Array-Objekte bieten standardmäßig `clone()`. Speichern die Arrays jedoch nichtprimitive Werte, liefert `clone()` nur eine flache Kopie, was bedeutet, dass das neue Array-Objekt, 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 nichtstatischen 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 10.6 src/main/java/com/tutego/insel/object/Player.java

```
package com.tutego.insel.object;
```

```
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();
    }
}
}

```

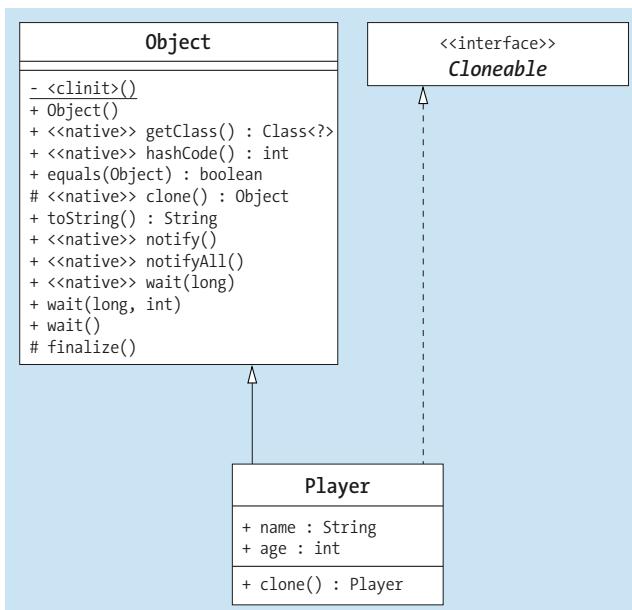
Da es kovariante Rückgabetypen gibt, gibt `clone()` nicht lediglich `Object`, sondern den Untertyp `Player` zurück. Testen wir die Klasse etwa so:

**Listing 10.7** src/main/java/com/tutego/insel/object/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

```



**Abbildung 10.2** Player erweitert Object und implementiert Cloneable.



### 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()`, zusammen. Wenn die `clone()`-Methode überschrieben wird, sollte auch `equals(...)` angepasst werden, denn ohne ein überschriebenes `equals(...)` bleibt Folgendes in `Object` stehen:

**Listing 10.8** `java/lang/Object.java, equals()`

```
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:

**Listing 10.9** java/lang/Object.java, Ausschnitt

```
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:

**Listing 10.10** java.awt.geom/Point2D.java, Ausschnitt

```
public abstract class Point2D implements Cloneable {
    ...
    public Object clone()
    ...
}
```

**Listing 10.11** java.awt/Point.java, Ausschnitt

```
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 wieder einführen – Unterklassen können `throws`-Klauseln 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 10.12** src/main/java/com/tutego/insel/object/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.ColoredPoint.clone(ColoredPoint.java:10)
at ...
Caused by: java.lang.CloneNotSupportedException
... 2 more
```

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.

### 10.1.5 Hashwerte über hashCode() liefern \*

Die Methode hashCode() soll zu jedem Objekt eine möglichst eindeutige Integer-Zahl (sowohl positiv als auch negativ) liefern, die das Objekt identifiziert. Die Ganzzahl heißt Hashcode oder Hashwert, und hashCode() ist die Implementierung einer Hashfunktion. Nötig sind Hashwerte, 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.

```
class java.lang.Object
```

- `int hashCode()`  
Liefert den Hashwert eines Objekts. Die Basisklasse `Object` implementiert die Methode nativ.

### Spieler mit Hashfunktion

Im folgenden Beispiel soll die Klasse `Player` die Methode `hashCode()` aus `Object` überschreiben. Um die Objekte erfolgreich in einem Assoziativspeicher abzulegen, ist gleichfalls `equals(...)` nötig, das die Klasse `Player` ebenfalls implementiert:

**Listing 10.13** `src/main/java/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 10.14** src/main/java/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 ihn die Standardimplementierung 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 des Restwerts 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/Hashwerte

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 bzw. 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 10.15** src/main/java/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 10.16** src/main/java/com/tutego/insel/object/hashcode/Player.java, `hashCode()`, Ausschnitt

```
result = 31 * result + ((name == null) ? 0 : name.hashCode());
```

Dass eine `equals(...)`-Methode bzw. `hashCode()` einer Klasse den Vergleich bzw. die Hashcode-Berechnung nicht an die Unterobjekte delegiert, sondern selbst umsetzt, ist unüblich.

### hashCode()-Methoden der Wrapper-Klassen

Jede Wrapper-Klasse deklariert eine statische `hashCode(...)`-Methode, mit der sich der Hashwert eines primitiven Elements berechnen lässt. (Genauer geht der [Abschnitt 10.5](#), »Wrapper-Klassen und Autoboxing«, darauf ein.) Um den Hashwert eines ganzen Objekts zu errechnen, müssen folglich alle einzelnen Hashwerte berechnet und dann zu einer Ganzzahl verknüpft werden. Schematisch sieht das so aus:

```

int h1 = WrapperKlasse.hashCode( value1 );
int h2 = WrapperKlasse.hashCode( value2 );
int h3 = WrapperKlasse.hashCode( value3 );
...

```

Eclipse nutzt zur Verknüpfung der Hashwerte folgendes Muster, das ein guter Ausgangspunkt ist:

```

int result = h1;
result = 31 * result + h2;
result = 31 * result + h3;
...

```

Nutzen wir die statischen `hashCode(...)`-Methoden der Wrapper-Klassen, müssen wir nur noch mit dem Datentyp `int` arbeiten und nicht wissen, wie etwa aus einem `double` der Hashwert berechnet wird. Uninteressant ist es aber nicht, daher kurz die Implementierung:

Klasse	Klassenmethode, static int	Implementierung
Boolean	<code>hashCode(boolean value)</code>	<code>value ? 1231 : 1237</code>
Byte	<code>hashCode(byte value)</code>	<code>(int)value</code>
Short	<code>hashCode(short value)</code>	<code>(int)value</code>
Integer	<code>hashCode(int value)</code>	<code>value</code>
Long	<code>hashCode(long value)</code>	<code>(int)(value ^ (value &gt;&gt;&gt; 32))</code>
Float	<code>hashCode(float value)</code>	<code>floatToIntBits(value)</code>
Double	<code>hashCode(double value)</code>	<code>(int)(doubleToLongBits(value) ^ (doubleToLongBits(value) &gt;&gt;&gt; 32));</code>
Character	<code>hashCode(char value)</code>	<code>(int)value</code>

Tabelle 10.1 Statische `hashCode(...)`-Methoden und Implementierung

### 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 Unter-Arrays 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 Unter-Arrays von zwei zu testenden Arrays nur identisch oder auch gleich sein dürfen. Diese Frage hatten wir

schon in [Abschnitt 4.1.19](#), »Die Klasse Arrays zum Vergleichen, Füllen, Suchen, Sortieren nutzen«, angesprochen.

Enthält unsere Klasse ein Array und soll es in einem equals(...) mitberücksichtigt werden, so sind prinzipiell drei Varianten zum Umgang mit diesem Array möglich. Arrays selbst mit == wie primitive Werte zu vergleichen ist in Ordnung, wenn die Identität der Arrays beim Vergleich gewünscht ist. Während viele Klassen die equals(...)-Methode von Object überschreiben, bieten Array-Objekte keine eigene equals(...)-Methode. Ergebnis eines arrays1.equals(arrays2)-Aufrufs wäre folglich ein Identitätsvergleich. Ein wirklicher inhaltlicher Vergleich ist mit Methoden der Utility-Klasse Arrays möglich. Hier gibt es jedoch zwei Methoden, die in Frage kämen: Arrays.equals(Object[] a, Object[] a2) geht jedes Element von a durch, also bei mehrdimensionalen Arrays jede Referenz auf ein Unter-Array, und testet, ob es identisch mit einem zweiten Array a2 ist. Wenn also zwei gleiche, aber nicht identische Haupt-Arrays identische Unter-Arrays besitzen, liefert Arrays.equals(...) die Rückgabe true, aber nicht, wenn die Unter-Arrays zwar gleich, aber nicht identisch sind. Spielt das eine Rolle, so ist Arrays.deepEquals(...) die passende Methode, denn sie fragt immer mit equals(...) die Unter-Arrays ab.

Bei der Berechnung des Hashwerts gibt es eine vergleichbare Frage. Die Arrays-Klasse bietet zur Berechnung des Hashwerts 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 Hashwert. Nehmen wir ein mehrdimensionales Array an. Dann ist das Unterelement ebenfalls ein Array. Arrays.hashCode(...) wird dann, wie erwähnt, nur die hashCode()-Methode auf dem Array-Objekt aufrufen, während Arrays.deepHashCode(...) auch in das Unter-Array hinabsteigt und so lange Arrays.deepHashCode(...) auf allen Unter-Arrays aufruft, bis ein equals(...)-Vergleich auf einem Nicht-Array 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 10.17** src/main/java/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;
}

```

### Hashwert einer Fließkommazahl

Abhängig von den Datentypen sehen die Berechnungen immer etwas unterschiedlich aus. Während Ganzzahlen direkt in einen Ganzzahlausdruck für den Hashwert eingebracht werden können, sind im Fall von `double` die statischen Konvertierungsmethoden `Double.doubleToLongBits(double)` und `Float.floatToIntBits(float)` im Einsatz.

Die Datentypen `double` und `float` haben eine weitere Spezialität, da `Double.NaN` und `Float.NaN` und das Vorzeichen der 0 zu beachten sind, wie [Kapitel 22, »Bits und Bytes, Mathematisches und Geld«](#), 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 »-9.223.372.036.854.775.808 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 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 Hashwerte übereinstimmen müssen. Formal gesehen heißt das:

```
x.equals( y ) &
x.hashCode() == y.hashCode()
```

So berechnet sich der Hashwert bei Point-Objekten aus den Koordinaten. Zwei Punkt-Objekte, die inhaltlich gleich sind, haben die gleichen Koordinaten und damit auch den gleichen

Hashwert. Wenn Objekte den gleichen Hashwert aufweisen, aber nicht gleich sind, handelt es sich um eine Kollision und den Fall, dass in der Gleichung nicht die Äquivalenz gilt. Anders ausgedrückt: Es ist falsch, davon auszugehen, dass, wenn der Hashwert von zwei Objekten gleich ist, auch die Objekte gleichwertig sind.

### 10.1.6 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 `hashCode()`, 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 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 `equals(...)`-gleichen 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.

#### Beispiel

Einige Klassen überschreiben `hashCode()` nicht, sodass `identityHashCode(...)` gleich dem Hashwert ist. Dazu zählt etwa die Klasse `StringBuilder`:

```
StringBuilder sb1 = new StringBuilder(), sb2 = new StringBuilder();
System.out.printf( "%d %d%n", System.identityHashCode(sb1), sb1.hashCode() );
// zum Beispiel 1829164700 1829164700
System.out.printf( "%d %d%n", System.identityHashCode(sb2), sb2.hashCode() );
// zum Beispiel 460141958 460141958
```

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 problemlos zwei unterschiedliche Objekte im Speicher geben, für die `System.identityHashCode(...)` gleich ist.

### 10.1.7 Aufräumen mit `finalize()` \*

Wenn die automatische Speicherbereinigung feststellt, dass es keine Referenz mehr auf ein bestimmtes Objekt gibt, so ruft sie automatisch die besondere Methode `finalize()` auf diesem Objekt auf. Danach kann die automatische Speicherbereinigung das Objekt entfernen. Wir können diese Methode für eigene Aufräumarbeiten überschreiben, die *Finalizer* genannt wird (ein Finalizer hat nichts mit dem `finally`-Block einer Exception-Behandlung zu tun).

Seit Java 9 ist die Methode deprecated<sup>3</sup>, um Entwickler zu ermutigen, auf diese Methode zu verzichten, obwohl sie natürlich weiterhin von der JVM aufgerufen wird. Es gibt mehrere Probleme. Einige Entwickler verstehen die Arbeitsweise der Methode nicht richtig und denken, sie wird immer aufgerufen. Doch hat die virtuelle Maschine Fantastillionen Megabyte an Speicher zur Verfügung und wird dann beendet, gibt sie den Heap-Speicher als Ganzes dem Betriebssystem zurück. In so einem Fall gibt es keinen Grund für eine automatische Speicherbereinigung als Grabträger und folglich keinen Aufruf von `finalize()`. Und wann genau 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 – alles ist vollständig nichtdeterministisch und von der Implementierung der automatischen Speicherbereinigung abhängig. Üblicherweise werden Objekte mit `finalize()` von einem Extra-Garbage-Collector behandelt, und der arbeitet langsamer als der normale GC, was somit ein Nachteil ist.

### Sprachvergleich

Einen Destruktor, der wie in C++ am Ende eines Gültigkeitsbereichs einer Variablen aufgerufen wird, gibt es in Java nicht.

<sup>3</sup> <https://bugs.openjdk.java.net/browse/JDK-8165641>

```
class java.lang.Object
```

- `@Deprecated(since="9") protected void finalize() throws Throwable`

Die Methode wird von der automatischen Speicherbereinigung 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 nicht `public` setzen.

### Einmal Finalizer, vielleicht mehrmals die automatische Speicherbereinigung

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 die automatische Speicherbereinigung vielleicht mehrmals laufen muss, um das Objekt zu löschen. Es gilt zwar, dass der Garbage-Collector aus dem Grund `finalize()` aufruft, weil das Objekt nicht mehr benötigt wird, es kann aber sein, dass die `finalize()`-Methode die `this`-Referenz nach außen gibt, sodass das Objekt wegen einer bestehenden Referenz nicht gelöscht werden kann und so zurück von den Toten geholt wird. Das Objekt wird zwar irgendwann entfernt, aber der Finalizer läuft nur einmal und nicht immer pro GC-Versuch.<sup>4</sup>

Löst eine Anweisung in `finalize()` eine Ausnahme aus, so wird diese ignoriert. Das bedeutet aber, dass die Finalisierung des Objekts stehen bleibt. Die automatische Speicherbereinigung beeinflusst das in ihrer 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 verschwinden darf. Wir müssen daher in unserer Implementierung `super.finalize()` aufrufen (es wäre gut, wenn der Compiler das wie beim Konstruktoraufzug immer automatisch machen würde). Leere `finalize()`-Methoden ergeben im Allgemeinen keinen Sinn, es sei denn, das `finalize()` der Oberklasse soll explizit übergangen werden:

**Listing 10.18** src/main/java/com/tutego/insel/object/SuperFont.java, `finalize()`

```
@Deprecated @Override
protected void finalize() throws Throwable {
    try {
        // ...
    }
    finally {
```

---

<sup>4</sup> Einige Hintergründe erfährt der Leser unter <http://www.iecc.com/gclist/GC-lang.html#Finalization>.

```

    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 mehr aufgerufen werden, denn in zukünftigen Versionen wird sie entfernt.

### Gültige Alternativen

Gedacht war die überschriebene Methode `finalize()`, um wichtige Ressourcen zur Not freizugeben, etwa File-Handles via `close()` oder Grafikkontexte des Betriebssystems, wenn ein Programmierer das vergessen hat. Alle diese Freigaben müssten eigentlich vom Entwickler angestoßen werden, und `finalize()` ist nur ein Helfer, derrettend eingreifen kann. Doch da die automatische Speicherbereinigung `finalize()` nur dann aufruft, wenn sie tote Objekte freigeben möchte, dürfen wir uns nicht auf die Ausführung verlassen. Gehen zum Beispiel die File-Handles aus, wird der Garbage-Collector nicht aktiv; es erfolgen keine `finalize()`-Aufrufe, und nicht mehr erreichbare, aber noch nicht weggeräumte Objekte belegen weiter die knappen File-Handles. Es muss also ein Mechanismus her, der korrekt ist und immer funktioniert. Hier gibt es zwei Ansätze:

1. try mit Ressourcen ruft automatisch die `close()`-Methode auf und gibt so Ressourcen frei.  
Der Nachteil ist, dass dann die Freigabe der Ressource an dem Aufruf von `close()` hängt; fehlt das Schließen, erfolgt immer noch keine Freigabe.
2. Die Klasse `java.lang.ref.Cleaner` hilft beim Aufräumen, dazu mehr in [Abschnitt 10.2, »Schwache Referenzen und Cleaner«](#).

#### 10.1.8 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.

## 10.2 Schwache Referenzen und Cleaner

Die Referenzen auf Objekte, die wir im Alltag um uns herum haben, heißen *starke Referenzen*, weil die automatische Speicherbereinigung niemals ein benutztes Objekt freigeben würde. Neben den starken Referenzen gibt es jedoch auch *schwache Referenzen*, die es dem GC erlauben, die Objekte zu entfernen. Was erst einmal verrückt klingt, wird dann interessant, wenn es um die Implementierung von Caching-Datenstrukturen geht; ist das Objekt im Cache, ist das schön und der Zugriff schnell – ist das Objekt nicht im Cache, dann ist das auch in Ordnung, und der Zugriff dauert etwas länger. Wir können schwache Referenzen also gut verwenden, um im Cache liegende Objekte aufzubauen, die die automatische Speicherbereinigung wegräumen darf, wenn es im Speicher knapp wird.

Schwache Referenzen interagieren also in einer einfachen Weise mit der automatischen Speicherbereinigung, und dafür gibt es im java.base-Modul im Paket `java.lang.ref` ein paar Typen. Es gibt spezielle Behälter, die ein Objekt referenzieren, aber von der automatischen Speicherbereinigung geleert werden können.

- ▶ `SoftReference<T>`. Ein Behälter für *softly reachable* Objekte. Die Objekte werden vom GC spät freigegeben, wenn die JVM kurz vor einem `OutOfMemoryError` steht.
- ▶ `WeakReference<T>`. Ein Behälter für *weakly reachable* Objekte. Die Objekte werden vom GC schon relativ früh beim ersten GC freigegeben.
- ▶ `PhantomReference<T>`. Ein Behälter, der immer leer ist, aber dazu dient, mitzubekommen, wenn der GC sich von einem Objekt trennt
- ▶ `Reference<T>`. Abstrakte Basisklasse für die »reference objects« von `PhantomReference`, `SoftReference`, `WeakReference`

Die Behälter selbst werden vom GC nicht entfernt und landen in einer `ReferenceQueue<T>`. Über die Queue lässt sich feststellen, welche `Reference`-Behälter leer sind und z. B. aus einer Datenstruktur entfernt werden können – leere Behälter sind nutzlos und können nicht recycelt werden.

Ein weiterer Typ im Paket ist `Cleaner`, der eine Alternative zur Finalisierung ist. Beim `Cleaner` lässt sich eine Operation (vom Typ `Cleaner.Cleanable`) anmelden, die immer dann aufgerufen wird, wenn die automatische Speicherbereinigung zuschlägt und das Objekt nicht mehr erreichbar ist. Intern greift die Klasse auf `PhantomReference` zurück.

### Beispiel

Lege einen Punkt an und registriere einen `Cleaner`. Rege danach durch den Gebrauch von richtig viel Speicher den GC an, der den `Cleaner` anstößt.



**Listing 10.19** src/main/java/com/tutego/insel/lang/ref/CleanerDemo, main()

```
Point p = new Point( 1, 2 );
Cleaner.create().register( p, () -> System.out.println( "Punkt ist weg!" ) );
p = null;
byte[] bytes = new byte[ (int) Runtime.getRuntime().freeMemory() ];
```

Die Methode register(Object obj, Runnable action) bekommt im ersten Parameter das zu beobachtende Objekt und dann ein Runnable, wobei immer dann die run()-Methode aufgerufen wird, wenn der Cleaner aufräumt. Mit unserem Bedarf an Speicher ist daher eine Konsoleausgabe »Punkt ist weg!« wahrscheinlich. Auf keinen Fall darf die Aufräumoperation p wieder referenzieren.

## 10.3 Die Utility-Klasse java.util.Objects

Die Klasse Objects hält einige statische Utility-Funktionen bereit. Sie führen in erster Linie null-Tests durch, um eine spätere NullPointerException beim Aufruf von Objektmethoden zu vermeiden.

### 10.3.1 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, die wir im nächsten Abschnitt kennenlernen werden.

#### 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 Hashwert. Die Implementierung ist einfach ein return Arrays.hashCode(values). Die Nutzung der Methode ist eher teuer durch den Aufbau des Varargs-Arrays und mögliche Boxing-Operationen für primitive Werte.

- 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 von der Reihenfolge der Parameter ab.

### Beispiel

[zB]

Erinnern wir uns an die überschriebene Methode hashCode() des Spielers, bei der der Spielername in den Hashwert eingehen soll:

**Listing 10.20** src/main/java/com/tutego/insel/object/hashcode/Player.java, hashCode(), Ausschnitt

```
result = 31 * result + ((name == null) ? 0 : name.hashCode());
```

Mit Objects.hashCode(Object) kann der null-Test entfallen, da er schon in der statischen Methode vorgenommen wird:

```
result = 31 * result + Objects.hashCode( name );
```

### 10.3.2 Objects.toString(...)

Eine weitere statische Methode ist Objects.toString(Object). Sie ist aus Symmetriegründen in der Klasse, da toString() zu den Standardmethoden 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

[K]

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.

### 10.3.3 null-Prüfungen mit eingebauter Ausnahmebehandlung

Bei den vorangehenden Methoden wird `null` als Sonderfall behandelt, und Ausnahmen werden vermieden. So sind etwa `Objects.toString(null)` oder `Objects.hashCode(null)` in Ordnung, und es wird um `null` »herumgearbeitet«. Das ist nicht immer sinnvoll, denn traditionell gilt es, `null` als Argument und in den Rückgaben zu vermeiden. Es ist daher gut, als Erstes in einem Methodenrumpf zu testen, ob die Argumente ungleich `null` sind – es sei denn, das ist unbedingt gewünscht.

Für diese Tests, dass Referenzen ungleich `null` sind, bietet `Objects` ein paar `requireNonNullXXX(...)`-Methoden, die `null`-Prüfungen übernehmen und im Fehlerfall eine `NullPointerException` auslösen. Diese Tests sind praktisch bei Konstruktoren oder Settern, die Werte initialisieren sollen, aber verhindern möchten, dass `null` durchgeleitet wird.



#### 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.
- `static <T> T requireNonNull(T obj, Supplier<String> messageSupplier)`  
Wie `requireNonNull(obj, message)`, nur kommt die Meldung aus dem `messageSupplier`. Das ist praktisch für Nachrichten, deren Aufbau teurer ist, denn der `Supplier` schiebt die Kosten für die Erstellung des Strings so lange hinaus, bis es wirklich zu einer `NullPointerException` kommt, denn erst dann ist die Meldung nötig.

- static <T> T requireNonNullElse(T obj, T defaultObj)  
Liefert das erste Objekt, das nicht null ist. defaultObj darf nicht null sein, sonst folgt eine NullPointerException. Implementiert als return (obj != null) ? obj : requireNonNull(defaultObj, "defaultObj");
- static <T> T requireNonNullElseGet(T obj, Supplier<? extends T> supplier)  
Liefert das erste Objekt, was nicht null ist. Ist obj gleich null, holt sich die Methode die Referenz aus dem Supplier, der dann kein null liefern darf, sonst folgt eine NullPointerException.

#### 10.3.4 Tests auf null

Hinter `isNull(Object o)` und `nonNull(Object o)` verbirgt sich ein einfacher Test auf `o == null` bzw. `o != null`.

```
class java.util.Objects
```

- static boolean isNull(Object obj)
- static boolean nonNull(Object obj)  
Liefert true, wenn obj gleich null bzw. nicht null ist, sonst false.

Im normalen Programmcode werden Entwickler diese Methoden nicht nutzen, doch sind sie praktisch für Methodenreferenzen, sodass es dann zum Beispiel heißen kann: `stream().filter(Objects::nonNull)` usw. Darauf kommen wir in [Kapitel 12](#) und in [Kapitel 17](#) noch einmal zu sprechen.

#### 10.3.5 Indexbezogene Programmargumente auf Korrektheit prüfen

Im Kapitel über Ausnahmen haben wir schon auf die Notwendigkeit hingewiesen, Wertebereiche zu prüfen und im Fehlerfall Ausnahmen wie `IllegalArgumentException` oder `IndexOutOfBoundsException` auszulösen, um keine falschen Werte in das Objekt zu lassen.

Drei weitere Methoden prüfen die gültigen Wertebereiche von indexbasierten Methoden und lösen im Fehlerfall eine `IndexOutOfBoundsException` aus.

```
class java.util.Objects
```

- static int checkIndex(int index, int length)
- static int checkFromToIndex(int fromIndex, int toIndex, int length)
- static int checkFromIndexSize(int fromIndex, int size, int length)



### Beispiel

Implementierung der get(int)-Methode in java.util.ArrayList:

```
public E get(int index) {
    Objects.checkIndex(index, size);
    return elementData(index);
}
```

## 10.4 Vergleichen von Objekten und Ordnung herstellen

Die Object-Methode equals(Object) gibt Auskunft darüber, ob zwei Objekte die gleichen Eigenschaften haben, es sagt aber nichts darüber aus, welches Objekt »größer« oder »kleiner« ist. Doch in vielen Anwendungen spielt die Ordnung von Objekten eine Rolle. Offensichtlich ist das bei der Sortierung, aber auch bei einfacheren Fragen, wie der nach dem größten oder kleinsten Element einer Sammlung. Sollen Objekte in Java verglichen werden, muss es immer eine Ordnung dieser Objekte geben. Das System wird nie selbstständig entscheiden können, und oftmals gibt es mehrere Kriterien. Warum ist zum Beispiel eine Person kleiner als eine andere Person? Weil die eine Person 1,50 Meter groß ist, die andere aber 1,80 Meter, oder weil es die eine Person beim Dschungelcamp bis ins Finale geschafft hat?<sup>5</sup>

### 10.4.1 Natürlich geordnet oder nicht?

In Java gibt es zwei unterschiedliche funktionale 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 so genannte 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.

<sup>5</sup> 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:

- ▶ BigDecimal, BigInteger, Byte, Character, Double, Float, Integer, Long, Short
- ▶ Date, Calendar, LocalTime, LocalDate
- ▶ String, StringBuilder, StringBuffer (die letzten beiden erst seit Java 11)
- ▶ File, URI
- ▶ Enum
- ▶ TimeUnit

Von Comparator finden wir in der API-Dokumentation nur java.text.Collator (und seine Unterklasse) vermerkt.

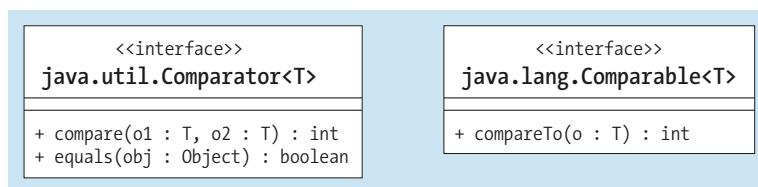


Abbildung 10.3 Vereinfachtes UML-Diagramm von Comparator und Comparable

#### Hinweis

Da es mit Comparator und Comparable **zwei Möglichkeiten** zur Definition einer Ordnung gibt, bietet die Java-API oftmals **zwei Methoden**, wenn sie eine Ordnung benötigen: einmal mit **einem Comparator** – dann ist keine Anforderung an die Elemente gestellt – und einmal ohne **Comparator** – dann müssen die Elemente jedoch die Schnittstelle Comparable implementieren.



### 10.4.2 compareXXX()-Methode der Schnittstellen Comparable und Comparator

Die funktionale Schnittstelle Comparable kommt aus dem `java.lang`-Paket und deklariert eine Methode `compareTo(...)`:

```
interface java.lang.Comparable<T>
```

- `int compareTo(T o)`

Vergleicht sich mit einem anderen Objekt.

Die funktionale Schnittstelle `Comparator` kommt aus dem Paket `java.util` (nicht wie `Comparable` aus `java.lang`) und deklariert eine abstrakte Methode:

```
interface java.util.Comparator<T>
```

- `int compare(T o1, T o2)`  
Vergleicht zwei Argumente auf ihre Ordnung.



#### Hinweis \*

Neben `compare(...)` deklariert `Comparator` auch das aus `Object` bekannte `boolean equals(Object obj)`. Die Methode muss nicht zwingend implementiert werden, da `Object` ja schon eine Implementierung bereitstellt. Die Methode steht nur deshalb in der Schnittstelle, damit eine API-Dokumentation erklärt, dass `equals(...)` nur testet, ob zwei `Comparator`-Objekte gleich sind.

### 10.4.3 Rückgabewerte kodieren die Ordnung

Der Rückgabewert von `compare(...)` beim `Comparator` bzw. `compareTo(...)` bei `Comparable` ist kleiner 0 (negativ), gleich 0 oder größer 0 (positiv) und bestimmt so die Ordnung der Objekte – das wird auch *Drei-Wege-Vergleich* (engl. *three-way comparison*) genannt.<sup>6</sup> Nehmen wir zwei Objekte `o1` und `o2` an, deren Klassen `Comparable` implementieren. Dann gilt folgende Übereinkunft:

<code>o1.compareTo( o2 ) &lt; 0</code>	↔	<code>o1</code> ist »kleiner als« <code>o2</code> .
<code>o1.compareTo( o2 ) == 0</code>	↔	<code>o1</code> ist »gleich« <code>o2</code> .
<code>o1.compareTo( o2 ) &gt; 0</code>	↔	<code>o1</code> ist »größer als« <code>o2</code> .

Ein externer `Comparator` (symbolisch `comp` genannt) verhält sich ähnlich:

<code>comp.compare( o1, o2 ) &lt; 0</code>	↔	<code>o1</code> ist »kleiner als« <code>o2</code> .
<code>comp.compare( o1, o2 ) == 0</code>	↔	<code>o1</code> ist »gleich« <code>o2</code> .
<code>comp.compare( o1, o2 ) &gt; 0</code>	↔	<code>o1</code> ist »größer als« <code>o2</code> .

<sup>6</sup> Programmiersprachen wie Perl, Groovy und Facebooks PHP-Dialekt Hack haben hierfür einen `<=>`-Operator. Da der wie ein Raumschiff aussieht, heißt er auch *spaceship operator*.



### Beispiel

LocalTime implementiert Comparable:

```
LocalTime elevenses = LocalTime.of( 11, 0 );
LocalTime lunchtime = LocalTime.of( 12, 0 );
System.out.println( elevenses.compareTo( lunchtime ) ); // -1
System.out.println( lunchtime.compareTo( elevenses ) ); // 1
```

#### 10.4.4 Beispiel-Comparator: den kleinsten 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; weniger Quadratmeter bedeuten, der Raum ist kleiner.

Die Raum-Klasse enthält für das kleine Demoprogramm einen parametrisierten Konstruktor, der sich die Quadratmeter merkt, und einen Getter.

**Listing 10.21** src/main/java/com/tutego/insel/util/Room.java, Room

```
public class Room {
    private int sqm;
    public Room( int sqm ) {
        this.sqm = sqm;
    }
    public int getSqm() {
        return sqm;
    }
}
```

Ein Raum, aufgebaut durch new Room(100), soll kleiner sein als new Room(1123). Dazu muss der Raum-Comparator auf die Quadratmeter zurückgreifen, andere Kriterien gibt es nicht:

**Listing 10.22** src/main/java/com/tutego/insel/util/RoomComparatorDemo.java, RoomComparator

```
class RoomComparator implements Comparator<Room> {
    @Override public int compare( Room room1, Room room2 ) {
        return Integer.compare( room1.getSqm(), room2.getSqm() );
    }
}
```



Abbildung 10.4 Der eigene Comparator muss nur compare(...) von der Schnittstelle überschreiben.

Alle Wrapper-Klassen haben `compare(...)`-Methoden für einen Drei-Wege-Vergleich. Das ist kürzer als Anweisungen wie:

```

int v1 = room1.getSqm(), v2 = room2.getSqm();
if ( v1 < v2 ) return -1;
else if ( v1 > v2 ) return 1;
return 0;
  
```



### Designtipp

Ein Comparator kann grundsätzlich einen Zustand haben, zum Beispiel für die Sprache, wenn etwa Zeichenfolgen mit im Vergleich stecken. Ein zustandsloser Comparator kann gut mit einem enum programmiert werden.

Es gibt in den Bibliotheken viele Stellen, die Ordnung benötigen, etwa bei der Suche nach dem maximalen Element oder Sortierung einer Liste. `Collections.max(...)` zum Beispiel findet das größte Element einer Liste, `Arrays.sort(...)` sortiert ein Array.

Beispiel: Mit dem `Comparator`-Objekt lässt sich ein Array mit Räumen sortieren, vorne steht dann der kleinste Raum:

**Listing 10.23** src/main/java/com/tutego/insel/util/RoomComparatorDemo.java, Ausschnitt

```
Room[] rooms = { new Room(1123), new Room(100), new Room(123) };
Arrays.sort( rooms, new RoomComparator() );
System.out.println( rooms[0].getSqm() ); // 100
```

Die Sortierungsmethode greift für die Raumpaare immer wieder auf den `Comparator` zurück und fragt nach der Ordnung. Wer in die `compare(...)`-Methode ein `println(...)` einbaut, kann dem Algorithmus bei der Arbeit zusehen.

#### 10.4.5 Tipps für Comparator und Comparable-Implementierungen

Sollen Objekte mit einem `Comparator` verglichen werden, aber `null`-Werte vorher aussortiert werden, so ist 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 von der Reihenfolge der Parameter ab.

Bei Implementierungen von `Comparable`: 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 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 Hashwerte 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. `null` ist in der Regel nicht größer/kleiner/gleich einem anderen mit `compareTo(...)` verglichenen Wert, daher ist eine Ausnahme die einzige vernünftige Reaktion.

Java unterstützt eine so genannte Serialisierung (siehe [Kapitel 19](#), »Einführung in Dateien und Datenströme«), bei der Zustände von komplexen Objekten seriell in einen Datenstrom geschrieben werden; aus diesen Daten lässt sich später ein Objekt rekonstruieren, wir sprechen von Deserialisierung. Serialisierbare Objekte implementieren die Schnittstelle `Serializable`. Ist ein `Comparator` mit einer Datenstruktur – wie dem `TreeSet` oder der `TreeMap` –

verbunden und soll die Datenstruktur serialisiert werden, muss auch die Comparator-Implementierung Serializable implementieren.

#### 10.4.6 Statische und Default-Methoden in Comparator

Die Schnittstelle Comparator bietet eine ganze Reihe von statischen und Default-Methoden. (In Comparable gibt es übrigens keine statischen oder Default-Methoden.) Besonders interessant sind die Möglichkeiten, mehrere Comparatoren zusammenzubinden.

Beginnen wir mit den einfach zu verstehenden Methoden:

```
interface java.util.Comparator<T>
```

- static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
 Liefert einen Comparator, der die natürliche Ordnung von Objekten verwendet, sprich, int compare(Comparable<Object> c1, Comparable<Object> c2) ist implementiert als return c1.compareTo(c2);
- static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
 Die statische Methode liefert einen Comparator, der wie naturalOrder() die natürliche Ordnung verwendet, sie allerdings umdreht. Entspricht Collections.reverseOrder(). Im Grunde ein Comparator mit einer compare(Comparable<Object> c1, Comparable<Object> c2)-Methode, die c2.compareTo(c1) liefert.
- default Comparator<T> reversed()
 Liefert für diesen aktuellen Comparator einen, der die Sortierreihenfolge umdreht. Entspricht Collections.reverseOrder(this).
- static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator) und
- static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)
 Liefert für einen gegebenen comparator einen neuen Comparator, der Vergleiche mit null erlaubt und null von der Ordnung her entweder vor oder hinter die anderen Werte setzt.

#### 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 dies 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. Genau für so eine Aneinanderreihung gibt es in Comparator eine nützliche Methode:

```
interface java.util.Comparator<T>
```

- default Comparator<T> thenComparing(Comparator<? super T> other)

Wendet erst den eigenen Comparator an, wenn er Zustände als gleich anzeigt, dann den anderen, other.

### Implementierung

Die Implementierung von thenComparing(...) ist einfach: Wenn der eigene Comparator 0 liefert, kommt der zweite übergebene Comparator ins Spiel. Das sieht im Grunde so aus:

```
default Comparator<T> thenComparing(Comparator<? super T> other) {
    return (Comparator<T>) (c1, c2) -> {
        int res = compare( c1, c2 );
        if ( res != 0 )
            return res;
        return other.compare( c1, c2 );
    };
}
```

Wir wollen diese thenComparing(...)-Methoden aus Comparator für ein Beispiel nutzen, das eine Liste nach Nach- und Vornamen sortiert.

**Listing 10.24** src/main/java/com/tutego/insel/util/ComparatorThenComparingDemo.java,  
Ausschnitt

```
public class ComparatorThenComparingDemo {

    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 =
    (p1, p2) -> p1.firstname.compareTo( p2.firstname );

public final static Comparator<Person> PERSON_LASTNAME_COMPARATOR =
    (p1, p2) -> 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" ) );

    persons.sort( PERSON_LASTNAME_COMPARATOR );
    System.out.println( persons );
    persons.sort( PERSON_FIRSTNAME_COMPARATOR );
    System.out.println( persons );
    persons.sort( PERSON_LASTNAME_COMPARATOR.thenComparing(
        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]
```

### Vergleichswert extrahieren und Vergleiche anstellen \*

Die verbleibenden Methoden in Comparator bieten alle die Spezialität, dass sie besondere Funktionsobjekte annehmen, die den »Schlüssel« für die Vergleiche extrahieren und dann für den Vergleich heranziehen. Mit der Syntax der Methodenreferenzen lassen sich sehr kompakte Comparator-Objekte formulieren.

Zu den Methoden:

```
interface java.util.Comparator<T>
```

- static <T,U> Comparator<T> comparing(Function<? super T,> extends U> keyExtractor, Comparator<? super U> keyComparator)
- static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,> ? extends U> keyExtractor)

- static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)
- static <T> Comparator<T> comparingLong(ToLongFunction<? super T> keyExtractor)
- static<T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor)
- default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,>? extends U> keyExtractor)
- default <U> Comparator<T> thenComparing(Function<? super T,>? extends U> keyExtractor, Comparator<? super U> keyComparator)
- default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)
- default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)
- default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor)

### Beispiel

[zB]

Das Beispiel mit den Raumvergleichen alternativ formuliert (wir greifen auf eine kompakte Syntax zu, die *Methodenreferenz* heißt; wir kommen in späteren Kapiteln noch einmal detailliert darauf zurück):

**Listing 10.25** src/main/java/com/tutego/insel/util/ComparatorDemo.java, Ausschnitt

```
Comparator<Room> comp = Comparator.comparingInt( Room::getSqm );
// kurz für Comparator.comparingInt( (Room r) -> r.getSqm() );
list.sort( comp );
```

Komplett ohne Implementierung eigener Comparator-Klassen kann dieser Einzeiler mithilfe der Extraktionsfunktionen nach Vor-/Nachnamen sortieren, unter der Voraussetzung, dass es zwei Getter gibt:

```
persons.sort( Comparator.comparing( Person::getLastname )
               .thenComparing( Person::getFirstname ) );
```

## 10.5 Wrapper-Klassen und Autoboxing

Die Klassenbibliothek bietet für jeden primitiven Datentyp wie int, double, char spezielle Klassen an. Diese so genannten *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 Daten-typ (Parsen).

- 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 eine spezielle Funktion eine Ganzzahl auf eine Fließkommazahl abbildet, ist eine Implementierung mit `Function<int, double>` nicht korrekt – es muss `Function<Integer, 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 10.2** 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 5](#), »Der Umgang mit Zeichenketten«, gewidmet, als es um Zeichen und Zeichenketten ging.

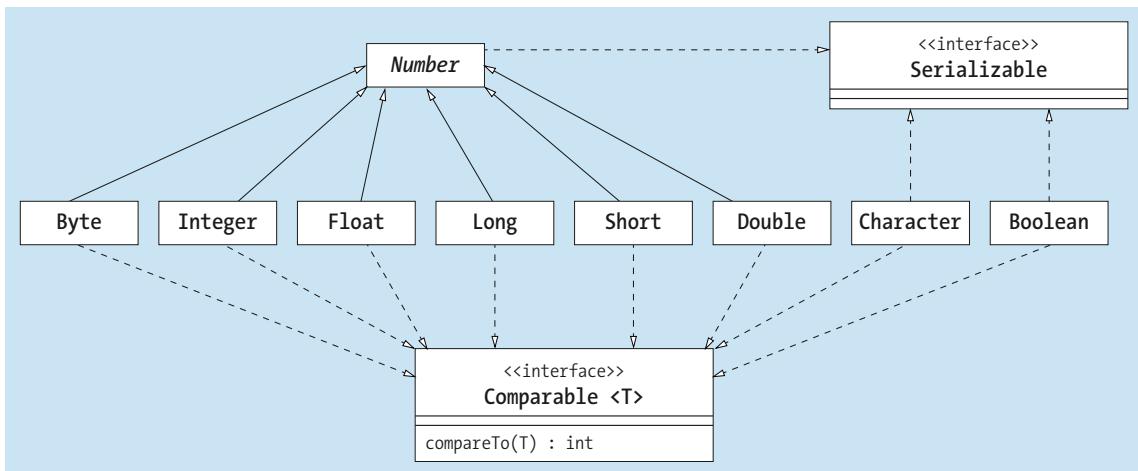


Abbildung 10.5 Essenzielle Typbeziehungen der Wrapper-Klassen

### 10.5.1 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, die jedoch seit Java 9 deprecated sind

#### Beispiel

Erzeuge einige Wrapper-Objekte:

```

Integer int1 = Integer.valueOf( "30" ); // valueOf()
Long lng1 = Long.valueOf( 0xC0B0L ); // valueOf()
Integer int2 = new Integer( 29 ); // Konstruktor (deprecated)
Long lng2 = new Long( 0xC0B0L ); // Konstruktor (deprecated)
Double dobl = new Double( 12.3 ); // Konstruktor (deprecated)
Boolean bool = true; // Boxing
Integer int3 = 42; // Boxing
  
```

[zB]

#### Konstruktoren sind deprecated

Lassen wir die veraltete Variante über den Konstruktor weg, bleiben zwei Möglichkeiten, an Wrapper-Objekte zu kommen. Boxing ist vom Schreibaufwand her gesehen die kürzeste und

im Allgemeinen die beste, weil kompakteste Variante (Boxing ist allerdings nicht ganz unproblematisch, wie [Abschnitt 10.5.13, »Autoboxing: Boxing und Unboxing«](#), zeigt). Da Boxing auf die `valueOf(...)`-Methoden zugreift, sind die beiden Varianten semantisch identisch und unterscheiden sich nur im Programmcode, aber nicht im Bytecode. Gegenüber einem Konstruktor eine statische Methode `valueOf(...)`-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 Aufrufer 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. 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 Umgebungsvariablen.

### 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 referenzieren:

```
Integer io = Integer.valueOf( 12 );
io = Integer.valueOf( io.intValue() + 1 );
```

Die Variable `io` referenziert zum Schluss ein zweites `Integer`-Objekt, und der Wert des ersten `io`-Objekts mit `12` bleibt unangetastet. Das entspricht im Wesentlichen auch der Nutzung von `String`. Heißt es `String s = "Co"; s = s + "ra";`, wird der Referenzvariablen `s` auch nur ein neues Objekt zugewiesen, aber der `String` an sich ändert sich nicht.

### 10.5.2 Konvertierungen in eine String-Repräsentation

Alle Wrapper-Klassen bieten statische `toString(value)`-Methoden zur Konvertierung des primitiven Elements in einen String an:

**Listing 10.26** src/main/java/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-Idiom<sup>7</sup> 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.

#### 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 Werts, den das Wrapper-Objekt speichert. Dass es gleichlautende statische Metho-

<sup>7</sup> 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.

den `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 `Integer.valueOf(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)`).

### 10.5.3 Von einer String-Repräsentation parsen

Die Wrapper-Klassen bieten statische `parseXXX(...)`-Methoden, die einen String in einen primitiven Datentyp konvertieren. Dem String kann ein Minus für negative Zahlen vorangestellt sein, auch ein Plus für positive Zahlen ist erlaubt. Die Methoden wurden bereits in [Abschnitt 5.7.2, »String-Inhalt in einen primitiven Wert konvertieren«](#), vorgestellt.

### 10.5.4 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`<sup>8</sup> die `xxxValue()`-Methoden<sup>9</sup>.

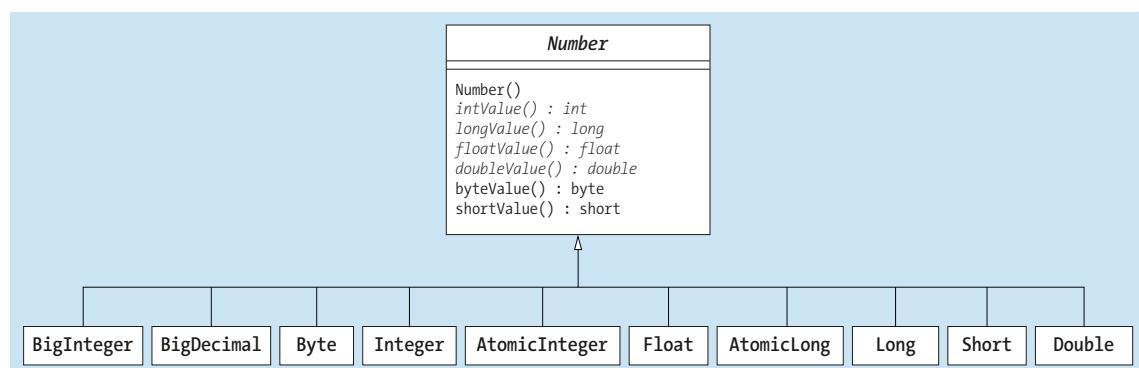


Abbildung 10.6 UML-Diagramm von Number

<sup>8</sup> Zusätzlich erweitern `BigDecimal` und `BigInteger` die Klasse `Number` und haben damit ebenfalls die `xxxValue()`-Methoden. `AtomicInteger` und `AtomicLong` erweitern ebenfalls `Number`, sind aber nicht immutable wie die restlichen Klassen.

<sup>9</sup> 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 Typumwandlung in `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.

### 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
```

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 sind natürlich genauso wie vorher, doch Entwickler könnten annehmen, dass der Compiler keine Konvertierung durchführt, sondern entweder das origi-

nale Integer- oder das Double-Objekt referenziert; das macht er aber nicht. Die Variable n1 referenziert hier ein Integer-unboxedes-double-konvertiertes-Double-boxed Objekt, und so sind die Referenzen von n1 und i 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
```

### 10.5.5 Vergleiche durchführen mit compareXXX(...), compareTo(...), equals(...) und Hashwerten

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 und positive 0 gibt. Damit das Vergleichen von zwei Werten immer nach dem gleichen Stil durchgeführt werden kann, gibt es zwei Sorten von Methoden in den Wrapper-Klassen, die uns sagen, ob zwei Werte kleiner, größer oder gleich sind:

- ▶ Auf der einen Seite findet sich die Objektmethode `compareTo(...)` in den Wrapper-Klassen. 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 zudem statische `compare(x, y)`- und teilweise `compareUnsigned(x, y)`-Methoden.

Die Rückgabe der Methoden ist ein `int`, und es kodiert, ob ein Wert größer, kleiner oder gleich ist.



#### Beispiel

Teste verschiedene Werte:

**Listing 10.27** src/main/java/com/tutego/insel/wrapper/CompareToDemo.java, main()

```
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 10.3 fasst die Methoden der Wrapper-Klassen zusammen:

Klasse	Methode aus Comparable	Statische Methode compareXXX(...)
Byte	int compareTo(Byte aByte)	int compare[Unsigned](byte x, byte y)
Short	int compareTo(Short aShort)	int compare[Unsigned](short x, short y)
Float	int compareTo(Float aFloat)	int compare(float f1, float f2)
Double	int compareTo(Double aDouble)	int compare(double d1, double d2)
Integer	int compareTo(Integer aInteger)	int compare(int x, int y)
Long	int compareTo(Long aLong)	int compare(long x, long y)
Character	int compareTo(Character aChar)	int compare[Unsigned](char x, char y)
Boolean	int compareTo(Boolean aBoolean)	int compare(boolean x, boolean y)

Tabelle 10.3 Methoden der Wrapper-Klassen

Die Implementierung einer statischen Methode `WrapperKlasse.compare(...)` ist äquivalent zu `WrapperKlasse.valueOf(x).compareTo(WrapperKlasse.valueOf(y))`, nur ist die zweite Variante langsamer und auch länger in der Schreibweise.

### Hinweis

Nur die genannten Wrapper-Klassen besitzen eine statische `compare(...)`-Methode. Es ist kein allgemeingültiges Muster, dass eine Klasse, wenn sie `Number` erweitert und `Comparable` implementiert, 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>	true
<code>Integer.valueOf( 1 ).equals( Integer.valueOf( 1 ) )</code>	true
<code>Integer.valueOf( 1 ).equals( Integer.valueOf( 2 ) )</code>	false
<code>Integer.valueOf( 1 ).equals( Long.valueOf( 1 ) )</code>	false
<code>Integer.valueOf( 1 ).equals( 1L )</code>	false

Es ist wichtig, zu wissen, dass der Parametertyp von `equals(...)` immer `Object` ist, aber die Typen identisch sein müssen, da andernfalls automatisch der Vergleich falsch ergibt. Das zeigen das vorletzte und das letzte Beispiel. Die `equals(...)`-Methode aus den Zeilen 4 und 5 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 `wrapObject.compareTo(anotherWrapperObject) == 0`.

### Ausblick

Dass die Wrapper-Klassen `equals(...)` implementieren, ist gut, denn so können Wrapper-Objekte problemlos in Datenstrukturen wie einer `ArrayList` untergebracht und wiedergefunden 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.

### Hashwerte

Der Hashwert eines Objekts bildet den Zustand auf eine kompakte Ganzzahl ab. Haben zwei Objekte ungleiche Hashwerte, so müssen auch die Objekte ungleich sein (mindestens, wenn die Berechnung korrekt ist). Zur Bestimmung des Hashwerts deklariert jede Klasse über die Oberklasse `java.lang.Object` die Methode `int hashCode()`. Alle Wrapper-Klassen überschreiben diese Methode. Zudem gibt es statische Methoden, sodass sich leicht der Hashwert berechnen lässt, ohne extra ein Wrapper-Objekt zu bilden.

Klasse	Klassenmethode	Objektmethode
<code>Boolean</code>	<code>static int hashCode(boolean value)</code>	<code>int hashCode()</code>
<code>Byte</code>	<code>static int hashCode(byte value)</code>	<code>int hashCode()</code>
<code>Short</code>	<code>static int hashCode(short value)</code>	<code>int hashCode()</code>

Tabelle 10.4 Statische Methode `hashCode(...)` und Objektmethoden im Vergleich

Klasse	Klassenmethode	Objektmethode
Integer	static int hashCode(int value)	int hashCode()
Long	static int hashCode(long value)	int hashCode()
Float	static int hashCode(float value)	int hashCode()
Double	static int hashCode(double value)	int hashCode()
Character	static int hashCode(char value)	int hashCode()

Tabelle 10.4 Statische Methode hashCode(...) und Objektmethoden im Vergleich (Forts.)

Zur Berechnung von Hashwerten und deren Bedeutung für Datenstrukturen lesen Sie [Abschnitt 10.1.5, »Hashwerte über hashCode\(\) liefern \\*«](#).

### 10.5.6 Statische Reduzierungsmethoden in Wrapper-Klassen

In den numerischen Wrapper-Klassen Integer, Long, Float und Double gibt es drei Methoden `sum(...)`, `max(...)` und `min(...)`, die genau das machen, was der Methodename verspricht.

```
final class java.lang.Integer|Long|Float|Double
extends Number
implements Comparable<Integer>
```

- static *Typ* `sum(Typ a, Typ b)`  
Bildet die Summe zweier Werte und liefert sie zurück. Entspricht einem einfachen `a + b`. Die Angabe *Typ* steht dabei für den entsprechenden primitiven Typ byte, short, int, long, float oder double, etwa in `int sum(int a, int b)`.
- static *Typ* `min(Typ a, Typ b)`  
Liefert das Minimum der zwei Zahlen.
- static *Typ* `max(Typ a, Typ b)`  
Liefert das Maximum der zwei Zahlen.

Des Weiteren bieten die Boolean-Klassen drei Methoden für drei boolesche Operationen:

```
final class java.lang.Boolean
implements Comparable<Boolean>, Serializable
```

- static boolean `logicalAnd(boolean a, boolean b)`  
Liefert `a && b`.

- static boolean logicalOr(boolean a, boolean b)  
Liefert  $a \mid\mid b$ .
- static boolean logicalXor(boolean a, boolean b)  
Liefert  $a \wedge b$ .

Die Methoden sind für sich genommen nicht spannend. Für die Summe (Addition) tut es genauso gut der `+`-Operator – er steckt sowieso hinter den `sum(...)`-Methoden –, und so wird keiner auf die Idee kommen, `i = Integer.sum(i, 1)` statt `i++` zu schreiben. Für das Maximum/Minimum bietet die `Math`-Klasse auch schon entsprechende Methoden `min(a, b)` und `max(a, b)`. Der Grund für alle genannten Methoden ist vielmehr, dass sie im Zusammenhang mit funktionaler Programmierung interessant sind und zwei Werte auf einen Wert reduzieren.

### 10.5.7 Konstanten für die Größe eines primitiven Typs

Alle Wrapper-Klassen besitzen eine `int`-Konstante `SIZE` für die Anzahl der Bits und eine `int`-Konstante `BYTES`, die mit der Größe des zugehörigen primitiven Datentyps belegt ist.

Wrapper-Klasse	Belegung SIZE	Belegung BYTES
Byte	8	1
Short	16	2
Integer	32	4
Long	64	8
Float	32	4
Double	64	8
Character	16	2

Tabelle 10.5 BYTES- und SIZE-Konstanten in den Wrapper-Klassen

Jedem Entwickler sind die Belegungen im Grunde klar, vielmehr sind die Konstanten für Tools gedacht.

### 10.5.8 Behandeln von vorzeichenlosen Zahlen \*

Alle Ganzahltypen (`char` ausgenommen) sind immer vorzeichenbehaftet, und Java wird wohl nie neue Datentypen für eine Unterscheidung einführen, wie es etwa C(++) bietet. Entwickler müssen damit leben, dass ein Byte immer zwischen  $-128$  und  $+127$  liegt und nicht zwischen 0 und 255. Doch wenn die Sprache keine neuen Typen einführt, kann hilft doch die Java-API mit Methoden in den Wrapper-Klassen.

### toUnsignedXXX(...)-Methoden

Der erste Typ von Methoden interpretiert ein Bit-Muster als vorzeichenlose Zahl. Ist zum Beispiel ein byte mit dem Bit-Muster 11111111 belegt, so steht dies nicht für 255, sondern für -1. Die statischen Konvertierungsmethoden `toUnsignedXXX(...)` der Wrapper-Klassen helfen dabei, das Bit-Muster ohne Vorzeichen-Bit zu interpretieren, wobei ein Wechsel zu einem nächsthöheren Datentyp nötig ist – das ist einleuchtend, denn im Fall von 11111111 kann ein byte wegen der Beschränkung auf -127 bis 128 die Zahl 255 nicht aufnehmen, int aber schon.

Byte	<code>static int toUnsignedInt(byte x)</code>
	<code>static long toUnsignedLong(byte x)</code>
Short	<code>static int toUnsignedInt(short x)</code>
	<code>static long toUnsignedLong(short x)</code>
Integer	<code>static long toUnsignedLong(int x)</code>

Tabelle 10.6 `toUnsignedXXX(...)`-Methoden in Byte, Short, Integer

Naheliegenderweise bietet Long so eine Methode nicht, denn es gibt keinen eingebauten Datentyp größer als long.

### toUnsignedString(...) und parseUnsignedString(...)

Was Long jedoch bietet, und auch Integer, sind Umwandlungen in String-Präsentationen und auch Parse-Methoden:

Integer	<code>static String toUnsignedString(int i)</code>
	<code>static String toUnsignedString(int i, int radix)</code>
	<code>static int parseUnsignedInt(String s)</code>
	<code>static int parseUnsignedInt(String s, int radix)</code>
Long	<code>static String toUnsignedString(long i)</code>
	<code>static String toUnsignedString(long i, int radix)</code>
	<code>static long parseUnsignedLong(String s)</code>
	<code>static long parseUnsignedLong(String s, int radix)</code>

Tabelle 10.7 String-Präsentationen von vorzeichenlosen Zahlen erstellen und parsen

Die `parseXXX(...)`-Methoden lösen bei falschem Format eine `NumberFormatException` aus.

## Vergleich, Division und Rest

Es gibt eine dritte Gruppe von Methoden, die eher mathematisch sind, und zwar statische Methoden zum Vergleichen, Dividieren und zur Restwertbildung:

Integer	static int compareUnsigned(int x, int y)
	static int divideUnsigned(int dividend, int divisor)
	static int remainderUnsigned(int dividend, int divisor)
Long	static int compareUnsigned(long x, long y)
	static long divideUnsigned(long dividend, long divisor)
	static long remainderUnsigned(long dividend, long divisor)

Tabelle 10.8 Mathematische Methoden

### 10.5.9 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.

Um aus dem String eine Zahl zu machen, nutzen wir `Integer.parseInt(String)`.

[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 [Kapitel 8](#), »Ausnahmen müssen sein« –, 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 5.7.2](#), »String-Inhalt in einen primitiven Wert konvertieren«, 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.

### 10.5.10 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 22](#), »Bits und Bytes, Mathematisches und Geld«, noch genauer auf die mathematischen Objekt- und Klassenmethoden zurück.

### 10.5.11 Die Long-Klasse

`Integer` und `Long` sind im Prinzip API-gleich, nur ist der kleinere Datentyp `int` durch `long` ersetzt.

### 10.5.12 Die Boolean-Klasse

Die Klasse `Boolean` kapselt den Datentyp `boolean`.

#### Boolean-Objekte aufbauen

Neben dem Konstruktor und Fabrikmethoden deklariert `Boolean` zwei Konstanten `TRUE` und `FALSE`.

```
final class java.lang.Boolean
implements Serializable, Comparable<Boolean>
```

- 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.
- static boolean parseBoolean(String s)
 

Parst den String und liefert entweder true oder false.

Der Konstruktor Boolean(String name) bzw. 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 liefern.



### Tipp

Würde jeder Entwickler ausschließlich die Konstanten Boolean.TRUE und Boolean.FALSE nutzen, so wären bei lediglich zwei Objekten Vergleiche mit == bzw. != 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.

### Boolean nach Ganzzahl konvertieren

Der primitive Typ boolean lässt sich nicht über eine Typumwandlung in einen anderen primitiven Typ konvertieren. Doch in der Praxis kommt es vor, dass true auf 1 und false auf 0 abgebildet werden muss; der übliche Weg ist:

```
int val = aBoolean ? 1 : 0;
```

Exotischer ist:

```
int val = Boolean.compare( aBoolean, false );
```

Noch exotischer ist Folgendes:

```
int val = 1 & Boolean.hashCode( aBoolean ) >> 1;
```

### 10.5.13 Autoboxing: Boxing und Unboxing

Autoboxing ist eine Eigenschaft von Java, bei der primitive Datentypen und Wrapper-Objekte bei Bedarf ineinander umgewandelt werden. Ein Beispiel:

```
int primInt = 4711;
Integer wrapInt = primInt; // steht für wrapInt = Integer.valueOf(primInt) (1)
primInt = wrapInt; // steht für primInt = wrapInt.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 primitives Element erwartet, aber ein Wrapper-Objekt vorhanden ist, entnimmt er den Wert mit einer passenden xxxValue(...)-Methode dem Wrapper. Der Compiler wird den Code automatisch generieren, vom Autoboxing ist später im Bytecode nichts mehr zu sehen.

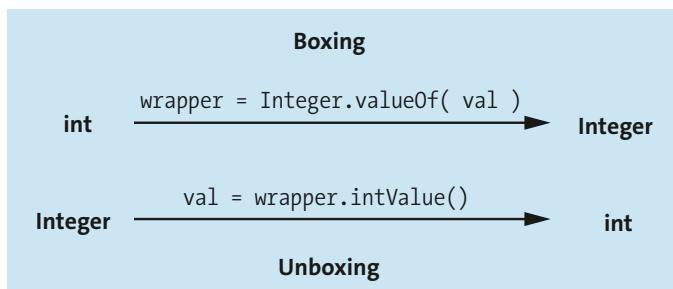


Abbildung 10.7 Autoboxing von int/Integer

### Die Operatoren ++, -- \*

Der Compiler konvertiert nach festen Regeln, und auch die Operatoren ++, -- sind erlaubt:

```
Integer i = 12;
i = i + 1; // (1)
i++;
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 nur andere Integer-Objekte für 13 und 14 referenziert.

### Boxing für dynamische Datenstrukturen (Ausblick)

Angenehm ist das Autoboxing, wenn in Datenstrukturen »primitive« Elemente abgelegt werden sollen:

```
ArrayList<Double> list = new ArrayList<Double>();
list.add( Math.sin(Math.PI / 4) );    // Boxing bei boolean add(Double)
double d = list.get( 0 );           // Unboxing bei Double get(int)
```

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. Das Autoboxing macht es jedenfalls etwas erträglicher.

### Keine Konvertierung der null-Referenz in 0

Beim Unboxing führt der Compiler bzw. die Laufzeitumgebung keine Konvertierung von null in 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 in int durch. Beim Unboxing gibt es aber die Gefahr einer NullPointerException:

```
Integer integer = null;
switch ( integer ) { } // ☠ NullPointerException zur Laufzeit
```

### Autoboxing bei Arrays?

Da primitive Datentypen und Wrapper-Objekte durch Autoboxing automatisch konvertiert werden, fällt im Alltag der Unterschied nicht so auf. Bei Arrays ist der Unterschied jedoch außergewöhnlich, 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 Array-Initialisierung 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 Array-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.<sup>10</sup>

### Mehr Probleme als Lösungen durch Autoboxing? (Teil 1) \*

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 10.28** src/main/java/com/tutego/insel/wrapper/Autoboxing.java, Ausschnitt

```
Double d1 = Double.valueOf( 1 );
Double d2 = Double.valueOf( 1 );

System.out.println( d1 >= d2 );    // true
System.out.println( d1 <= d2 );    // true
System.out.println( d1 == d2 );    // false
System.out.println( d1 == d2-0 ); // true
```

Der Vergleich mit == ist ein Referenzvergleich, und es findet kein Unboxing auf primitive Werte statt, sodass es auf einen Vergleich von primitiven Werten hinauslief. Wenn der Compiler zwei Referenzvariablen vor sich sieht, gibt es für Unboxing keinen Grund, auch wenn sie Wrapper-Typen sind. Daher muss bei zwei unterschiedlichen Integer-Objekten dieser Identitätsvergleich 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

---

<sup>10</sup> Die Lücke füllt etwa die Open-Source-Bibliothek *Apache Commons Lang* (<http://commons.apache.org/proper/commons-lang>) mit der Klasse `ArrayUtils`, die mit `toObject(...)` und `toPrimitive(...)` die Konvertierungen durchführt.

nicht. Mit dem Minus-null-Trick erreichen wir ein Unboxing, sodass wieder zwei Zahlen verglichen werden. Alternativ hilft eine explizite Typumwandlung in `int` oder ein Zugriff auf die Objektmethode `intValue()` zum bewussten Auspacken.

### Mehr Probleme als Lösungen durch Autoboxing? (Teil 2) \*

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 10.29** src/main/java/com/tutego/insel/wrapper/Autoboxing.java, Ausschnitt

```
Integer n1 = new Integer( 10 );
Integer n2 = Integer.valueOf( 10 );
Integer n3 = 10;    // Integer.valueOf( 10 );
Integer n4 = 10;    // Integer.valueOf( 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
```

In dem Beispiel vergleichen wir viermal die Identität der Wrapper-Objekte. Es fällt auf, dass einige Vergleiche falsch sind, was kein Wunder ist und durch `new` schnell erklärt ist, denn hier legt die Laufzeitumgebung immer neue Objekte an. Auffällig sind die identischen Objekte (`n2, n3, n4`), die beim Boxing durch das dahinterliegende `valueOf(int)` entstehen. Das Rätsel löst sich nur mit einem Blick in die von Oracle gewählte Implementierung auf, dass nämlich über Boxing gebildete `Integer`-Objekte einem Pool entstammen. Mit dem Pool versucht das JDK, das Problem mit dem `==` zu lösen, dass ausgewählte Wrapper-Objekte wirklich identisch sind und `==` sich bei Wrapper-Referenzen so verhält wie `==` bei primitiven Werten.

Da nicht beliebig viele Wrapper-Objekte aus einem Pool kommen können, gilt die Identität der über Boxing gebildeten Objekte nur in einem ausgewählten Wertebereich zwischen `-128` und `+127`, also dem Wertebereich eines Bytes:

**Listing 10.30** src/main/java/com/tutego/insel/wrapper/Autoboxing.java, Ausschnitt

```
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 Referenzvergleich 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 );
```

## 10.6 Iterator, Iterable \*

In Programmen spielen nicht nur einzelne Daten eine Rolle, sondern auch Sammlungen dieser Daten. Arrays sind zum Beispiel solche Sammlungen, aber auch Standarddatenstrukturen wie `ArrayList` oder `HashSet` oder Dateien. Eine Sammlung zeichnet sich hauptsächlich durch Methoden aus, die Daten hinzufügen, wieder entfernen und das Vorhandensein von Elementen prüfen. Natürlich hat jede dieser veränderbaren Datenstrukturen eine bestimmte API, doch im Sinne der guten objektorientierten Modellierung ist es wünschenswert, dieses Verhalten in Schnittstellen zu beschreiben. Zwei Schnittstellen treten besonders hervor:

- ▶ **Iterator:** Bietet eine Möglichkeit, Schritt für Schritt durch Sammlungen zu laufen.
- ▶ **Iterable:** Liefert so einen Iterator, ist also ein Iterator-Produzent.

### 10.6.1 Die Schnittstelle Iterator

Ein Iterator ist ein Datengeber, der über eine Methode verfügen muss, die das nächste Element liefert. Dann muss es eine zweite Methode geben, die Auskunft darüber gibt, ob der Datengeber noch weitere Elemente zur Verfügung stellt. Genau dafür deklariert die Java-API eine Schnittstelle `Iterator` mit zwei Operationen:

	Hast du mehr?	Gib mir das Nächste!
Iterator	hasNext()	next()

Tabelle 10.9 Zwei zentrale Methoden des Iterators



### Beispiel

Das Ablaufen (auf Neudeutsch »iterieren«) sieht immer gleich aus:

```
while ( iterator.hasNext() )
    process( iterator.next() );
```

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. Wahlfreien Zugriff haben wir nicht. Übergehen wir ein `false` von `hasNext()` und fragen trotzdem mit `next()` nach dem nächsten Element, bestraft uns eine `NoSuchElementException`.

`interface java.util.Iterator<E>`

- `boolean hasNext()`  
Liefert `true`, falls die Iteration weitere Elemente bietet.
- `E next()`  
Liefert das nächste Element in der Aufzählung und setzt die Position weiter. Es gibt eine `NoSuchElementException`, wenn keine weiteren Elemente mehr vorhanden sind.

Prinzipiell könnte die Methode, die das nächste Element liefert, auch per Definition `null` zurückgeben und so anzeigen, dass es keine weiteren Elemente mehr gibt, und auf `hasNext()` könnte verzichtet werden. Allerdings kann `null` dann kein gültiger Iterator-Wert sein, und das wäre ungünstig.

Die Schnittstelle `Iterator` erweitert selbst keine weitere Schnittstelle.<sup>11</sup> Die Deklaration ist generisch, da das, was der Iterator liefert, immer von einem bekannten Typ ist.

### Beim Iterator geht es immer nur vorwärts

Im Gegensatz zum Index eines Arrays können wir beim Iterator ein Objekt nicht noch einmal auslesen (`next()` geht automatisch zum nächsten Element), nicht vorspringen oder 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 müssten 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. Grundsätzlich entscheidet die Implementierung der Datenstruktur und des Iterators, in welcher Reihenfolge die Elemente herausgegeben wer-

---

<sup>11</sup> Konkrete Enumeratoren (und Iteratoren) können nicht automatisch serialisiert werden; die realisierenden Klassen müssen hierzu die Schnittstelle `Serializable` implementieren.

den. Sind die Daten sortiert, so wird auch der Iterator die Ordnung achten, die Dokumentation gibt Details.

### Hinweis

In Java steht der Iterator nicht auf einem Element, sondern zwischen Elementen. Die Methode `hasNext()` sagt, ob es ein nächstes Element gibt, und `next()` liefert es und setzt die interne Position weiter. Es gibt kein Konzept vom aktuellen Element, das immer wieder erfragbar ist; `next()` ist zustandsbehaftet. Zu Beginn der Iteration steht der Iterator vor dem ersten Element.



### Code auf verbleibenden Elementen eines Iterators ausführen

In der Schnittstelle `Iterator` gibt es die Default-Methode `forEachRemaining(Consumer<? super E> action)`, die ein beliebiges Stückchen Code – transportiert über einen `Consumer` – auf jedem Element ausführt. Die Implementierung der Default-Methode ist ein Dreizeiler:

**Listing 10.31** `java.util.Iterator.java, forEachRemaining()`

```
default void forEachRemaining( Consumer<? super E> action ) {
    Objects.requireNonNull( action );
    while ( hasNext() )
        action.accept( next() );
}
```

Mithilfe dieser Methode lässt sich eine externe Iteration über eine selbst gebaute Schleife in eine interne Iteration umbauen, und Lambda-Ausdrücke machen die Implementierung der Schnittstelle kurz – mehr zu Lambda-Ausdrücken in [Kapitel 12](#), »Lambda-Ausdrücke und funktionale Programmierung«.



### Beispiel

Gibt jedes Argument der Konsoleneingabe aus:

```
new Scanner( System.in ).forEachRemaining( System.out::println );
```

```
interface java.util.Iterator<E>
```

- `default void forEachRemaining(Consumer<? super E> action)`

Führt `action` auf jedem kommenden Element des Iterators bis zum letzten Element aus.



### Hinweis

Jede Collection-Datenstruktur liefert mit `iterator()` einen Iterator, auf dem dann wiederum ein Aufruf von `forEachRemaining(...)` möglich ist. Allerdings gibt es mit der Stream-API eine flexiblere Alternative zum Abarbeiten von Programmcode, die sich auch anbietet, wenn der Stream aus anderen Quellen kommt, wie Arrays.

### Optional: Elemente über Iterator löschen

Die Iterator-Methode `next()` ist eine reine Lesemethode und verändert die darunterliegenden Datenstruktur nicht. Doch bietet die Schnittstelle `Iterator` auch eine Methode `remove()`, die das zuletzt von `next()` gelieferte Objekt aus der Datensammlung entfernen kann. Da diese Operation nicht immer Sinn ergibt – etwa bei immutablen Datenstrukturen oder wenn ein Iterator zum Beispiel Dateien Zeile für Zeile ausliest –, ist sie in der API-Dokumentation als optional gekennzeichnet. Das heißt, dass ein konkreter Iterator keine Löschoperation unterstützen muss und etwa einfach nichts macht oder eine `UnsupportedOperationException` auslösen könnte.

```
interface java.util.Iterator<E>
```

- `default void remove()`

Löscht das zuletzt von `next()` gelieferte Objekt aus der darunterliegenden Sammlung. Die Operation muss nicht zwingend von Iteratoren angeboten werden und löst, falls nicht anderweitig überschrieben, eine `UnsupportedOperationException("remove")` aus.

### 10.6.2 Wer den Iterator liefert

Iteratoren spielen in Java eine sehr große Rolle und kommen im JDK tausendfach vor. Die Frage ist nur: Woher kommt ein Iterator, sodass sich eine Sammlung ablaufen lässt? Datengeber müssen dazu eine Methode anbieten.

#### 1. Beispiel

`iterator()` von `Path` liefert ein `Iterator<Path>` über die Pfad-Elemente. Es liefert

```
Iterator<Path> iterator = Paths.get( "/chris/brain/java/9" ).iterator();
while ( iterator.hasNext() )
    System.out.println( iterator.next() );
```

vier Zeilen mit »chris«, »brain«, »java« und »9«.

## 2. Beispiel

Datenstrukturen wie Listen und Mengen deklarieren ebenfalls eine iterator()-Methode:

```
Iterator<Integer> iter = new TreeSet<>( Arrays.asList( 4, 2, 9 ) ).iterator();
```

Die Ausgabe wäre mit der while-Schleife von oben sortiert 2, 4 und 9.

## 3. Beispiel

Die Klasse Scanner implementiert Iterator<String>, sodass das bekannte Paar von hasNext()/next() über die Tokens laufen kann:

```
Iterator<String> iterator = new Scanner( "Hund Katze Maus" );
```

Die Ausgabe besteht mit oberer Schleife aus drei Zeilen.

Im ersten und zweiten Fall ist es also der Aufruf von iterator(), der uns einen Iterator verschafft, im dritten Fall ist es eine Klasse, die selbst ein Iterator mit Konstruktor ist.

### 10.6.3 Die Schnittstelle Iterable

Eine Methode iterator(), die einen Iterator liefert, ist häufig bei Datenstrukturen anzutreffen. Das hat einen Grund: Die Klassen mit iterator()-Methode implementieren eine Schnittstelle java.lang.Iterable, und die schreibt die Operation iterator() vor. Das TreeSet, das wir im Beispiel verwendet haben, implementiert genauso Iterable, wie Path es auch implementiert.

```
interface java.lang.Iterable<T>
```

- Iterator<T> iterator()

Liefert einen java.util.Iterator, der über alle Elemente vom Typ T iteriert.

<<interface>> <b>Iterable&lt;T&gt;</b>
iterator() : Iterator<T> forEach(Consumer<? super T>) : void spliterator() : Spliterator<T>

<<interface>> <b>Iterator&lt;E&gt;</b>
hasNext() : boolean next() : E remove() : void forEachRemaining(Consumer<? super E>) : void

Abbildung 10.8 UML-Diagramm von Iterable und Iterator

### 10.6.4 Erweitertes for und Iterable

Bisher haben wir das erweiterte `for` für kleine Beispiele eingesetzt, in denen es darum ging, ein Array von Elementen abzulaufen:

**Listing 10.32** src/main/java/com/tutego/insel/iterable/Iterable.java, main()

```
for ( String s : new String[]{ "T. Noah", "S. Colbert", "J. Oliver" } )
    System.out.printf( "%s ist toll.%n", s );
```

Die erweiterte `for`-Schleife läuft nicht nur Arrays ab, sondern alles, was vom Typ `Iterable` ist. Da insbesondere viele Datenstrukturklassen diese Schnittstelle implementieren, lässt sich mit dem erweiterten `for` praktisch durch Ergebnismengen iterieren:

```
for ( String s : Arrays.asList( "T. Noah", "S. Colbert", "J. Oliver" ) )
    System.out.printf( "%s ist toll.%n", s );
```

### 10.6.5 Interne Iteration

Die Schnittstelle `Iterable` bietet zwei Methoden mit Default-Implementierung. Die interessante Methode ist `forEach(...)`, die Methode `spliterator()` ist an dieser Stelle nicht interessant.

```
interface java.util.Iterable<T>
```

- `default void forEach(Consumer<? super T> action)`

Holt den Iterator, läuft über alle Elemente und ruft dann den Konsumenten auf, der das Element übergeben bekommt.

Die Methode `forEach(...)` realisiert eine so genannte *interne Iteration*. Das heißt, nicht wir müssen eine Schleife mit dem Paar `hasNext()/next()` formulieren, sondern das macht `forEach(...)` für uns.



#### Beispiel

Laufe über die ersten Primzahlen und gib sie aus:

```
Arrays.asList( 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 )
    .forEach( e -> System.out.printf( "Primzahl: %d%n", e ) );
```

Der Aufruf `Arrays.asList(...)` liefert eine `java.util.List` und ist `Iterable`.

### 10.6.6 Einen eigenen Iterable implementieren \*

Damit unsere eigenen Objekte rechts hinter dem Doppelpunkt vom erweiterten `for` stehen können, muss die entsprechende Klasse die Schnittstelle `Iterable` implementieren und somit eine `iterator()`-Methode anbieten. `iterator()` muss einen passenden `Iterator` zurückgeben. Der wiederum muss die Methoden `hasNext()` und `next()` implementieren, die das nächste Element in der Aufzählung angeben und das Ende anzeigen. Zwar schreibt der `Iterator` auch `remove()` vor, doch das wird leer implementiert.

Unser Beispiel soll einen praktischen `Iterable` implementieren, der über Wörter eines Satzes geht. Als grundlegende Implementierung dient der `StringTokenizer`, der über `nextToken()` die nächsten Teilstücke liefert und über `hasMoreTokens()`, 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. Der Nachteil ist, dass es für ein `WordIterable` nicht mehrere unterschiedliche `Iterator`-Exemplare geben kann.

**Listing 10.33** src/main/java/com/tutego/insel/iterable/WordIterable.java, Ausschnitt

```
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();
}
}

```

Im Beispiel:

**Listing 10.34** src/main/java/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 Java-Compiler intern um zu:

```

{
    String word; Iterator<String> iterator = new WordIterable(s ).iterator();
    while ( iterator.hasNext() ) {
        word = iterator.next();
        System.out.println( word );
    }
    word = null;
    iterator = null;
}

```

## 10.7 Die Spezial-Oberklasse Enum

Jeder Aufzählungstyp erbt von der Spezialklasse `Enum`. Nehmen wir erneut die Wochentage:<sup>12</sup>

**Listing 10.35** src/main/java/com/tutego/insel/enums/Weekday.java, Ausschnitt

```

public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

```

Der Compiler übersetzt dies in eine Klasse, die etwa so aussieht:

```

public class Weekday extends Enum {

    public static final Weekday MONDAY = new Weekday( "MONDAY", 0 );
    public static final Weekday TUESDAY = new Weekday( "TUESDAY ", 1 );
    // weitere Konstanten ...
}

```

---

<sup>12</sup> Mit `java.time.DayOfWeek` deklariert die Java SE einen Aufzählungstyp für Wochentage inklusive Methoden, der in Produktivsoftware bevorzugt werden sollte. Wir bilden ihr nach, weil er so anschaulich ist.

```

private Weekday( String s, int i ) {
    super( s, i );
}

// weitere Methoden ...
}

```

### 10.7.1 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.

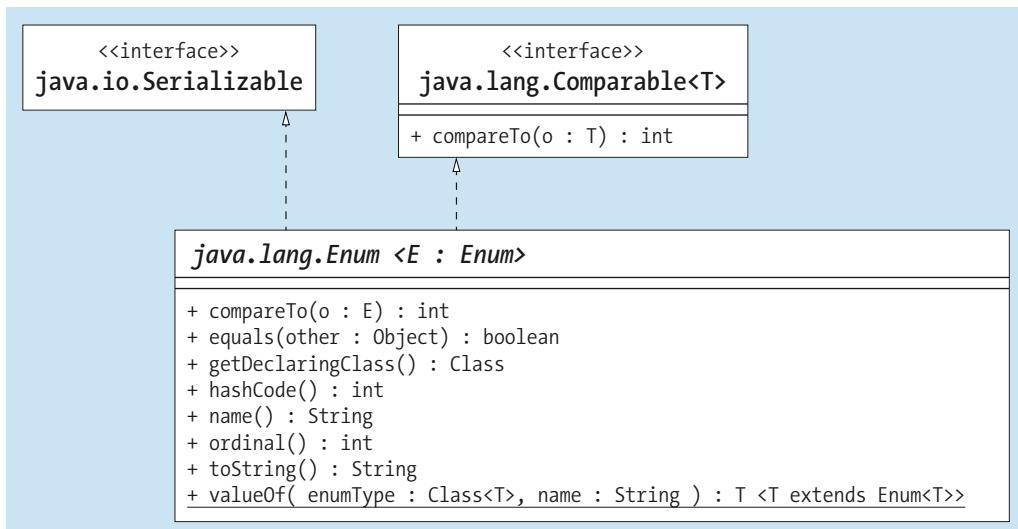


Abbildung 10.9 Typbeziehung von Enum

#### String-Repräsentation

Über die Methode `name()` liefert ein Enum-Objekt den Namen der Konstanten. 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(String)` 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)`.



### 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 Array von allen Aufzählungen vom Aufzählungstyp. Nützlich ist das für das erweiterte `for`, das alle Konstanten aufzählen soll. Eine Alternative mit dem gleichen Ergebnis ist die `Class`-Methode `getEnumConstants()`:

**Listing 10.36** src/main/java/com/tutego/insel/enums/WeekdayDemo.java, Ausschnitt

```
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 so genannte *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:

**Listing 10.37** src/main/java/com/tutego/insel/enums/WeekdayDemo.java, Ausschnitt

```
static int getOrdinal( String name ) {
    try {
        return Weekday.valueOf( name ).ordinal();
    }
    catch ( IllegalArgumentException e ) {
        return -1;
    }
}
```

Damit liefert unser `getOrdinal("MONDAY") == 0` und `getOrdinal("WOCENTAG") == -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.

### Beispiel

[zB]

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 Konstanten 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.
- `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, dass 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 von der Aufzählungsklasse zu einem konkreten `Enum`. Achtung: Die Methode liefert, auf der Aufzählungsklasse selbst angewendet, `null` und nur auf den Elementen der Aufzählung einen sinnvollen Wert. So wäre `Weekday.class.getDeclaringClass()` gleich `null`, aber `Weekday.MONDAY.getDeclaringClass()` wie gewünscht `com.tutego.weekday.Weekday`.



### Hinweis

Vom `Class`-Objekt ist die Methode `getEnumConstants()` noch interessant, denn auch sie gibt wie `values()` ein Array mit allen Einträgen zurück. Der Vorteil über das `Class`-Objekt ist jedoch, dass es generischer ist; der Aufruf der statischen `values()`-Methode ist immer mit der Klasse verbunden, `getEnumConstants()` funktioniert bei jedem `Class`-Objekt, und selbst wenn es keine Aufzählungsklasse repräsentieren sollte, ist die Rückgabe `null`.

## 10.7.2 Aufzählungen mit eigenen Methoden und Initialisierern \*

Da ein `enum`-Typ eine besondere Form der Klassendeklaration ist, kann er ebenso Attribute, Methoden, statische bzw. Objekt-Initialisierer und Konstruktoren deklarieren. Jede Aufzählung hat automatisch Methoden wie `name()` und `ordinal()`, und da können Entwickler auch eigene hinzufügen.

## Country mit zusätzlicher Klassenmethode

Ein Aufzählungstyp kann statische Methoden besitzen. Diese Methoden können auf statische Eigenschaften des Aufzählungstyps zugreifen und zum Beispiel Konstanten auswählen. `values()` ist so eine gegebene statische Methode, die ein Array aller Aufzählungselemente liefert.

### Beispiel

Deklariere eine Aufzählung `Country` und zwei statische Methoden, sodass `Country.getDefault()` GERMANY liefert und `Country.random()` ein Zufallsland:

```
public enum Country {
    GERMANY, UK, CHINA;
    public static Country getDefault() { return GERMANY; }
    public static Country random() { return values()[ (int)(Math.random()*3) ]; }
}
```

[zB]

## Zusätzliche statische Initialisierung

Es sind Blöcke der Art `static { ... }` im Rumpf eines Aufzählungstyps erlaubt. Lädt die Laufzeitumgebung einer Klasse, initialisiert sie der Reihe nach alle statischen Variablen bzw. führt die `static`-Blöcke aus. Die Aufzählungen sind statische Variablen und werden beim Laden initialisiert. Steht der statische Initialisierer hinter den Konstanten, so wird auch er später aufgerufen als die Konstruktoren, die vielleicht auf statische Variablen zurückgreifen wollen, die der `static`-Block initialisiert. Ein Beispiel:

```
public enum Country {
    GERMANY, UK, CHINA;

    {
        System.out.println( "Objektinitialisierer" );
    }

    static {
        System.out.println( "Klasseninitialisier" );
    }

    private Country() {
        System.out.println( "Konstruktor" );
    }
}
```

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

Die Ausgabe ist:

```
Objektinitialisierer
Konstruktor
Objektinitialisierer
Konstruktor
Objektinitialisierer
Konstruktor
Klasseninitialisier
GERMANY
```

Die Ausführung und Ausgabe hängt von der Reihenfolge der Deklaration ab, und jede Umsortierung führt zu einer Verhaltensänderung. Jetzt könnten Programmierer auf die Idee kommen, mögliche static-Blöcke an den Anfang zu setzen, vor die Konstanten. Meine Leser sollten das Ergebnis testen ... Auf Konstruktoren kommen wir gleich noch zu sprechen.

### Country mit zusätzlicher Objektmethode

Geben wir einer Aufzählung Country eine Methode, die den ISO-3166-2-Landescode des jeweiligen Aufzählungselements liefert:

**Listing 10.38** src/main/java/com/tutego/insel/enums/Country.java, Ausschnitt

```
public enum Country {

    GERMANY, UK, CHINA;

    public String getISO3Country() {
        switch ( this ) {
            case GERMANY : return "DEU";
            case UK       : return "GBR";
            default       : return "CHN";
        }
    }
}
```

Die Methode getISO3Country() kann nun auf der Aufzählung aufgerufen werden:

```
System.out.println( Country.CHINA.getISO3Country() ); // CHN
```

Die switch-Anweisung ist auf Aufzählungen erlaubt, das ist komfortabel. Schreiben wir ein kleines Demoprogramm:

**Listing 10.39** src/main/java/com/tutego/insel/enums/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" );
}
```

### 10.7.3 enum mit eigenen Konstruktoren \*

Neben der ersten Variante für getISO3Country() wollen wir eine zweite Implementierung nutzen und nun Konstruktoren hinzuziehen, um das gleiche Problem auf andere Weise zu lösen:

**Listing 10.40** src/main/java/com/tutego/insel/enums/Country2.java, Ausschnitt

```
public enum Country2 {

    GERMANY( "DEU" ),
    UK( "GBR" ),
    CHINA( "CHN" );

    private String iso3Country;

    Country2( String iso3Country ) {
        this.iso3Country = iso3Country;
    }

    public String getISO3Country() {
        return iso3Country;
    }
}
```

Bei der Deklaration der Konstanten wird in runden Klammern ein Argument für den Konstruktor übergeben. Der Konstruktor speichert den String in der internen Variablen iso3Country, auf die dann getISO3Country() Bezug nimmt.



### Hinweis

Konstruktoren von Aufzählungstypen sind immer automatisch privat und können auch keine andere Sichtbarkeit besitzen. Das ist logisch, denn die Konstruktoren sollen von außen nicht aufgerufen werden können. Die Methoden können durchaus unterschiedliche Sichtbarkeiten haben.

### enum mit überschriebenen Methoden

In dem Aufzählungstyp lassen sich nicht nur Methoden hinzufügen, sondern auch Methoden überschreiben. Beginnen wir mit einer lokalisierten und überladenen Methode `toString()`:

**Listing 10.41** src/main/java/com/tutego/insel/enums/WeekdayInternational.java, Ausschnitt

```
public enum WeekdayInternational {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

    @Override
    public String toString() {
        return toString( Locale.getDefault() );
    }

    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:

**Listing 10.42** src/main/java/com/tutego/insel/enums/WeekdayInternationalDemo.java, Ausschnitt

```
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 zu einer Referenzwährung, dem Euro, in Beziehung gesetzt werden; der Wechselkurs ist einfach 1:2:

**Listing 10.43** src/main/java/com/tutego/insel/enums/GameCurrency.java, Ausschnitt

```
public enum GameCurrency {

    EURO() {
        @Override public double convertTo( GameCurrency targetCurrency, double value ) {
            return targetCurrency == EURO ? value : value / 2;
        }
    },
    PONRODOLLAR() {
        @Override public double convertTo( GameCurrency targetCurrency, double value ) {
            return targetCurrency == PONRODOLLAR ? value : value * 2;
        }
    };

    public 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 10.44** src/main/java/com/tutego/insel/enums/GameCurrencyDemo.java, Ausschnitt

```
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 `Enum`, 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 Beispiel für ein `enum DefaultIcons` implementiert die Schnittstelle `Icon` für grafische Symbole. Da die Symbole alle die gleichen Ausmaße haben, sind die `Icon`-Operationen `getIconWidth()` und `getIconHeight()` immer gleich und werden nur einmal implementiert; die tatsächlichen `paintIcon(...)`-Implementierungen (die hier nur angedeutet werden) unterscheiden sich.

**Listing 10.45** src/main/java/com/tutego/insel/enums/DefaultIcons.java, Ausschnitt

```
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.

## 10.8 Annotationen in der Java SE

Wir haben Annotationen als benutzerdefinierte Modifizierer kennengelernt. Dabei ist uns `@Override` schon mehrfach über den Weg gelaufen. Die Annotation wird nur vom Compiler betrachtet und ist auch nur an Methoden gültig.

### 10.8.1 Orte für Annotationen

Annotationen lassen sich setzen an:

Deklarationen von

- ▶ Typen: Klassen, Schnittstellen, Aufzählungen, andere Annotationstypen
- ▶ Eigenschaften: Konstruktoren, Methoden, Attributen

Nutzung eines Typs bei:

- ▶ new
- ▶ Typumwandlung
- ▶ implements-Klausel
- ▶ throws-Klausel bei Methoden

Die Annotationen bei der Typnutzung nennen sich kurz *Typ-Annotationen*.

Java bringt einige Annotationstypen mit, doch die werden bisher ausschließlich für Deklarationen eingesetzt, wie das bekannte @Override. Vordefinierte Typ-Annotationen sind bisher in der Java SE nicht zu finden.

### 10.8.2 Annotationstypen aus java.lang

Das Paket java.lang deklariert fünf Annotationstypen, wobei uns @Override schon oft begleitet hat.

Annotationstyp	Wirkung
@Override	Die annotierte Methode überschreibt eine Methode aus der Oberklasse oder implementiert eine Methode einer Schnittstelle.
@Deprecated	Das markierte Element ist veraltet und sollte nicht mehr verwendet werden.
@SuppressWarnings	Unterdrückt bestimmte Compilerwarnungen.
@SafeVarargs	Besondere Markierung für Methoden mit variabler Argumentanzahl und generischem Argumenttyp
@FunctionalInterface	Für Schnittstellen, die nur genau eine (abstrakte) Methode besitzen

Tabelle 10.10 Annotationen aus dem Paket java.lang

Die fünf Annotationen haben vom Compiler bzw. Laufzeitsystem eine besondere Semantik. Die Java SE deklariert in anderen Paketen (wie dem java.lang.annotation- und javax.annotation-Paket) noch weitere Annotationstypen, doch die sind an dieser Stelle nicht relevant.

Dazu kommen spezielle technologiespezifische Annotationstypen wie für die XML-Objekt-Abbildung oder Webservice-Deklarationen.

### 10.8.3 @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

Die Methode `fubar()`<sup>13</sup> 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.

Auch über ein Javadoc-Tag kann ein Element als veraltet markiert werden. Ein Unterschied bleibt: Das Javadoc-Tag kann nur von Javadoc (oder einem anderen Doclet) ausgewertet werden, während Annotations auch andere Tools auswerten können.

### 10.8.4 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-Wert-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-Wert-Paaren

Tabelle 10.11 Annotationen mit und ohne zusätzliche Informationen

Klammern sind bei einer Marker-Annotation optional.

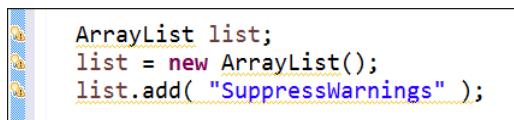
13 Im US-Militär-Slang steht das für: »Fucked up beyond any recognition« – »vollkommen ruiniert«.

### 10.8.5 @SuppressWarnings

Die Annotation `@SuppressWarnings` steuert Compilerwarnungen. Unterschiedliche Werte bestimmen genauer, welche Hinweise unterdrückt werden. Nützlich ist die Annotation bei der Umstellung von Quellcode, der vor Java 5 entwickelt wurde, denn 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 aktuellen Java-Compiler eine Vielzahl von Warnungen ausgibt. Nehmen wir folgenden Programmcode:

```
ArrayList list;
list = new ArrayList();
list.add( "SuppressWarnings" );
```

Eclipse zeigt die Meldungen direkt an, IntelliJ dagegen standardmäßig nicht.



```
ArrayList list;
list = new ArrayList();
list.add( "SuppressWarnings" );
```



Abbildung 10.10 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 keinen Typ 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.



### Beispiel

Der Compiler soll keine Meldungen für die Klasse geben:

**Listing 10.46** src/main/java/com/tutego/insel/annotation/SuppressWarningsAllWarnings.java

```
@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 mit @SuppressWarnings("all") 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.

### Beispiel

[zB]

Der Compiler soll für die ungenerisch verwendete Liste und ihre Methoden keine Meldungen geben:

```
@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" })`.

Neben den von Generics kommenden Kennungen `rawtypes` und `unchecked` gibt es weitere, die allerdings nicht sonderlich gut dokumentiert sind. Das liegt auch daran, dass Meldungen während der Programmübersetzung zur Compilerinfrastruktur gehören und nicht zur Laufzeitumgebung, und damit nicht zur traditionellen Java-API. Der Compiler kann im Prinzip beliebige Codeanalysen beliebiger Komplexität vornehmen und bei vermuteten Fehlern Alarm schlagen. Und wir dürfen auch nicht vergessen, dass es nur Warnungen sind: Wer als Programmierer alles richtig macht, wird die Meldungen nicht zu Gesicht bekommen. Den-

noch ist es relevant, sie zu kennen, denn der Compiler wird manches Mal etwas anmerken, was Entwickler bewusst nutzen wollen, und dann gilt es, die Meldungen abzuschalten.

Die Macher vom Eclipse-Compiler (JDT) dokumentieren die unterstützten Warnungen.<sup>14</sup> Neben den aufgeführten Meldungen all, rawtype und unchecked sind folgende interessant:

@SuppressWarnings	Unterdrückt Meldungen für
deprecation	veraltete Elemente, wie new java.util.Date(2012-1970, 3, 3)
incomplete-switch	ausgelassene Aufzählungen in switch-case-Anweisungen
resource	ein nicht geschlossenes AutoCloseable, wie in new java.util.Scanner(System.in).nextLine() <sup>15</sup>
serial	eine serialisierbare Klasse, die keine Serialisierungs-ID besitzt
unused	nicht benutzte Elemente, etwa nicht aufgerufene private Methoden

Tabelle 10.12 Einige Werte von @SuppressWarnings

## 10.9 Zum Weiterlesen

Die API-Dokumentation der Standardklassen aus dem `java.lang`-Paket ist sehr hilfreich und sollte komplett studiert werden.

<sup>14</sup> [https://help.eclipse.org/oxygen/index.jsp?topic=%2Org.eclipse.jdt.doc.user%2Ftasks%2Ftask-suppress\\_warnings.htm](https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-suppress_warnings.htm)

<sup>15</sup> `AutoCloseable` ist eine Schnittstelle, die Klassen implementieren, die über eine `close()`-Methode verfügen. [Abschnitt 8.6.2](#) stellt diese Schnittstelle vor.

# Kapitel 11

## Generics<T>

»Irdisches Glück heißt: Das Unglück besucht uns nicht zu regelmäßig.«  
– Karl Gutzkow (1811–1878)

### 11.1 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.

#### 11.1.1 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 Typintelligenz, 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 Fehler die JVM ins Verderben stürzen und zu einer `Exception` führen. Alles hat mit der expliziten Typumwandlung 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 Typumwandlung anweisen:

```
Object o = Integer.valueOf( 42 );           // oder mit Autoboxing: Object o = 42;
String s = (String) o;
```

Der Compiler akzeptiert die Typumwandlung, 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.

### 11.1.2 Raketen

In unseren vorangegangenen Beispielen drehte sich alles um Spieler und in einem Raum platzierte Spielobjekte. Stellen wir uns vor, der Spieler hat eine Rakete (engl. *rocket*), die etwas transportiert. Da nicht bekannt ist, was genau die Rakete trägt, 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 mit seiner Rakete transportieren kann:<sup>1</sup>

**Listing 11.1** src/main/java/com/tutego/insel/nongeneric/Rocket.java, Rocket

```
public class Rocket {
    private Object value;
    public Rocket() {}
    public Rocket( Object value ) { this.value = value; }
    public void set( Object value ) { this.value = value; }
    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 Rakete setzen und über die Zugriffsmethode `get()` wieder auslesen.

Geben wir einem Spieler eine rechte und eine linke Rakete, damit er genug Schub hat:

**Listing 11.2** src/main/java/com/tutego/insel/nongeneric/Player.java, Player

```
public class Player {
    public String name;
    public Rocket rightRocket;
    public Rocket leftRocket;
}
```

---

<sup>1</sup> Primitive Datentypen können über Wrapper-Objekte gespeichert werden.

Zusammen mit einem Spieler, der eine rechte und eine linke Rakete hat, ist ein Beispiel schnell geschrieben. Unser Spieler `michael` soll in beide Raketen Zahlen transportieren. Dann wollen wir sehen, in welcher Rakete er die größere Zahl versteckt hat.

```
Listing 11.3 src/main/java/com/tutego/insel/nongeneric/PlayerRocketDemo.java, main()

Player michael = new Player();
michael.name = "Omar Arnold";
Rocket rocket = new Rocket();
Long aBigNumber = 111111111111L;
rocket.set( aBigNumber );                                // (1)
michael.leftRocket = rocket;
michael.rightRocket = new Rocket( 2222222222222222L );

System.out.println( michael.name + " transportiert in der Rakete " +
                    michael.leftRocket.get() + " und " + michael.rightRocket.get() );

Long val1 = (Long) michael.leftRocket.get();    // (2)
Long val2 = (Long) michael.rightRocket.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 `Rocket` mit dem Typ `Object` zum Speichern der Raketeninhalte sehr allgemein deklariert wurde und alles aufnehmen kann:

- ▶ Beim Initialisieren wäre es gut, zu sagen, dass die Rakete 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 Rakete setzen und nichts anderes, etwa `Integer`-Objekte.
- ▶ Beim Entnehmen (2) des Raketeninhalts 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 Rakete auf jeden Fall ein `Long` speichert, dann könnte die Typumwandlung wegfallen, und der Programmcode wäre kürzer. Auch könnte uns der Compiler warnen, wenn wir versuchen würden, das `Long` als `Integer` aus der Rakete zu ziehen. Unser Wissen möchten wir gerne dem Compiler geben! Denn wenn in der Rakete ein `Long`-Objekt ist, wir es aber als `Integer` annehmen und eine explizite Typumwandlung 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 Typumwandlungen sind in der Regel unschön und sollten vermieden werden. Aber wie können wir die Raketen typsicher machen?

Eine Lösung wäre, eine neue Klasse für jeden in der Rakete zu speichernden Typ zu deklarieren, also einmal eine RocketLong für den Datentyp long, dann vielleicht RocketInteger für int, RocketString 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 Rakete 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*.<sup>2</sup> Sie bietet Entwicklern ganz neue Möglichkeiten, Datenstrukturen und Algorithmen zu programmieren, die von einem Datentyp unabhängig, somit *generisch* sind.

### 11.1.3 Generische Typen deklarieren

Wollen wir Rocket in einen *generischen Typ* umbauen, so müssen wir an den Stellen, an denen Object vorkam, einen Typstellvertreter, einen so genannten *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 Rocket ist folgende:

**Listing 11.4** src/main/java/com/tutego/insel/generic/Rocket.java, Rocket

```
public class Rocket<T> {
    private T value;
    public Rocket() {}
    public Rocket( 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 sie jetzt an Stelle von Object in der Rocket-Klasse.

Bei generischen Typen steht die Angabe der Typvariablen nur einmal zu Beginn der Klassendeklaration in spitzen Klammern hinter dem Klassennamen. Der Typparameter kann nun

---

<sup>2</sup> In C++ werden diese Typen von Klassen *parametisierte Klassen* oder *Templates* (Schablonen) genannt.

fast<sup>3</sup> 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

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 Typparameter aussieht:

```
public class Rocket<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 sind

Es gibt eine ganze Reihe von Beispielen, in denen Speicherstrukturen wie unsere Rakete 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 ein Element größer, kleiner oder gleich einem anderen ist, muss ein Algorithmus lediglich diese Ordnung 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* bzw. 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«.

#### 11.1.4 Generics nutzen

Um die neue `Rocket`-Klasse nutzen zu können, müssen wir sie zusammen mit einem Typargument angeben; es entstehen hier zwei *parametisierte Typen*:

---

<sup>3</sup> `T t = new T();` ist zum Beispiel nicht möglich.

**Listing 11.5** src/main/java/com/tutego/insel/generic/RocketPlayer.java, main(), Teil 1

```
Rocket<Integer> intRocket = new Rocket<Integer>();
Rocket<String> stringRocket = new Rocket<String>();
```

Der konkrete Typ steht immer hinter dem Klassen-/Schnittstellennamen in spitzen Klammern.<sup>4</sup> Die Rakete intRocket ist eine Instanz eines generischen Typs mit dem konkreten Typ-argument Integer. Diese Rakete kann jetzt offiziell nur Integer-Werte enthalten, und die Rakete stringRocket enthält nur Zeichenketten. Das prüft der Compiler auch, und wir benötigen keine Typumwandlung mehr:

**Listing 11.6** src/main/java/com/tutego/insel/generic/RocketPlayer.java, main(), Teil 2

```
intRocket.set( 1 );
int x = intRocket.get();           // Keine Typumwandlung mehr nötig
stringRocket.set( "Selbstzerstörungsauslösungsschalterhintergrundbeleuchtung" );
String s = stringRocket.get();
```

Der Entwickler macht so im Programmcode sehr deutlich, dass die Raketen 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 in [Abschnitt 11.1.5](#), »Diamonds are forever«, sehen werden, lässt sich das abkürzen.

Das Schöne an der Typsicherheit ist, dass alle Eigenschaften mit dem angegebenen Typ geprüft werden. Wenn wir z. B. aus intRocket 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 Typumwandlungen kürzer und lesbarer.

### Keine Primitiven

Typargumente können in Java Klassen, Schnittstellen, Aufzählungen oder Arrays davon sein, aber keine primitiven Datentypen. Das schränkt die Möglichkeiten zwar ein, doch da es Auto-boxing gibt, lässt sich damit leben. Und wenn null in der Rocket<Integer> liegt, führt ein Unboxing zur Laufzeit zur NullPointerException.

<sup>4</sup> 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
generischer Typ (engl. <i>generic type</i> )	Rocket<T>
Typparameter oder Typvariable (engl. <i>formal type parameter</i> )	T
parametrisierter Typ (engl. <i>parameterized type</i> )	Rocket<Long>
Typargument (engl. <i>actual type parameter</i> )	Long
Originaltyp (engl. <i>raw type</i> )	Rocket

**Tabelle 11.1** Zusammenfassung der bisherigen Generics-Begriffe

### Geschachtelte Generics

Ist ein generischer Typ wie Rocket<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 11.7** src/main/java/com/tutego/insel/generic/RocketPlayer.java, main(), Teil 3

```
Rocket<Rocket<String>> rocket0fRockets = new Rocket<Rocket<String>>();
rocket0fRockets.set( new Rocket<String>() );
rocket0fRockets.get().set( "Inner Rocket<String>" );
System.out.println( rocket0fRockets.get().get() ); // Inner Rocket<String>
```

Hier enthält die Rakete eine »Innenrakete«, die eine Zeichenkette "Inner Rocket<String>" speichert. Bei Dingen wie diesen ist schnell offensichtlich, wie hilfreich Generics für den Compiler (und uns) sind. Ohne Generics sähen eben alle Raketen gleich aus.

Präzise mit Generics	Unpräzise ohne Generics
Rocket<String> stringRocket;	Rocket stringRocket;
Rocket<Integer> intRocket;	Rocket intRocket;
Rocket<Rocket<String>> rocket0fRockets;	Rocket rocket0fRockets;

**Tabelle 11.2** Präzisierung durch Generics

Nur ein gut gewählter Name und eine präzise Dokumentation können bei nichtgenerisch deklarierten Variablen helfen. Vor der Einführung von Generics behelfen sich Entwickler damit, mithilfe eines Blockkommentars Typinformationen anzudeuten, zum Beispiel in Rocket/\*<String>\*/ stringRocket.

### Keine Arrays von parametrisierten Typen

Die folgende Anweisung bereitet eine einzige Rakete mit einem Array von Strings vor:

```
Rocket<String[]> rocketForArray = new Rocket<String[]>();
```

Aber lässt sich auch ein Array von mehreren Raketen, die jeweils Strings enthalten, deklarieren? Ja. Doch während die Deklaration noch möglich ist, ist die Initialisierung ungültig:

```
Rocket<String[]>[] arrayOfRocket;
arrayOfRocket = new Rocket<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<Rocket<String>>` sowieso sexyer als ein `Rocket<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 Compilermeldung zu unterdrücken.

### 11.1.5 Diamonds are forever

Bei der Initialisierung einer Variablen, deren Typ generisch ist, fällt auf, dass das Typargument zweimal angegeben werden muss. Bei geschachtelten Generics ist die Mehrarbeit unangenehm. Nehmen wir eine Liste, die Maps enthält, wobei der Assoziativspeicher Datums-werte mit Strings verbindet:

```
List<Map<Date, String>> listOfMaps;
listOfMaps = new ArrayList<Map<Date, String>>();
```

Das Typargument `Map<Date, String>` steht einmal auf der Seite der Variablen-deklaration und einmal hinter dem Schlüsselwort `new`.

### Der Diamantoperator

Verfügt der Compiler über alle Typinformationen, so können hinter `new` die generischen Typargumente entfallen, und es bleibt lediglich ein Pärchen spitzer Klammern:

[zB]

#### Beispiel

Statt

```
List<Map<Date, String>> listOfMaps = new ArrayList<Map<Date, String>>();
```

ist möglich:

```
List<Map<Date, String>> listOfMaps = new ArrayList<>();
```

Dass der Compiler die Typen aus dem Kontext ableiten kann, geht auf eine Compilereigenschaft zurück, die *Typ-Inferenz* (engl. *type inference*) heißt – sie wird uns noch einmal über den Weg laufen. 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 das gesamte Typargument `Map<Date, 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<> listOfTypeMaps;` 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 im folgenden Abschnitt anschauen – immer dort eingesetzt werden, wo Exemplare gebildet werden. Das nächste Nonsense-Beispiel zeigt vier Einsatzgebiete:

```
import java.util.*;

public class WhereToUseTheDiamond {

    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/Parametervariablen
- ▶ als Argument bei Methoden-/Konstruktoraufufen
- ▶ bei Methodenrückgaben

Ohne Frage sind der erste und zweite Fall die sinnvollsten. 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 mitten in einer Methode eine Datenstruktur mit `list = new ArrayList<>()` initialisiert wird, aber die Variablen-deklaration nicht auf der gleichen Bildschirmseite liegt, ist mitunter für den Leser nicht sofort sichtbar, was denn genau für Typen in der Liste sind.<sup>5</sup>

### Diamant nicht immer möglich

Es gibt Situationen, in denen die Typableitung nicht so funktioniert wie erwartet. Oftmals hat das mit dem Einsatz des Diamanten bei Methodenaufrufen oder aufeinander aufbauenden Aufrufen zu tun, sodass anzuraten ist – auch schon aus Gründen der Programmverständlichkeit –, auf Diamanten zu verzichten.



#### Beispiel

Für den Compiler ist das ein unlösbarer Fall:

```
List<String> list = new ArrayList<>().subList( 0, 1 );
```

Die Typ-Inferenz ist komplex,<sup>6</sup> und glücklicherweise muss sich ein Entwickler nicht um die interne Arbeitsweise kümmern. Wenn der Diamant wie im Beispiel nicht möglich ist, weil der Compiler ein »Type mismatch: cannot convert from List<Object> to List<String>« meldet, löst eine explizite Typangabe das Problem, also `new ArrayList<String>().subList(0, 1)`.

<sup>5</sup> Um den Diamanten zu testen, haben die Entwickler ein Tool geschrieben, das durch das JDK läuft und schaut, welche generisch genutzten `new`s durch den Diamanten vereinfacht werden könnten. (Heraus kamen etwa 5.000 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.

<sup>6</sup> Als dieses Feature in Java 7 eingebaut wurde, standen zwei Algorithmen zur Typauswahl zur Auswahl: simpel und komplex. Der komplexe Ansatz bezieht neben den Typeninformationen, die eine Zuweisung liefert, noch den Argumenttyp 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.

### Diamant vs. var

Diamant und var haben vergleichbare Aufgaben, unterscheiden sich aber durch die Quelle der Informationen. Beim Diamanten ist es zum Beispiel bei einer Zuweisung die linke Seite, die dem Compiler die Information gibt, was auf der rechten Seite der Zuweisung für ein Typ gemeint ist. Bei var wiederum ist das anderes herum: die rechte Seite hat den Kontext, und daher kann links der Variablenotyp entfallen:

```
List<String> list1 = new ArrayList<>(); // List<String>
var list2 = new ArrayList<String>();      // ArrayList<String>
var list3 = new ArrayList<>();            // ArrayList<Object>
```

Im letzten Fall gibt es keinen Compilerfehler, nur ist eben nichts bekannt über das Typargument, und daher gilt Object.

Um Code abzukürzen haben wir damit zwei Möglichkeiten: var oder Diamant.

### 11.1.6 Generische Schnittstellen

Eine Schnittstelle kann genauso als generischer Typ deklariert werden wie eine Klasse. Werfen wir einen Blick auf die Schnittstelle `java.lang.Comparable` und einen Ausschnitt von `java.util.Set` (Schnittstelle, die Operationen für Mengenoperationen vorschreibt, mehr dazu in [Kapitel 17, »Einführung in Datenstrukturen und Algorithmen«](#)).

Schnittstelle Comparable	Schnittstelle Set
<pre>public interface Comparable&lt;T&gt; {     int compareTo(T o); }</pre>	<pre>public interface Set&lt;E&gt; extends Collection&lt;E&gt; {     boolean add(E e);     int size();     boolean isEmpty();     boolean contains(Object o);     Iterator&lt;E&gt; iterator();     Object[] toArray();     &lt;T&gt; T[] toArray(T[] a);     ... }</pre>

Tabelle 11.3 Generische Deklaration der Schnittstellen Comparable und Set

Wie bekannt, greifen die Methoden auf die Typvariablen `T` und `E` zurück. Bei `Set` ist außerdem zu erkennen, dass sie selbst eine generisch deklarierte Schnittstelle erweitert.

Beim Einsatz von generischen Schnittstellen lassen sich die folgenden zwei Benutzungsmuster ableiten:

- ▶ Ein nichtgenerischer Typ löst Generics bei der Implementierung auf.
- ▶ Ein generischer Klassentyp implementiert eine generische Schnittstelle und gibt die Parametervariable weiter.

### Nichtgenerischer Typ 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 das Typargument ist genau der Typ vom Wrapper:

**Listing 11.8** java/lang/Integer.java, Ausschnitt

```
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.



#### Tipp

Komplexe generische Typen lassen sich gut durch eigene Typdeklarationen vereinfachen. Anstatt zum Beispiel immer wieder `HashMap<String, List<Integer>>` zu schreiben, lässt sich eine Abkürzung nehmen:

```
class StringToIntListMap extends HashMap<String, List<Integer>> {}
```

### 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 Typargument 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 das Typargument `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.

### 11.1.7 Generische Methoden/Konstruktoren und Typ-Inferenz

Die bisher genannten generischen Konstruktionen sahen im Kern wie folgt aus:

- ▶ `class Klassenname<T> { ... }`
- ▶ `interface Schnittstellenname<T> { ... }`

Eine an der Klassen- oder Schnittstellendeklaration angegebene Typvariable kann in allen nichtstatischen Eigenschaften des Typs angesprochen werden.



## Beispiel

Folgendes führt zu einem Fehler:

```
class Rocket<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 Typparameter deklarieren. Sie stehen dann nicht mehr an der Klasse, sondern an der Methoden-/Konstruktordeklaration und sind »lokal« für die Methode bzw. den Konstruktor. Das allgemeine Format ist:

Modifizierer <**Typvariable(n)**> Rückgabetyp `Methodenname(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 11.9** src/main/java/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 werden. 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 Methode:

```
interface Lupilu<T> { <T> void sit( T val ); } // Verwirrend
interface 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 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
random("Essen", 1)	String, Integer	Object, Serializable, Comparable
random(1L, 1D)	Long, Double	Object, Number, Comparable
random(new Point(), new StringBuilder())	Point, StringBuilder	Object, Serializable, Cloneable

Tabelle 11.4 Gemeinsame Basistypen

Es fällt auf, aber überrascht nicht, dass `Object` immer in die Gruppe gehört.

Die Schnittmenge der Typen bilden 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 Diamanttyp kürzt Variablen Deklarationen mit Initialisierung einer Referenzvariablen angenehm ab. Muss bei

```
Rocket<String> r = new Rocket<String>();
```

der generische Typ `String` zweimal angegeben werden, so erlaubt der Diamant (`<>`) Folgendes:

```
Rocket<String> r = new Rocket<>();
```

Mit der Typ-Inferenz gibt es eine alternative Lösung, falls wir selbst Erzeugermethoden bauen. Geben wir unserer Klasse `Rocket` eine Fabrikmethode

```
public static <T> Rocket<T> newInstance() {
    return new Rocket<T>();
}
```

so ist folgende Alternative möglich:

```
Rocket<String> r = Rocket.newInstance();
```

Aus dem Ergebnistyp Rocket<String> leitet der Compiler das Typargument String für die Rakete ab. Und ist bei einem einfachen Typ wie String die Schreibersparnis noch gering, wird der Code bei verschachtelten Datenstrukturen kürzer. Soll die Rakete einen Assoziativspeicher aufnehmen, der eine Zeichenkette mit einer Liste von Zahlen assoziiert, so schreiben wir kompakt:

```
Rocket<Map<String, List<Integer>>> r = Rocket.newInstance();
```

### Generische Methoden mit explizitem Typargument \*

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 hasRocket = true;
Rocket<String> rocket = hasRocket ? Rocket.newInstance() : null;
```

Der Eclipse-Compiler meldet "Type mismatch: cannot convert from Rocket<Object> to Rocket <String>".

Die Lösung: Wir müssen bei Rocket.newInstance() das Typargument String explizit angeben:

```
Rocket<String> rocket = hasRocket ? Rocket.<String>.newInstance() : null;
```

Die Syntax ist etwas gewöhnungsbedürftig, doch in der Praxis ist die explizite Angabe selten nötig.

Ein Beispiel: Ist das Argument der statischen Methode Arrays.asList(...) ein Array, dann ist das explizite Typargument nötig, da der Compiler nicht erkennen kann, ob das Array selbst das eine Element der Rückgabeklasse ist oder ob das Array die Vararg-Umsetzung ist und alle Elemente des Arrays 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" } );
// 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 Array um und übergibt das Array der `asList(...)`-Methode. Im Bytecode sehen daher die Aufrufe gleich aus. Bei `list21` und `list22` ist das Typargument jeweils explizit angegeben, aber nicht wirklich nötig, da ja das Ergebnis wie `list11` bzw. `list12` ist. Doch das Typargument `String` macht deutlich, dass die Elemente im Array, 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 Array das erste Element des vom Compiler automatisch aufgebauten Vararg-Arrays sein könnte (dann wäre es ein Array im Array) oder – und das ist die interne Standardumsetzung – der Java-Compiler das übergebene Array als die Vararg-Umsetzung interpretiert. Diese Doppeldeutigkeit löst `<String[]>`, da in dem Fall klar ist, dass das von uns aufgebaute `String`-Array das einzige Element eines neuen Vararg-Arrays sein muss. Und `Arrays.<String[]> asList(...)` stellt heraus, dass der Typ der Array-Elemente `String[]` ist. Daher funktioniert auch die letzte Variablen Deklaration nicht, denn bei `asList("A", "B")` ist der Elementtyp `String`, aber nicht `String[]`.

## 11.2 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.

### 11.2.1 Realisierungsmöglichkeiten

Im Allgemeinen gibt es zwei Möglichkeiten, 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 *Codespezialisierung*.
- ▶ **Homogene Übersetzung:** Aus der parametrisierten Klasse wird eine Klasse erzeugt, die an Stelle des Typparameters nur zum Beispiel `Object` einsetzt. Für das konkrete Typargument setzt der Compiler Typumwandlungen in die Anweisungen.

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.

### 11.2.2 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.<sup>7</sup>

Mit Generics	Nach der Typlösung
<pre>public class Rocket&lt;T&gt; {     private T value;     public void set( T value ) {         this.value = value; }      public T get() { return vale; } }</pre>	<pre>public class Rocket {     private Object value;     public void set( Object value ) {         this.value = value; }      public Object get() { return value; } }</pre>

Tabelle 11.5 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>Rocket&lt;Integer&gt; r = new Rocket&lt;Integer&gt;( 1 ); r.set( 1 ); Integer i = r.get();</pre>	<pre>Rocket r = new Rocket( 1 ); r.set( 1 ); Integer i = (Integer) r.get();</pre>

Tabelle 11.6 Nutzung generischer Klassen und wie es nach der Typlösung aussieht

Beim Herausholen über `get()` fügt der Compiler genau die explizite Typumwandlung 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 Information abgespeichert ist, ob 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 Disassembler `javap` und dem Schalter `-verbose` lassen sich diese Informationen anzeigen.

<sup>7</sup> Sind *Bounds* im Spiel – eine Typeinschränkung, die später noch vorgestellt wird –, wird ein präziserer Typ statt `Object` genutzt.

### Das große Ziel: Interoperabilität

Interoperabilität stand bei der Einführung der Generics ganz oben auf der Wunschliste. Zwei wichtige Anforderungen waren:

- ▶ Die mit Generics deklarierten Typen – wie `List<E>` – müssen auf jeden Fall noch für alten Programmcode, der zum Beispiel mit einem Java 1.4-Compiler erzeugt wurde, nutzbar sein. Das funktioniert so, dass generisch deklarierte Klassen im Bytecode für einen »alten« Compiler oder eine »alte« Laufzeitumgebung 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 das Typargument 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 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 beliebiges Typargument 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 Typumwandlungen unterscheiden. 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 Codespezialisierung 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 optimale Programmcode zu generieren. Das ist funktionale Programmierung mit einem Compiler ...

### 11.2.3 Probleme 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 Übersetzungs vorgang genauso aus wie unter Java 1.4 und sind

damit voll kompatibel. Sonst aber stellt die Typlöschung ein riesiges Problem dar, weil die Typinformationen zur Laufzeit nicht vorhanden sind.<sup>8</sup>

### Reified Generics

Generische Parameter sind nicht zur Laufzeit zugänglich. Vielleicht kommt das irgendwann, in Java X, Java 2020 ... Das Stichwort dazu ist *Reified Generics*, also generische Informationen, die auch zur Laufzeit komplett zugänglich sind.

### Kein new T()

Da durch die Typlöschung bei Deklarationen wie `Rocket<T>` die Parametervariable durch `Object` ersetzt wird, lässt sich zum Beispiel in der Rakete *nicht* Folgendes schreiben, um ein neues Transportgut vom Typ `T` zu erzeugen:

Gedacht: mit Generics (Compilerfehler!)	Konsequenz aus Typlöschung
<pre>class Rocket&lt;T&gt; {     T newRocketContent() {         return new T(); } // ☠ }</pre>	<pre>class Rocket&lt;T&gt; {     Object newRocketContent() {         return new Object(); } }</pre>

Tabelle 11.7 Warum `new T()` nicht funktionieren kann: Nur ein `new Object()` würde gebildet.

Als Aufrufer von `newRocketContent()` 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( Rocket<?> p ) {
    if ( p instanceof Rocket<Number> )      // ☠ illegal generic type for instanceof
        System.out.println( "Rocket mit Number" );
    else if ( p instanceof Rocket<String> ) // ☠ illegal generic type for instanceof
        System.out.println( "Rocket mit String" );
}
```

<sup>8</sup> 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.

Der Compiler meldet zu Recht einen Fehler – nicht nur eine Warnung –, weil es die Typen `Rocket<String>` und `Rocket<Number>` zur Laufzeit gar nicht gibt: Es sind nur typgelöschte `Rocket`-Objekte. Nach der Typlösung würde unsinniger Code entstehen:

```
void printType( Rocket r ) {
    if ( r instanceof Rocket )
        ...
    else if ( r instanceof Rocket )
        ...
}
```

### Keine Typumwandlungen in parametrisierten Typ

Typumwandlungen wie

```
Rocket<String> r = (Rocket<String>) new Rocket<Integer>(); // ☠ Compilerfehler
```

sind illegal. Wir haben ja extra Generics, damit der Compiler die Typen testet. Und durch die Typlösung verschwindet das Typargument, sodass der Compiler Folgendes erzeugen würde:

```
Rocket r = (Rocket) new Rocket();
```

### Kein .class für generische Typen und keine Class-Objekte mit Typargument 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<Rocket> rocketClass = Rocket.class;
```

aber dies nicht mehr:

```
Class<Rocket<String>> rocketClass = Rocket<String>.class; // ☠ Compilerfehler
```

Der Grund ist die Typlösung: Alle `Class`-Objekte für einen Typ sind gleich und haben zur Laufzeit keine Information über das Typargument:

```
Rocket<String> r1 = new Rocket<String>();
Rocket<Integer> r2 = new Rocket<Integer>();
System.out.println( r1.getClass() == r2.getClass() ); // true
```

Alle Exemplare von generischen Typen werden zur Laufzeit vom gleichen Class-Objekt repräsentiert. Hinter `Rocket<String>` und `Rocket<Integer>` steckt also immer nur `Rocket`. Kurz gesagt: Alles in spitzen 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 Typlöschung 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. `Rocket` kann zum Beispiel einmal als parametrisierter Typ `Rocket<String>` und einmal als `Rocket<Integer>` auftauchen, also als zwei Instanzen. Aber kann `Rocket` auch eine statische Methode deklarieren, die auf den Typparameter der Klasse zurückgreift? Nein, das geht nicht. Würden wir in `Rocket` 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 `Rocket<String>` oder `Rocket<Integer>` mit einem Exemplar verbinden. Bei `Rocket.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 wurde. Das Nutzen von Code wie `Rocket<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 Typlösung einfach nur `Object` heraus, kann natürlich keine Methode einmal mit einer Typvariablen und einmal mit `Object` parametrisiert sein. Folgendes ist nicht erlaubt:

```
public class Rocket<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 `Rocket<T>`«.

Ist der Typ spezieller, also etwa `String`, sieht das wieder anders aus. Dann taucht die Frage auf, welche Methode bei `Rocket<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

```
Rocket[] rockets = new Rocket[1];
```

gültig ist und mit einer Warnung versehen wird, führt bei

```
Rocket<String>[] rockets; // ❶
rockets = new Rocket<String>[1]; // ❷ ☠ Compilerfehler
```

nicht die erste, aber die zweite Zeile zum Compilerfehler »Cannot create a generic array of `Rocket<String>`«.

Typsicher kann das nicht genutzt werden, aber drei schnelle Lösungen sind denkbar:

- ▶ auf Generics ganz zu verzichten und ein `@SuppressWarnings("unchecked")` an die Array-Variable zu setzen
- ▶ Den Typ durch eine Wildcard zu ersetzen, sodass es etwa zu einem `Rocket<?>[] rockets = new Rocket<?>[1];` kommt. Wildcards sind Platzhalter, die in [Abschnitt 11.5.3](#), »Wildcards mit ?«, detaillierter vorgestellt werden.
- ▶ gleich auf Datenstrukturen der Collection-API umzusteigen, bei denen ein `Collection<String> rockets = new ArrayList<>();` keine Probleme bereitet

Als Zusammenfassung lässt sich festhalten, dass Array-Variablen von generischen Typen zwar deklariert ❶, dass aber keine Array-Objekte gebaut werden können ❷. Mit einem Trick funktioniert es:

```
class RocketFullOfMoney extends Rocket<BigInteger> {}  
Rocket<BigInteger>[] rockets = new RocketFullOfMoney[1];
```

Hübsch ist das nicht, denn es muss extra eine temporäre Klasse angelegt werden.

#### 11.2.4 Raw-Type

Generisch deklarierte Typen müssen nicht unbedingt parametrisiert werden, doch es ist einleuchtend, dass wir dem Compiler so viel Typinformation wie möglich geben sollten. Auf das Typargument 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 Rocket 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:

```
Rocket r = new Rocket();      // Gefährlich, wie wir gleich sehen werden  
r.set( "Was ist die Telefonnummer der NASA? 10 9 8 7 6 5 4 3 2 1." );  
String content = (String) r.get();
```

Ein generischer Typ, der nicht als parametrisierter Typ, also ohne Typargument, genutzt wird, heißt *Raw-Type*. In unserem Beispiel ist Rocket der Raw-Type von Rocket<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 aktueller 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.

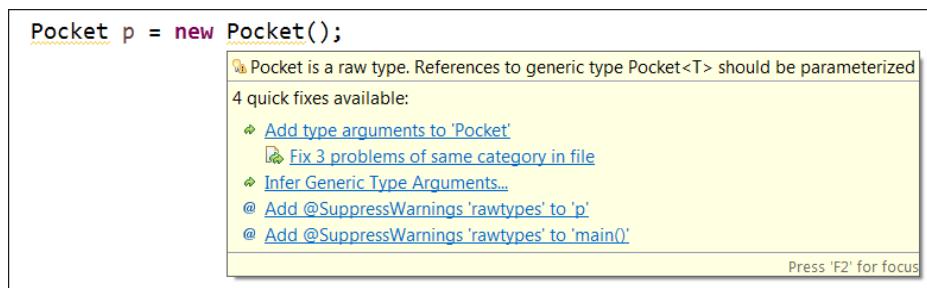


Abbildung 11.1 Eclipse warnt standardmäßig vor Raw-Types.

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 Rocket. References to generic type " +
       "Rocket<T> should be parameterized" );
```

Der Hinweis besagt, dass die Rakete hätte typisiert werden müssen. Wenn wir nicht darauf achten, kann das schnell zu Problemen führen:

```
Rocket<String> r1 = new Rocket<>();
Rocket r2 = r1;                      // Compilerwarnung
r2.set( new java.util.Date() );        // Compilerwarnung
String string = r1.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 Rakete können wir beliebige Objekte eintüten. Da aber das Objekt hinter r2 und dem typgelöschten r1 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.

## Typumwandlungen

Ein Raw-Type lässt sich automatisch in eine speziellere Form bringen, wobei es natürlich Warnungen vom Compiler gibt.

```
Rocket r = new Rocket();           // ❶ Warnung
r.set( "Roh macht nicht froh" ); // ❷ Warnung
Rocket<Point> stringRocket = r; // ❸ Warnung
Point result = stringRocket.get(); // ❹ ClassCastException zur Laufzeit
```

Bei der Variablen p, die wir über den Raw-Type nutzen ❷, prüft der Compiler gar keine Typen in `set(...)`, denn er hat sie ja nie kennengelernt. Zeile ❸ verkauft dem Compiler den Raw-Type als parametrisierten Typ. Eine explizite Typumwandlung ist nicht nötig, denn Casts sind nur zwischen »echten« Typen gültig, wie `Object` auf `Rocket`, nicht aber von `Rocket` auf `Rocket<String>`, da `Rocket<String>` ja der gleiche `Class-Typ` ist (siehe [Abschnitt 11.2.3, »Probleme der Typlösung«](#)). Eine Anweisung wie ❹, die keinen `String-Typ` aus der Rakete holt, bringt keinen Fehler zur Übersetzungszeit, sondern es knallt wegen einer internen Typumwandlung zur Laufzeit. So kann über diese Raw-nicht-raw-Hintertür ein falscher Typ in die Rakete kommen.



**Warnings (4 items)**

- ⚠ Rocket is a raw type. References to generic type Rocket<T> should be parameterized
- ⚠ Rocket is a raw type. References to generic type Rocket<T> should be parameterized
- ⚠ Type safety: The expression of type Rocket needs unchecked conversion to conform to Rocket<Point>
- ⚠ Type safety: The method set(Object) belongs to the raw type Rocket. References to generic type Rocket<T> should be parameterized

Abbildung 11.2 Warnung von Eclipse bei Raw-Types

### Annotation SuppressWarnings

In seltenen Fällen muss in den Typ konvertiert werden. Als Beispiel soll `cast(Object)` 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 Raw-Types. Dass er Warnungen ausgibt, ist voreingestellt, und diese Vorgabe ist ganz gut.

## 11.3 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 Folgendes möglich:

```
Object o1 = new Object();
Object o2 = new Point();
System.out.println( random( o1, o2 ) );
```

Da der Typ beliebig ist, können Objekte übergeben werden, die vielleicht wenig Sinn ergeben, insbesondere in ihrer Kombination.

### 11.3.1 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 Schnittstelle 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 11.10** src/main/java/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 Rückgabetypr 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.

## Typeinschränkung für gemeinsame Methoden

Eine Typeinschränkung wie `<T extends CharSequence>` ist interessant, da wir so wissen, dass ein konkretes Typargument (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, dass 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 11.11** src/main/java/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 Type-Bound können nur die Methoden von `Object` verwendet werden, somit Methoden wie `equals(...)`, `hashCode()` und `toString()`.

Betrachten wir einen Fehlerfall. Intuitiv ist anzunehmen, dass alles, was vom Typ `Comparable` ist, ein gültiger Argumenttyp 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 "incompatible 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:

```
error: method max in class ... cannot be applied to given types;
System.out.println( max( 12L, 100F ) );
^
required: V,V
```

```
found: long, float
reason: inference variable V has incompatible bounds
    equality constraints: Long,Float
    lower bounds: Float,Long
where V is a type-variable:
  V extends Comparable<V> declared in method <V>max(V,V)
```

### Hinweis

Bei `<T extends Comparable<T>>` handelt es sich um einen so genannten *rekursiven Type-Bound*. Er kommt selten vor und soll hier 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«.



### 11.3.2 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 In) ist: `T extends C & I1 & I2 & ... & In`.

Nehmen wir eine fiktive Oberklasse `Endeavour` und die Schnittstellen `Serializable` und `Comparable` an.<sup>9</sup> Dann sind die folgenden Deklarationen prinzipiell erlaubt:

- ▶ `<T extends Endeavour>`
- ▶ `<T extends Serializable & Comparable>`
- ▶ `<T extends Endeavour & Serializable>`
- ▶ `<T extends Endeavour & Comparable & Serializable>`

Syntaktisch falsch wäre etwa `<T extends Endeavour & T extends Comparable>`, da das Schlüsselwort `extends` nur einmal vorkommen darf.

---

<sup>9</sup> Comparable bekommt selbst einen Typparameter, was das Beispiel aus Gründen der Übersichtlichkeit auslässt.

## 11.4 Typparameter in der throws-Klausel \*

Wir haben in [Abschnitt 11.2.3](#), »Probleme der Typlösung«, 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.

### 11.4.1 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 11.12** `src/main/java/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() );
```

### 11.4.2 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-/Ausgabeausnahme zu erwarten, daher ist das Typargument `RuntimeException`. Beim Lesen aus Dateien oder Internetressourcen kann es jedoch zu `IOExceptions` kommen, sodass `WebIterable` das Typargument `IOException` wählt.

### Beispielimplementierungen für den parametrisierten Typ

Implementieren wir die beiden Klassen `StringIterable` und `WebIterable`. Da `StringIterable` bei der Implementierung der Schnittstelle das Typargument `RuntimeException` wählt, führt das zu einer `throws RuntimeException`, wobei die `throws`-Klausel wiederum optional ist und weggelassen werden kann:

**Listing 11.13** src/main/java/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 aus. Hier ist das Typargument `IOException`, und somit ist ein `throws IOException` an der Methodensignatur nötig:

**Listing 11.14** src/main/java/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 11.15** src/main/java/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://tutego.de/javabuch/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 Typargument `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 ungeprüfte Ausnahmen, wie beim Ablaufen von Strings bei `StringIterable`, gut möglich sind; geprüfte Ausnahmen verlangen ja immer etwas mehr Aufwand.

### 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.

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 eine `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 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.

## 11.5 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.

### 11.5.1 Arrays sind kovariant

Nehmen wir als Beispiel die Hierarchie der bekannten Wrapper-Klassen. Natürlich steht `Object` oben. Die numerischen Wrapper-Klassen erweitern alle die abstrakte Klasse `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 und Object[] über allen nichtprimitiven Felder liegt, nennt sich *Kovarianz*. Doch lässt sich das auf Generics übertragen?

### 11.5.2 Generics sind nicht kovariant, sondern invariant

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 kovariant, sie sind *invariant*. Diese Eigenschaft ist auf den ersten Blick nicht intuitiv, doch ein Beispiel rückt diesen Eindruck schnell gerade. Bleiben wir bei unserem Beispiel Rocket und den Wrapper-Klassen. Auch wenn Number die Oberklasse von Integer ist, so gilt dennoch nicht, dass Rocket<Number> ein Obertyp von Rocket<Integer> ist. Wäre es das, wäre Folgendes möglich und zur Laufzeit ein Problem:

```
Rocket<Number> r;
r = new Rocket<Integer>(); // Ist das OK?
r.set( 2.2 );
```

Das Argument 2.2 ist über Autoboxing ein Double, und daher scheint es auf Number zu passen. Allerdings sollte Double gar nicht erlaubt sein, da wir mit Rocket<Integer> ja eine Rakete für Integer aufgebaut haben, und ein Double darf nicht in die Integer-Rakete. Daher folgt: Die Ableitungsbeziehung zwischen Typen überträgt sich nicht auf generische Klassen. Ein Rocket<Number> ist also keine Oberklasse, die alle erdenklichen numerischen Typen in der Rakete erlaubt. Der Compiler meckert bei diesem Versuch sofort:

```
Rocket<Number> r;
r = new Rocket<Integer>(); // ☠ Type mismatch: cannot convert from Rocket<Integer>
                           // to Rocket<Number>
```

Auch durch eine alternative Schreibweise lässt sich der Compiler nicht in die Irre führen:

```
Rocket<Integer> r1 = new Rocket<>();
Rocket<Number> r2 = p; // ☠ Type mismatch: cannot convert
                        // from Rocket<Integer> to Rocket<Number>
```

### Hinweis



Im Fall von immutablen Objekten mit Nur-Lese-Zugriff bestünde eigentlich kein Grund für Kovarianz. Nehmen wir an, die folgende Deklaration wäre korrekt:

```
Rocket<Number> r = new Rocket<Integer>( 1 );
Number n = r.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 Basistyp, also hier `Number`, daraus zu lesen. Jedoch kann Java nicht erkennen, ob ein Typ immutable ist, und kann daher auch bei den Generics solche Ausnahmen nicht machen. Der Compiler nimmt immer an, Zugriffe wären lesend und schreibend.

### 11.5.3 Wildcards mit ?

Wir wollen eine Methode `isOneRocketEmpty(...)` schreiben, die eine variable Anzahl von Raketen bekommt – und dabei soll es egal sein, was die Rakete transportiert – und testet, ob eine Rakete leer ist. Ein Aufruf könnte so aussehen:

```
Rocket<String> r1 = new Rocket<>( "Bad-Bank" );
Rocket<Integer> r2 = new Rocket<>( 1500000 );
System.out.println( isOneRocketEmpty( r1, r2 ) ); // false
```

Die erste Idee für den Methodenkopf sieht so aus:

```
public static boolean isOneRocketEmpty( Rocket<Object>... rockets )
```

Doch halt! Da `Rocket<Object>` nicht Raketen mit allen Typen umfasst, sondern nur exakt eine Rakete trifft, die ein `Object`-Objekt enthält, ist das keine sinnvolle Parametrisierung für `isOneRocketEmpty(...)`. Das hatten wir im oberen Abschnitt schon festgestellt. Denn wäre das möglich, würde es die Typsicherheit gefährden. Wenn diese Methode tatsächlich Raketen mit allen Inhalten akzeptieren würde, so könnte einer Rakete leicht ein Wert mit falschem Typ untergeschoben werden. Wird `isOneRocketEmpty(...)` mit einem `Rocket<String>` aufgerufen, so würde wegen `isOneRocketEmpty(Rocket<Object>... rockets)` auch der Aufruf von `set(12)` auf der Rakete gültig sein, und dann stünde plötzlich statt des gewünschten Inhalts der Rakete `String` nun ein `Integer` in der Rakete. Das darf nicht gültig sein!

Ist der Typ egal, könnten wir an den Originaltyp (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 `isOneRocketEmpty(...)` beliebige Raketeninhalte entgegennehmen, aber gleichzeitig soll es der Methode auch verboten sein, falsche Dinge in die Rakete zu setzen. Ein `isOneRocketEmpty(Rocket... rockets)` 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 `Rocket<Object>` nicht der Basistyp aller Raketeninhalte ist, dann ist es `Rocket<?>`. Es ist wichtig zu verstehen, dass `?` nicht für `Object` steht, sondern für einen unbekannten Typ! Damit lässt sich `isOneRocketEmpty(...)` realisieren:

**Listing 11.16** src/main/java/com/tutego/insel/generic/RocketsEmpty.java

```
public static boolean isOneRocketEmpty( Rocket<?>... rockets ) {
    for ( Rocket<?> rocket : rockets )
        if ( rocket.isEmpty() )
            return true;

    return false;
}

public static void main( String[] args ) {
    Rocket<String> r1 = new Rocket<>( "Bad-Bank" );
    Rocket<Integer> r2 = new Rocket<>( 1500000 );
    System.out.println( isOneRocketEmpty( r1, r2 ) ); // false
    System.out.println( isOneRocketEmpty( r1, r2, new Rocket<Byte>() ) ); // true
}
```

Dass der Aufruf von `isOneRocketEmpty()` bei keiner übergebenen Rakete 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 eine 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!)
<code>Rocket&lt;?&gt; r = new Rocket&lt;Byte&gt;();</code>	<code>Rocket&lt;?&gt; r = new Rocket&lt;?&gt;();</code>
<code>static &lt;T&gt; T random( T m, T n ) { ... }</code>	<code>static &lt;?&gt; ? random( ? m, ? n ) { ... }</code>

**Tabelle 11.8** Möglicher und unmögliches Einsatz von Wildcards

## Auswirkungen auf Lese-/Schreiboperationen

Ist der Wildcard-Typ bei `Rocket<?>` im Einsatz, wissen wir nichts über den Typ, und dem Compiler gehen alle Informationen verloren. Deklarieren wir etwa

```
Rocket<?> r1 = new Rocket<Integer>();
```

oder

```
Rocket<Integer> r2 = new Rocket<Integer>();
Rocket<?> r3 = r2;
```

dann ist über die wirklichen Typargumente bei `r1` und `r3` nichts bekannt. Das hat wichtige Auswirkungen auf die Methoden, die wir auf `Rocket` aufrufen können:

- ▶ Ein Aufruf von `r1.get()` ist legal, denn alles, was die Methode liefern wird, ist immer ein Object, auch wenn es `null` ist. Die Anweisung `Object v = r1.get();` ist dementsprechend korrekt.
- ▶ Ein `r1.set(value)` ist nicht erlaubt, da dem Compiler die Typargumente von `r1` fehlen und er keine Typen prüfen kann. In `r1` dürfen wir kein Double einsetzen, da `Rocket` nur Integer speichern soll. Die einzige Ausnahme ist `null`, da `null` jeden Typ hat. `r1.set(null)` ist also eine zulässige Anweisung. Das heißt ebenso, dass mit `<?>` aufgebaute Objekte nicht automatisch immutable sind.

### 11.5.4 Bounded Wildcards

Die Angabe des Typarguments wie bei `Rocket<Integer>` und die Wildcard-Form `Rocket<?>` bilden Extreme. Die Rakete `Rocket<Integer>` nimmt nur Ganzzahlen auf, `Rocket <?>` auf der anderen Seite alles. Es muss aber auch etwas dazwischen geben, um zum Beispiel auszudrücken, dass die Rakete nur eine Zahl oder eine Zeichenkette enthalten soll.

Daher sind Typeinschränkungen mit `extends` und `super` möglich. Damit ergeben sich drei Arten von Wildcards:

Wildcard	Bezeichnung	Typargument
<code>?</code>	<i>Wildcard-Typ</i>	Ist beliebig.
<code>? extends Typ</code>	<i>Upper-bounded Wildcard-Typ</i>	alle Basistypen von Typ und Typ selbst
<code>? super Typ</code>	<i>Lower-bounded Wildcard-Typ</i>	alle Untertypen von Typ und Typ selbst

Tabelle 11.9 Die drei Wildcard-Typen

Eine Wildcard beschreibt also die Eigenschaft eines Typarguments. Wenn es

```
Rocket<? extends Number> r;
```

heißt, dann können in der Rakete p alle möglichen Number-Objekte sein. Machen wir extends und super noch an einem anderen Beispiel deutlich, das zeigt, welche Familie von Typen die Syntax beschreibt:

? extends CharSequence	? super String
CharSequence	String
String	CharSequence
StringBuffer	Object
StringBuilder	
...	

Tabelle 11.10 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-bounded Wildcard* und der *Lower-bounded Wildcard* zeigen die Sortiermethoden der Datenstrukturen und Algorithmen:

Beispiel	Bedeutung
Rocket<?> p;	Raketen mit beliebigem Inhalt
Rocket<? extends Number> p;	Raketen nur mit Zahlen, also Unterklassen von Number, wie Integer, Double, BigDecimal ...
Comparator<? super String> comp;	Comparator, der Objekte vom Typ String, Object oder CharSequence vergleicht, also Obertypen von String. Idee: Ein Comparator, der zum Beispiel CharSequence-Objekte vergleichen kann, kann auch Strings vergleichen, denn durch die Vererbung ist ein String eine Art von CharSequence. Alle Comparator-Typen <? super String> können (irgendwie) Strings vergleichen.

Tabelle 11.11 Beispiel für alle drei Wildcard-Typen

### Beispiel mit Upper-bounded-Wildcard-Typ

Die Upper-bounded Wildcard ist häufiger zu finden als die Lower-bounded-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 Rakete. Die Raketen 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 11.17** src/main/java/com/tutego/insel/generic/PortableDemo.java, Ausschnitt

```
interface Portable {
    double getWeight();
    void setWeight( double weight );
}

abstract class AbstractPortable implements Portable {
    private 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(...)` prüfen, 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. Außer-

dem ist Portable eine Schnittstelle, sodass die `areLighterThan(...)`-Methode mit einem Parametertyp `List<Portable>` überhaupt keinen Sinn ergibt. Eine vernünftige Schreibweise ist nur mit einem Upper-bounded-Wildcard-Typ möglich:

```
boolean areLighterThan( List<? extends Portable> list, double maxWeight )
```

Somit nimmt die Methode nur Listen mit Portable-Objekten an, und das ist auch nötig, denn Portable-Objekte haben ein Gewicht, und diese Eigenschaft brauchen wir.

**Listing 11.18** src/main/java/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-bounded Wildcards deklarierten Datenstruktur `List<? extends Portable>` nur gelesen, aber nicht verändert werden.

### 11.5.5 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 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 Typparameter (etwa T) nur in der Signatur auftaucht – die Signatur ergibt sich aus dem Methodenamen, Parameterliste, Ausnahmen – 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 Raketen zurückgeben soll:

**Listing 11.19** src/main/java/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 Compilerachtet 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 – es sei denn, es greift die Best Practice. 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 Aufrufer 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 Aufrufer 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 11.20** src/main/java/com/tutego/insel/generic/PortableDemo.java, PortableUtils

```
public static <T extends Portable> List<T> leftSublist( 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)`.

### 11.5.6 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-Prinzip<sup>10</sup> festhalten:

Lesen = Extends, Schreiben = Super (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 Anweisung `dest.add(portable)`. Wir erinnern uns: Mit einer Upper-bounded 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-bounded-Wildcard-Typ, den wir mit `super` schreiben. Damit folgt:

**Listing 11.21** src/main/java/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 )
        if ( portable.getWeight() < maxWeight )
            dest.add( portable );
}
```

---

<sup>10</sup> Im Englischen ist auch der Ausdruck PECS (*producer – extends, consumer – super*) in Umlauf.

Ein Beispiel für den Aufruf:

```
Listing 11.22 src/main/java/com/tutego/insel/generic/PortableDemo.java, Ausschnitt main()

List<? extends Portable> src = Arrays.asList( pen, cup );
List<? super Portable> dest = new ArrayList<>();
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 Obertypen 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 Listenoperationen 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 <T> void 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, sodass 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, mit einem Trick, denn reverse(List<?> list) kann auf eine interne

Umdrehmethode `reverse_(List<T>)` weiterleiten. Zwar müssen die Methoden anders benannt werden, aber wegen des so genannten *Wildcard-Captures* funktioniert die Abbildung von einer Wildcard auf eine Typvariable.

**Listing 11.23** src/main/java/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)`.

### 11.5.7 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 dass wir genau auf die Methodenrumpfe 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 einem Aufzählungstyp entsteht also eine Klasse, die `Enum` erweitert und als parametrierter Typ genau diese Klasse nennt: `Page extends Enum<Page>`. Vergleichen wir das mit der generischen Typdeklaration `Enum<E extends Enum<E>>`, so ist das Typargument `Page` eine Instanziierung der Typvariablen `E`. Und `Page` ist eine Unterklasse von `Enum` (`Page extends Enum`), genauso wie die Typvariable `E` das mit dem Typparameter-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 11.24** `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, zwei Aufzählungen zu vergleichen und zum Beispiel `A4.compareTo(A3)` zu schreiben. Java erlaubt dabei nur, zwei Aufzählungen vom gleichen Typ zu vergleichen: 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(String)`. 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 unfähige 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 einen Unterschied im Bytecode, der sich durch die Typlösung 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.

## 11.6 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.

### 11.6.1 Typ-Token

Wir haben zum Beispiel gesehen, dass, wenn eine Rakete mit der Typvariablen `T` deklariert wurde, dieses `T` nicht wirklich wie in einem Makro durch das Typargument ersetzt wird, sondern dass in der Regel nur einfach `Object` eingesetzt wird:

```
class Rocket<T> {
    T newRocketContent() { 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 sich ein Trick nutzen, nämlich ein `Class`-Objekt für den Typ einzusetzen.

Typargument	Class-Objekt repräsentiert Typargument
<code>String</code>	<code>String.class</code>
<code>Integer</code>	<code>Integer.class</code>

Tabelle 11.12 Transfer der Typargumente durch Class-Objekte

Dieses Class-Objekt, das nun den Typ repräsentiert, heißt *Typ-Token* (engl. *type token*). Es kommt uns natürlich entgegen, dass Class selbst als generischer Typ deklariert ist und zwei interessante Methoden ebenfalls »generifiziert« wurden.



### Beispiel

Erfrage die Class-Objekte:

```
Class<String> clazz1 = String.class;
String newInstance = clazz1.getConstructor().newInstance();
Class<? extends String> clazz2 = newInstance.getClass();
System.out.println( clazz1.equals( clazz2 ) ); // true
```

Zunächst ist da die Methode getConstructor(), die über das Class-Objekt den parameterlosen Konstruktor raus sucht. Über das Constructor-Objekt erzeugt dann newInstance() ein neues Exemplar mit dem Typ, den das Class-Objekt repräsentiert:

Mit einem gegebenen Objekt lässt sich mit getClass() das zugehörige Class-Objekt zur Klasse erfragen:

```
class java.lang.Object
```

- 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(); // 💀 Compilerfehler "Type mismatch"
```

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-Objekt 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 RuntimeException 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.getConstructor().newInstance();
    }
    catch ( ReflectiveOperationException e ) {
        throw new RuntimeException( e );
    }
}

```

Die `ReflectiveOperationException` ist die Oberklasse von `ClassNotFoundException`, `IllegalAccessException`, `InstantiationException`, `InvocationTargetException`, `NoSuchFieldException`, `NoSuchMethodException`. Es ist praktisch, diesen Basistyp anzugeben, weil das Schreibarbeit spart – bei Reflection kann immer eine Menge schiefgehen, und es gibt unzählige geprüfte Ausnahmen.

### 11.6.2 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:

Typargument	Class-Objekt repräsentiert Typargument
<code>String</code>	<code>String.class</code>
<code>Integer</code>	<code>Integer.class</code>
<code>Rocket&lt;String&gt;</code>	<code>Rocket&lt;String&gt;.class</code> Geht nicht! ☠

Tabelle 11.13 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 und ein Beispiel vorgestellt werden sollen. Hier die Klasse:

Listing 11.25 src/main/java/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 11.26** src/main/java/com/tutego/insel/generic/TypeRefDemo, main()

```
TypeRef<Rocket<String>> ref1 = new TypeRef<>(){};  
System.out.println( ref1.type ); // com.tutego.insel.generic.Rocket<java.lang.String>  
TypeRef<Rocket<Byte>> ref2 = new TypeRef<>(){};  
System.out.println( ref2.type ); // com.tutego.insel.generic.Rocket<java.lang.Byte>
```

Damit konnten wir das Typargument 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 `getConstructor().newInstance()` sind auf `Type` nicht möglich – die Schnittstelle deklariert überhaupt keine Methoden, sondern repräsentiert nur Typen.

### 11.6.3 Generics und Arrays

Die Typlöschung ist der Grund dafür, dass Arrays nicht so umgesetzt werden können, wie es sich der Entwickler denkt.<sup>11</sup> Folgendes ergibt einen Compilerfehler:

```
class TwoBox<T> {  
    T[] array = new T[ 2 ]; // ☠ Cannot create a generic array of T  
    T[] getArray() { return array; }  
}
```

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)
```

---

<sup>11</sup> Bei Oracle ([http://bugs.java.com/bugdatabase/view\\_bug.do?bug\\_id=4888066](http://bugs.java.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.«

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 Strings referenzieren, aber das Array 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` ergibt einen Compilerfehler, denn über die Typlösung wäre das ja sowieso immer `Object.class`, was den gleichen Fehler wie vorher zur Folge hätte und kein Fortschritt wäre.

## 11.6.4 Brückenmethoden

Aus der Tatsache, dass mit Generics übersetzte Klassen auf einer JVM lauffähig sein müssen, die kein generisches Typsystem besitzt, folgen diverse Hacks, die der Compiler zur Erhaltung der heiligen Kompatibilität vornimmt. Er fügt neue Methoden ein, so genannte *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 11.27** `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) );
    }
    ...
}

```

Die Klasse 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 kovarianter 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` überschreibt. Eine Klasse, die sich auch unter Java 1.4 übersetzen lässt, würde so aussehen:

**Listing 11.28** src/main/java/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 `javadoc`, das die Signaturen anzeigt:

```

$ javadoc 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 11.29** src/main/java/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 – in den Bytecode ein. Die Anweisung `font.clone()` sieht im Bytecode von `CloneableFontDemo.class` etwa so aus (disassembliert mit *javap*):

```
invokevirtual #23;
```

#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()` mit 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	Kovarianter Rückgabetyp
<code>com.tutego.insel.nongeneric.CloneableFont</code> <code>@Override public Object clone() {</code> <code>    return new Font( getAttributes() );</code> <code>}</code>	<code>com.tutego.insel.generic.CloneableFont</code> <code>@Override public Font clone() {</code> <code>    return new Font( getAttributes() );</code> <code>}</code>

**Tabelle 11.14** 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 kovariante Rückgabe 593 Byte groß und mit kovarianter Rückgabe 739 Byte. (Warum dieser satte Unterschied? Dazu gleich mehr im folgenden Abschnitt.)

Stellen wir uns nun Folgendes vor: Die erste Version von `CloneableFont` wurde lange vor der Existenz von Generics implementiert und konnte 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 zu der 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 kovariante Rückgabe nutzen kann. Er korrigiert die Methode und setzt statt `Object` den Typ `Font` ein. Er compiliert 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 aktuellen 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 keinen Anlass zu 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 aktueller 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 kovarianten 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 Brückmethode generiert. Das ist einfach, denn in die Brückmethode setzt der Compiler eine Weiterleitung:

```
@Override public Font clone() {
    return new Font( getAttributes() );
}
@Override public Object clone() {
    return (Font) clone();      // Vom Compiler in Bytecode generiert
}
```

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 sähen, die der Compiler eingeführt hat, weil dieser herumtricksen und Beschränkungen umschiffen musste. Die Brückmethoden 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.

### Brückmethode wegen einer Typvariablen 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 seit Java 5:

Java 1.4	Java 5
<pre><code>public interface Iterator {     boolean hasNext();     Object next();     void remove(); }</code></pre>	<pre><code>public interface Iterator&lt;E&gt; {     boolean hasNext();     E next();     void remove(); }</code></pre>

Tabelle 11.15 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 Rückgabewert 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 kompilierte 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 11.30** src/main/java/com/tutego/insel/nongeneric/EndlessRandomIterator.java,  
`EndlessRandomIterator`

```
public class EndlessRandomIterator implements Iterator {
    @Override public boolean hasNext() { return true; }
    @Override public Object next() { return Double.valueOf( Math.random() ); }
    @Override public void remove() { throw new UnsupportedOperationException(); }
}
```

Ein Programm, das den `EndlessRandomIterator` nutzt, empfängt bei `next()` nun ein `Double`. Aber durch den Ausschluss der Kovarianz 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 Rückgabewert `Object` erwarten und so im Bytecode vermerken – die Begründung hatten wir schon bei der Kovarianz im vorangehenden Abschnitt aufgeführt.

Passen wir den `EndlessRandomIterator` mit Generics an, ändern sich drei Dinge:

1. Erstens wird `implements Iterator` zu `implements Iterator<Double>`.
2. Dann wird `Object` `next()` zu `Double` `next()`.
3. Drittens kann es im Rumpf von `next()` durch das Autoboxing etwas kürzer werden, nämlich `return Math.random()`.

Der wichtige Punkt ist, dass sich der Rückgabewert bei `next()` von `Object` in `Double` ändert. An der Stelle muss der Compiler mit einer Brückenmethode eingreifen, sodass im Bytecode wieder zwei `next()`-Methoden stehen: einmal `Object` `next()` und dann `Double` `next()`. Denn ohne `Object` `next()` würde der alte Programmcode, der `Object` `next()` erwartet, plötzlich nicht mehr laufen, und das wäre ein Bruch der Abwärtskompatibilität.

## 11.7 Zum Weiterlesen

Generics sind für den Nutzer zum Glück relativ einfach zu verstehen, weil Anwender nur in die spitzen Klammern einen Typ setzen müssen und dann fertig sind. Bibliothekdesigner haben es da deutlich schwerer, weil sie überlegen müssen, welche Typen tatsächlich für eine Operation nötig sind, und entsprechend Wildcards anbieten müssen. Zum Training ist es sinnvoll, sich die Anwendung der Generics aller Methoden von `java.util.Collections` einmal vorzunehmen.

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-Type-Tokens kommen in der Java-Welt nicht besonders oft vor.



# Kapitel 12

## Lambda-Ausdrücke und funktionale Programmierung

»EDV-Systeme verarbeiten, womit sie gefüttert werden.

*Kommt Mist rein, kommt Mist raus.«*

– André Kostolany (1906–1999)

Bei der Entwicklung von Maschinensprache (bzw. Assembler) hin zur Hochsprache ist eine interessante Geschichte der Parametrisierung abzulesen. Schon die ersten Hochsprachen erlaubten eine Parametrisierung von Funktionen mit unterschiedlichen Argumenten. Die Programmiersprache Java, die im Jahr 1996 geboren wurde, bot das von Anfang an, da sie erst mehrere Jahrzehnte nach den ersten Hochsprachen entstand. Relativ spät folgten dann die Generics. Die Parametrisierung des Typs wurde erst 2004 mit der Version 5 realisiert. Bis dahin konnte eine Liste zum Beispiel Zeichenketten ebenso enthalten wie Pantoffeltierchen (als Java-Objekte). Funktionale Programmierung ermöglichte nun eine Parametrisierung des Verhaltens; eine Sortiermethode arbeitet immer gleich, aber ihr Verhalten bei den Vergleichen wird angepasst. Das ist eine ganz andere Qualität, als unterschiedliche Werte zu übergeben. Mit Lambda-Ausdrücken ist das ganz einfach.

### 12.1 Code = Daten

Wer den Begriff »Daten« hört, denkt zunächst einmal an Zahlen, Bytes, Zeichenketten oder auch komplexe Objekte mit ihrem Zustand. Wir wollen in diesem Kapitel diese Sicht ein wenig erweitern und auf Programmcode lenken. Java-Code, versinnbildlicht als Serie von Bytecodes, besteht auch aus Daten. Und wenn wir uns einmal auf diese Sichtweise einlassen, dass Code gleich Daten ist, dann lässt sich Code auch wie Daten übergeben und so von einem Punkt zum anderen übertragen, speichern und später referenzieren. Mit dieser Möglichkeit, Code zu übertragen, lässt sich das Verhalten von Algorithmen leicht anpassen. Beginnen wir mit ein paar Beispielen, bei denen Programmcode übergeben wird, auf den dann später zugriffen wird:

- ▶ Ein Thread führt Programmcode im Hintergrund aus. Der Programmcode, den der Java-Thread ausführen soll, wird in ein Objekt vom Typ Runnable verpackt, genau genommen in eine `run()`-Methode gesetzt. Kommt der Thread zum Zuge, ruft er die `run()`-Methode auf.

- ▶ Ein Timer ist eine `java.util`-Klasse, die zu bestimmten Zeitpunkten Programmcode ausführen kann. Der Objektmethode `scheduleAtFixedRate(...)` wird dabei ein Objekt vom Typ `TimerTask` übergeben, das den Programmcode enthält.
- ▶ Zum Sortieren von Daten kann eine eigene Ordnung definiert werden, die dem Sortierer als `Comparator` übergeben werden kann. Der `Comparator` deklariert eine Vergleichsmethode, an die sich der Sortierer wendet, um zwei Objekte in die gewünschte Reihenfolge zu bringen.
- ▶ Aktiviert der Benutzer auf der Oberfläche eine Schaltfläche, so führt das zu einer Aktion. Der Programmcode steckt – beim UI-Framework Swing – in einem Objekt vom Typ `ActionListener` und wird an der Schaltfläche `JButton` mit `addActionListener(...)` festgemacht. Kommt es zu einer Schaltflächenaktivierung, arbeitet das UI-System den Programmcode in der Methode `actionPerformed(...)` des gespeicherten `ActionListener` ab.

Um Programmcode von einer Stelle zur anderen zu bringen, wird in Java immer der gleiche Mechanismus eingesetzt: Eine Klasse implementiert eine (in der Regel nichtstatische) Methode, in der der auszuführende Programmcode steht. Ein Objekt dieser Klasse wird an eine andere Stelle übergeben, und der Interessent greift dann über die Methode auf den Programmcode zu. Dass ein Objekt noch mehr als diese eine Implementierung enthalten kann, etwa Variablen, Konstanten, Konstruktoren, ist dafür nicht relevant. Diesen Mechanismus schauen wir uns jetzt in verschiedenen Varianten genauer an.

### Geschachtelte Klassen als Code-Transporter

Bleiben wir bei dem Beispiel mit den Vergleichen. Angenommen, wir sollen `Strings` so sortieren, dass Weißraum vorne und hinten bei den Vergleichen ignoriert wird, also "Newton" gleich "Newton" ist. Bei Vorgaben dieser Art muss einem Sortieralgorithmus ein Stückchen Code übergeben werden, damit er die korrekte Reihenfolge herstellen kann. Praktisch sieht das so aus:

**Listing 12.1** `src/main/java/com/tutego/insel/lambda/CompareTrimmedStrings.java`, `main()`

```
public static void main( String[] args ) {
    class TrimmingComparator implements Comparator<String> {
        @Override public int compare( String s1, String s2 ) {
            return s1.trim().compareTo( s2.trim() );
        }
    }
    String[] words = { "M", "\nSkyfall", " 0", "\t\tAdele\t" };
    Arrays.sort( words, new TrimmingComparator() );
    System.out.println( Arrays.toString( words ) );
}
```

Die Ausgabe ist:

```
[      Adele      , M,  Q,
Skyfall]
```

Der `TrimmingComparator` enthält in der `compare(...)`-Methode den Programmcode für die Ver-  
gleichslogik. Ein Exemplar vom `TrimmingComparator` wird aufgebaut und `Arrays.sort(...)`  
übergeben. Das geht mit weniger Code!

### Innere anonyme Klassen als Code-Transporter

Klassen enthalten Programmcode, und Exemplare der Klassen werden an Methoden wie `sort(...)` übergeben, damit der Programmcode dort hinkommt, wo er gebraucht wird. Doch elegant ist das nicht. Für die Beschreibung des Programmcodes ist extra eine eigene Klasse erforderlich. Das ist viel Schreibarbeit, und über eine innere anonyme Klasse lässt sich der Programmcode schon ein wenig verkürzen:

```
String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
Arrays.sort( words, new Comparator<String>() {
    @Override public int compare( String s1, String s2 ) {
        return s1.trim().compareTo( s2.trim() );
    }
});
System.out.println( Arrays.toString( words ) );
```

Allerdings ist das immer noch aufwändig: Wir müssen eine Methode überschreiben und dann ein Objekt aufbauen. Für Programmautoren ist das lästig, und die JVM hat es mit vielen überflüssigen Klassendeklarationen zu tun. Die Frage ist: Wenn der Compiler weiß, dass bei `sort(...)` ein `Comparator` nötig ist, und wenn ein `Comparator` sowieso nur eine Methode hat, müssen dann `Comparator` und `compare(...)` überhaupt genannt werden?

### Abkürzende Schreibweise durch Lambda-Ausdrücke

Mit Lambda-Ausdrücken lässt sich Programmcode leichter an eine Methode übergeben, denn es gibt eine kompakte Syntax für die Implementierung von Schnittstellen mit einer Operation. Für unser Beispiel sieht das so aus:

```
String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
Arrays.sort( words,
    (String s1, String s2) -> { return s1.trim().compareTo(s2.trim()); } );
System.out.println( Arrays.toString( words ) );
```

Der in fett gesetzte Ausdruck nennt sich *Lambda-Ausdruck*. Er ist eine kompakte Art und Weise, Schnittstellen mit genau einer Methode zu implementieren; die Schnittstelle `Comparator` hat genau eine Operation `compare(...)`.

Optisch sind sich ein Lambda-Ausdruck und eine Methodendeklaration ähnlich; was wegfällt sind Modifizierer, der Rückgabetyp, der Methodenname und (mögliche) throws-Klauseln.

Methodendeklaration	Lambda-Ausdruck
<pre>public int compare ( String s1, String s2 ) { return s1.trim().compareTo( s2.trim() ); }</pre>	<pre>( String s1, String s2 ) -&gt; { return s1.trim().compareTo( s2.trim() ); }</pre>

Tabelle 12.1 Vergleich der Methodendeklaration einer Schnittstelle mit dem Lambda-Ausdruck

Wenn wir uns den Lambda-Ausdruck als Implementierung dieser Schnittstelle anschauen, dann lässt sich dort nichts von Comparator oder compare(...) ablesen – ein Lambda-Ausdruck repräsentiert mehr oder weniger nur den Java-Code und lässt das, was der Compiler aus dem Kontext herleiten kann, weg.

Alle Lambda-Ausdrücke lassen sich in einer Syntax formulieren, die die folgende allgemeine Form hat:

```
( LambdaParameter ) -> { Anweisungen }
```

Lambda-Parameter sind sozusagen die Eingabewerte für die Anweisungen. Die Parameterliste wird so deklariert, wie von Methoden oder Konstruktoren bekannt, allerdings gibt es keine Varargs. Es gibt syntaktische Abkürzungen, wie wir später sehen werden, doch vorerst bleiben wir bei dieser Schreibweise.

### Geschichte

Der Java-Begriff »Lambda-Ausdruck« geht auf das Lambda-Kalkül (in der englischen Literatur *Lambda calculus* genannt, auch geschrieben als  $\lambda$ -calculus) aus den 1930er Jahren zurück und ist eine formale Sprache zur Untersuchung von Funktionen.

## 12.2 Funktionale Schnittstellen und Lambda-Ausdrücke im Detail

In unserem Beispiel haben wir den Lambda-Ausdruck als Argument von `Array.sort(...)` eingesetzt:

```
Arrays.sort( words,
( String s1, String s2 ) -> { return s1.trim().compareTo(s2.trim()); } );
```

Wir hätten aber auch den Lambda-Ausdruck explizit einer lokalen Variablen zuweisen können, was deutlich macht, dass der hier eingesetzte Lambda-Ausdruck vom Typ `Comparator` ist:

```
Comparator<String> c = (String s1, String s2) -> {
    return s1.trim().compareTo( s2.trim() );
}
Arrays.sort( words, c );
```

### 12.2.1 Funktionale Schnittstellen

Nicht zu jeder Schnittstelle gibt es eine Abkürzung über einen Lambda-Ausdruck, und es gibt eine zentrale Bedingung, wann ein Lambda-Ausdruck verwendet werden kann.

#### Definition

Schnittstellen, die nur eine Operation (abstrakte Methode) besitzen, heißen *funktionale Schnittstellen*. Ein *Funktionsdeskriptor* beschreibt diese Methode. Eine abstrakte Klasse mit genau einer abstrakten Methode zählt *nicht* als funktionale Schnittstelle.

Lambda-Ausdrücke und funktionale Schnittstellen haben eine ganz besondere Beziehung, denn ein Lambda-Ausdruck ist ein Exemplar einer solchen funktionalen Schnittstelle. Natürlich müssen Typen und Ausnahmen passen. Dass funktionale Schnittstellen genau eine abstrakte Methode vorschreiben, ist eine naheliegende Einschränkung, denn gäbe es mehrere, müsste ein Lambda-Ausdruck ja auch mehrere Implementierungen anbieten oder irgendwie eine Methode bevorzugen und andere ausblenden.

Wenn wir ein Objekt vom Typ einer funktionalen Schnittstelle aufbauen möchten, können wir folglich zwei Wege einschlagen: Es lässt sich die traditionelle Konstruktion über die Bildung von Klassen wählen, die funktionale Schnittstellen implementieren, und dann mit `new` ein Exemplar bilden, oder es lässt sich mit kompakten Lambda-Ausdrücken arbeiten. Moderne IDEs zeigen uns an, wenn kompakte Lambda-Ausdrücke zum Beispiel statt innerer anonymer Klassen genutzt werden können, und bieten uns mögliche Refactorings an. Lambda-Ausdrücke machen den Code kompakter und nach kurzer Eingewöhnung auch lesbarer.

#### Hinweis

Funktionale Schnittstellen müssen auf genau eine zu implementierende Methode hinauslaufen, auch wenn aus Oberschnittstellen mehrere Operationen vorgeschrieben werden, die sich aber durch den Einsatz von Generics auf eine Operation verdichten:

```
interface I<S,T extends CharSequence> {
    void len( S text );
    void len( T text );
}
```



```

    }
interface FI extends I<String, String> { }
```

FI ist unsere funktionale Schnittstelle mit einer eindeutigen Operation `len(String)`. Statische und Default-Methoden stören in funktionalen Schnittstellen nicht.

### Viele funktionale Schnittstellen in der Java-Standardbibliothek

Java bringt schon viele Schnittstellen mit, die als funktionale Schnittstellen gekennzeichnet sind. Darüber hinaus führt das Paket `java.util.function` mehr als 40 neue funktionale Schnittstellen ein. Eine kleine Auswahl:

- ▶ `interface Runnable { void run(); }`
- ▶ `interface Supplier<T> { T get(); }`
- ▶ `interface Consumer<T> { void accept(T t); }`
- ▶ `interface Comparator<T> { int compare(T o1, T o2); }`
- ▶ `interface ActionListener { void actionPerformed(ActionEvent e); }`

Ob die Schnittstelle noch andere Default-Methoden hat – also Schnittstellenmethoden mit vorgegebener Implementierung –, ist egal, wichtig ist nur, dass sie genau eine zu implementierende Operation deklariert.

#### 12.2.2 Typ eines Lambda-Ausdrucks ergibt sich durch Zieltyp

In Java hat jeder Ausdruck einen Typ. Die Ausdrücke `1` und `1*2` haben einen Typ (nämlich `int`), genauso wie `"A" + "B"` (Typ `String`) oder `String.CASE_INSENSITIVE_ORDER` (Typ `Comparator<String>`). Lambda-Ausdrücke haben auch immer einen Typ, denn ein Lambda-Ausdruck ist immer ein Exemplar einer funktionalen Schnittstelle. Damit steht auch der Typ fest. Allerdings ist es im Vergleich zu Ausdrücken wie `1*2` bei Lambda-Ausdrücken etwas anders gelagert, denn der Typ von Lambda-Ausdrücken ergibt sich ausschließlich aus dem Kontext. Erinnern wir uns an den Aufruf von `sort(...)`:

```
Arrays.sort( words, (String s1, String s2) -> { return ... } );
```

Dort steht nichts vom Typ `Comparator`, sondern der Compiler erkennt aus dem Typ des zweiten Parameters von `sort(...)`, der ja `Comparator` ist, ob der Lambda-Ausdruck auf die Methode des Comparators passt oder nicht.

Der Typ eines Lambda-Ausdrucks ist folglich abhängig davon, welche funktionale Schnittstelle er im jeweiligen Kontext gerade realisiert. Der Compiler kann ohne Kenntnis des *Zieltyps* (engl. *target type*) keinen Lambda-Ausdruck aufbauen.

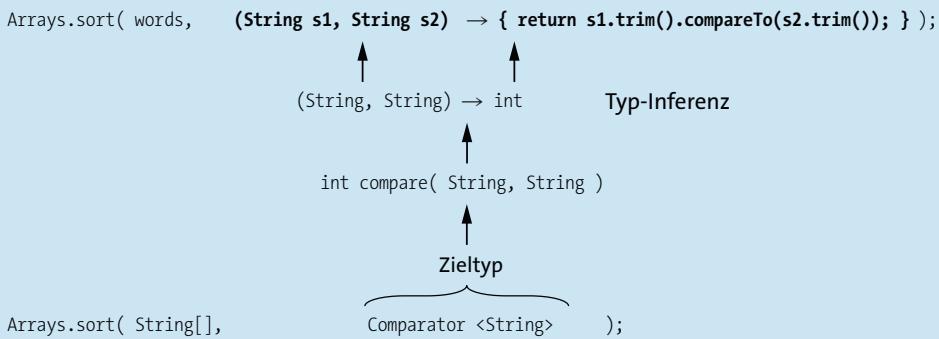


Abbildung 12.1 Typ-Inferenz vom Compiler

### Beispiel

Callable und Supplier sind funktionale Schnittstellen mit Methoden, die keine Parameterlisten deklarieren und eine Referenz zurückgeben; der Code für den Lambda-Ausdruck sieht gleich aus:

```
java.util.concurrent.Callable<String> c = () -> { return "Rückgabe"; };
java.util.function.Supplier<String> s = () -> { return "Rückgabe"; };
```



### Wer bestimmt den Zieltyp?

Gerade weil an dem Lambda-Ausdruck der Typ nicht abzulesen ist, kann er nur dort verwendet werden, wo ausreichend Typinformationen vorhanden sind. Das sind unter anderem die folgenden Stellen:

- ▶ Variablendeclarationen: etwa wie bei `Supplier<String> s = () -> { return ""; }`
- ▶ Argumente an Methoden oder Konstruktoren: Der Parameterotyp gibt alle Typinformationen. Ein Beispiel lieferte `Arrays.sort(..)`.
- ▶ Methodenrückgaben: Das könnte aussehen wie `Comparator<String> trimComparator() { return (s1, s2) -> { return ... }; }`.
- ▶ Bedingungsoperator: Der `?:-`-Operator liefert je nach Bedingung einen unterschiedlichen Lambda-Ausdruck. Beispiel: `Supplier<Double> randomNegOrPos = Math.random() > 0.5 ? () -> { return Math.random(); } : () -> { return Math.random(); };`



### Hinweis

Eine lokale Variablendeclaration mit `var` funktioniert mit Lambda-Ausdrücken nicht: Der Lambda-Ausdruck braucht die linke Seite und `var` den Typ auf der rechten.

```
var o = () -> {};
```

// ☠ Lambda expression needs an explicit target-type

## Parametertypen

In der Praxis ist der häufigste Fall, dass die Parametertypen von Methoden den Zieltyp vorgeben. Der Einsatz von Lambda-Ausdrücken ändert ein wenig die Sichtweise auf überladene Methoden. Unser Beispiel mit `() -> { return "Rückgabe"; }` macht das deutlich, denn es »passt« auf den Zieltyp `Callable<String>` genauso wie auf `Supplier<String>`. Nehmen wir zwei überladene Methoden `run(...)` an:

```
class OverloadedFunctionalInterfaceMethods {
    static <V> void run( Callable<V> callable ) { }
    static <V> void run( Supplier<V> callable ) { }
}
```

Spielen wir den Aufruf der Methoden einmal durch:

```
Callable<String> c = () -> { return "Rückgabe"; };
Supplier<String> s = () -> { return "Rückgabe"; };
run( c );
run( s );
// run( () -> { return "Rückgabe"; } ); // ☠ Compilerfehler
run( (Callable<String>) () -> { return "Rückgabe"; } );
```

Rufen wir `run(c)` bzw. `run(s)` auf, ist das kein Problem, denn `c` und `s` sind klar typisiert. Aber `run(...)` mit dem Lambda-Ausdruck aufzurufen funktioniert nicht, denn der Zieltyp (entweder `Callable` oder `Supplier`) ist mehrdeutig; der (Eclipse-)Compiler meldet: »The method `run(Callable<Object>)` is ambiguous for the type T«. Hier sorgt eine explizite Typumwandlung für Abhilfe.



### Tipp zum API-Design

Aus Sicht eines API-Designers sind überladene Methoden natürlich schön, aus Sicht des Nutzers sind Typumwandlungen aber nicht schön. Um explizite Typumwandlungen zu vermeiden, sollte auf überladene Methoden verzichtet werden, wenn diese den Parametertyp einer funktionalen Schnittstelle aufweisen. Stattdessen lassen sich die Methoden unterschiedlich benennen (was bei Konstruktoren natürlich nicht funktioniert). Wird in unserem Fall die Methode `runCallable(...)` und `runSupplier(...)` genannt, ist keine Typumwandlung mehr nötig, und der Compiler kann den Typ herleiten.

## Rückgabetypen

Typ-Inferenz spielt bei Lambda-Ausdrücken eine große Rolle – das gilt insbesondere für die Rückgabetypen, die überhaupt nicht in der Deklaration auftauchen und für die es gar keine Syntax gibt; der Compiler »inferrt« sie. In unserem Beispiel

```
Comparator<String> c =
  (String s1, String s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

ist `String` als ParameterTyp der `Comparator`-Methode ausdrücklich gegeben; der Rückgabetypr `int`, den der Ausdruck `s1.trim().compareTo( s2.trim() )` liefert, taucht dagegen nicht auf.

Mitunter muss dem Compiler etwas geholfen werden: Nehmen wir die funktionale Schnittstelle `Supplier<T>`, die eine Methode `T get()` deklariert, für ein Beispiel. Die Zuweisung

```
Supplier<Long> two = () -> { return 2; } // ☠ Compilerfehler
```

ist nicht korrekt und führt zum Compilerfehler »incompatible types: bad return type in lambda expression«. `2` ist ein Literal vom Typ `int`, und der Compiler kann es nicht an `Long` anpassen. Wir müssen schreiben

```
Supplier<Long> two = () -> { return 2L; };
```

oder

```
Supplier<Long> two = () -> { return (long) 2; };
```

Bei Lambda-Ausdrücken gelten keine wirklich neuen Regeln im Vergleich zu Methodenrückgaben, denn auch eine Methodendeklaration wie

```
Long two() { return 2; } // ☠ Compilerfehler
```

wird vom Compiler bemängelt. Doch weil Wrapper-Typen durch die Generics bei funktionalen Schnittstellen viel häufiger sind, treten diese Besonderheiten öfter auf als bei Methodendeklarationen.

### Sind Lambda-Ausdrücke Objekte?

Ein Lambda-Ausdruck ist ein Exemplar einer funktionalen Schnittstelle und tritt als Objekt auf. Bei Objekten besteht normalerweise zu `java.lang.Object` immer eine natürliche Ist-eine-Art-von-Beziehung. Fehlt aber der Kontext, ist selbst die Ist-eine-Art-von-Beziehung zu `java.lang.Object` gestört und Folgendes nicht korrekt:

```
Object o = () -> {}; // ☠ Compilerfehler
```

Der Compilerfehler ist: »incompatible types: the target type must be a functional interface«. Nur eine explizite Typumwandlung kann den Fehler korrigieren und dem Compiler den Zieltyp vorgeben:

```
Object r = (Runnable) () -> {};
```

Lambda-Ausdrücke haben keinen eigenen Typ an sich, und für das Typsystem von Java ändert sich im Prinzip nichts. Möglicherweise ändert sich das in späteren Java-Versionen.



### Hinweis

Dass Lambda-Ausdrücke Objekte sind, ist eine Eigenschaft, die nicht überstrapaziert werden sollte. So sind die üblichen Object-Methoden `equals(Object)`, `hashCode()`, `getClass()`, `toString()` und die zur Thread-Kontrolle ohne besondere Bedeutung. Es sollte auch nie ein Szenario geben, in dem Lambda-Ausdrücke mit `==` verglichen werden müssen, denn das Ergebnis ist laut Spezifikation undefined. Echte Objekte haben eine Identität, einen Identity-Hashcode, lassen sich vergleichen und mit `instanceof` testen, können mit einem synchronisierten Block abgesichert werden; all dies gilt für Lambda-Ausdrücke nicht. Im Grunde charakterisiert der Begriff »Lambda-Ausdruck« schon sehr gut, was wir nie vergessen sollten: Es handelt sich um einen Ausdruck, also etwas, was ausgewertet wird und ein Ergebnis produziert.

#### 12.2.3 Annotation @FunctionalInterface

Jede Schnittstelle mit genau einer abstrakten Methode eignet sich als funktionale Schnittstelle und damit für einen Lambda-Ausdruck. Jedoch soll nicht jede Schnittstelle in der API, die im Moment nur eine abstrakte Methode deklariert, auch für Lambda-Ausdrücke verwendet werden. So kann zum Beispiel eine Weiterentwicklung der Schnittstelle mit mehreren (abstrakten) Methoden geplant sein, aber zurzeit ist nur eine abstrakte Methode vorhanden. Der Compiler kann nicht wissen, ob sich eine Schnittstelle vielleicht weiterentwickelt.

Um kenntlich zu machen, dass ein interface als funktionale Schnittstelle gedacht ist, existiert der Annotationstyp `FunctionalInterface` im `java.lang`-Paket. Diese Annotation markiert, dass es bei genau einer abstrakten Methode und damit bei einer funktionalen Schnittstelle bleiben soll.



### Beispiel

Eine eigene funktionale Schnittstelle sollte immer als `FunctionalInterface` markiert werden:

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void foo();
}
```

Der Compiler prüft, ob die Schnittstelle mit einer solchen Annotation tatsächlich nur exakt eine abstrakte Methode enthält, und löst einen Fehler aus, wenn dem nicht so ist. Aus Kompatibilitätsgründen erzwingt der Compiler diese Annotation bei funktionalen Schnittstellen allerdings nicht; das ermöglicht es, geschachtelte Klassen, die herkömmliche Schnittstellen mit einer Methode implementieren, einfach in Lambda-Ausdrücke umzuschreiben. Die Annotation ist also keine Voraussetzung für die Nutzung der Schnittstelle in einem Lambda-Ausdruck und dient bisher nur der Dokumentation. In der Java SE sind aber alle zentralen funktionalen Schnittstellen so ausgezeichnet.



### Tipp

Was mit @FunctionalInterface ausgezeichnet ist, bekommt in der Javadoc einen Extrasatz: »Functional Interface: This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.« Das macht funktionale Schnittstellen noch besser sichtbar.



### Hinweis

Der Annotationstyp FunctionalInterface ist auch zur Laufzeit sichtbar, was heißt, dass auch Programme über Reflection testen können, ob eine Schnittstelle annotiert ist.<sup>1</sup>

## 12.2.4 Syntax für Lambda-Ausdrücke

Lambda-Ausdrücke haben wie Methoden mögliche Parameter- und Rückgabewerte. Die Java-Grammatik für die Schreibweise von Lambda-Ausdrücken sieht ein paar nützliche syntaktische Abkürzungen vor.

### Ausführliche Schreibweise

Lambda-Ausdrücke lassen sich auf unterschiedliche Art und Weise schreiben, da es für diverse Konstruktionen Abkürzungen gibt. Eine Form, die jedoch immer gilt, ist:

( *LambdaParameter* ) -> { *Anweisungen* }

Der Lambda-Parameter besteht (voll ausgeschrieben) wie ein Methodenparameter aus a) dem Typ, b) dem Namen und c) optionalen Modifizierern.

Der Parametername öffnet einen neuen Gültigkeitsbereich für eine Variable, wobei der Parametername *keine* anderen Namen von lokalen Variablen überlagern darf. Hier verhält sich die Lambda-Parametervariable wie eine neue Variable aus einem inneren Block und nicht wie eine Variable aus einer inneren Klasse, wo die Sichtbarkeit anders ist.



### Beispiel

Folgendes ergibt einen Compilerfehler im Lambda-Ausdruck, weil *s* schon deklariert ist; die Parametervariable vom Lambda-Ausdruck muss »frisch« sein:

```
String s = "Make Donald Drumpf Again";
Comparator<String> c = (String s, String t) -> { ... }; // 💀 Compilerfehler
```

<sup>1</sup> Der Annotationstyp ist selbst mit @Documented @Retention(value=RUNTIME) @Target(value=TYPE) annotiert.

### Abkürzung 1: Typ-Inferenz (impliziter Typ)

Der Java-Compiler kann viele Typen aus dem Kontext ablesen, was *Typ-Inferenz* genannt wird. Wir kennen so etwas vom Diamant-Operator, wenn wir etwa schreiben:

```
List<String> list = new ArrayList<>()
```

Sind für den Compiler genug Typinformationen verfügbar, dann erlaubt der Compiler bei Lambda-Ausdrücken eine Abkürzung. Bei der Deklaration

```
Comparator<String> c =
    (String s1, String s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

ist dem Compiler dank Typ-Inferenz klar, dass rechts vom Gleichheitszeichen ein Ausdruck vom Typ `Comparator<String>` kommen muss und die `Comparator`-Methode `compare(...)` zwei Parameter vom Typ `String` besitzt. Daher funktioniert die folgende Abkürzung:

```
Comparator<String> c = (s1, s2) -> { return s1.trim().compareTo( s2.trim() ); };
```



#### Hinweis

Die Parameterliste enthält entweder explizit deklarierte Parametertypen oder implizite Inferred-Typen. Eine Mischung ist nicht erlaubt, der Compiler blockt so etwas wie `(String s1, s2)` oder `(s1, String s2)` mit einem Fehler ab.

Wenn der Compiler die Typen ablesen kann, sind die Parametertypen optional. Aber Typ-Inferenz ist nicht immer möglich, weshalb die Abkürzung nicht immer möglich ist. Außerdem hilft die explizite Schreibweise auch der Lesbarkeit: Kurze Ausdrücke sind nicht unbedingt die verständlichsten.



#### Hinweis

Der Compiler liest aus den Typen ab, ob alle Eigenschaften vorhanden sind. Die Typen sind dabei entweder explizit oder implizit gegeben.

```
Comparator<String> sc = (a, b) -> { return Integer.compare( a.length(),
    b.length() ); };
Comparator<BitSet> bc = (a, b) -> { return Integer.compare( a.length(),
    b.length() ); };
```

Die Klassen `String` und `BitSet` besitzen beide die Methode `length()`, daher ist der Lambda-Ausdruck korrekt. Der gleiche Lambda-Code im Quellcode lässt sich für zwei völlig verschiedene Klassen einsetzen, die überhaupt keine Gemeinsamkeiten haben, nur dass sie zufällig beide eine Methode namens `length()` besitzen.

### Abkürzung 2: Typ-Inferenz durch var

In Lambda-Ausdrücken kann ebenfalls `var` eingesetzt werden.<sup>2</sup> Eine Mischung aus expliziten Typen, impliziten Typen und `var`-Typen ist nicht gestattet:

```
IntBinaryOperator f0 = (var x, var y) -> x + y;
IntBinaryOperator f1 = (x, var y) -> x + y;           // 💀 implizit und var
IntBinaryOperator f2 = (var x, y) -> x + y;           // 💀 var und implizit
IntBinaryOperator f3 = (int x, var y) -> x + y;        // 💀 var und explizit
IntBinaryOperator f4 = (int x, y) -> x + y;           // 💀 explizit und implizit
```

Eigentlich ergibt der Einsatz von `var` bei der Möglichkeit von impliziten Typen wenig Sinn. Allerdings ist er erstens konsistent und zweitens dann nützlich, wenn eine Annotation an die Variable kommt, denn hierfür ist die Typangabe notwendig.

### Abkürzung 3: Lambda-Rumpf ist entweder einzelner Ausdruck oder Block

Besteht der Rumpf eines Lambda-Ausdrucks nur aus einem einzelnen Ausdruck, kann eine verkürzte Schreibweise die Blockklammern und das Semikolon einsparen. Statt

```
( LambdaParameter ) -> { return Ausdruck; }
```

heißt es dann

```
( LambdaParameter ) -> Ausdruck
```

Lambda-Ausdrücke mit einer `return`-Anweisung im Rumpf kommen häufig vor, da dies den typischen Funktionen entspricht. Somit ist es eine willkommene Verkürzung, wenn die abgekürzte Syntax für Lambda-Ausdrücke lediglich den Ausdruck fordert, der dann die Rückgabe bildet. Tabelle 12.2 zeigt drei Beispiele.

Lange Schreibweise	Abkürzung
<code>(s1, s2) -&gt;</code> <code>{ return s1.trim().compareTo( s2.trim() ); }</code>	<code>(s1, s2) -&gt;</code> <code>s1.trim().compareTo( s2.trim() )</code>
<code>(a, b) -&gt; { return a + b; }</code>	<code>(a, b) -&gt; a + b</code>
<code>() -&gt; { System.out.println(); }</code>	<code>() -&gt; System.out.println()</code>

**Tabelle 12.2** Ausführliche und abgekürzte Schreibweise

Ausdrücke können in Java auch zu `void` ausgewertet werden, sodass ohne Probleme ein Aufruf wie `System.out.println()` in der kompakten Schreibweise ohne Block gesetzt werden

<sup>2</sup> Diese Möglichkeit ist neu in Java 11 dank <https://openjdk.java.net/jeps/323>.

kann. Das heißt, wenn Lambda-Ausdrücke mit der kurzen Ausdruckssyntax eingesetzt werden, können diese Ausdrücke etwas zurückgeben, müssen aber nicht.



### Hinweis

Die Schreibweise mit den geschweiften Klammern und den Rückgabe-Ausdrücken kann nicht gemischt werden. Entweder gibt es einen Block geschweifter Klammern und `return` oder keine Klammern und kein `return`-Schlüsselwort. Falsche Mischungen ergeben Fehler:

```
Comparator<String> c;
c = (s1, s2) -> { s1.trim().compareTo( s2.trim() ) };    // 💀 Compilerfehler ①
c = (s1, s2) -> return s1.trim().compareTo( s2.trim() ); // 💀 Compilerfehler ②
```

Würden wir in ① ein explizites `return` nutzen, wäre alles in Ordnung, würde bei ② das `return` wegfallen, wäre die Zeile auch compilierbar.

Ob Lambda-Ausdrücke eine Rückgabe haben, drücken zwei Begriffe aus:

- ▶ **void-kompatibel:** Der Lambda-Rumpf gibt kein Ergebnis zurück, entweder weil der Block kein `return` enthält oder ein `return` ohne Rückgabe oder weil ein void-Ausdruck in der verkürzten Schreibweise eingesetzt wird. Der Lambda-Ausdruck `() -> System.out.println()` ist also void-kompatibel, genauso wie `() -> {}`.
- ▶ **Wertkompatibel:** Der Rumpf beendet den Lambda-Ausdruck mit einer `return`-Anweisung, die einen Wert zurückgibt, oder besteht aus der kompakten Schreibweise mit einer Rückgabe ungleich void.

Eine Mischung aus void- und wertkompatibel ist nicht erlaubt und führt wie bei Methoden zu einem Compilerfehler.<sup>3</sup>

### Abkürzung 4: einzelner Identifizierer statt Parameterliste und Klammern

Besteht die Parameterliste

1. nur aus einem einzelnen Identifizierer
  2. und ist der Typ durch Typ-Inferenz klar,
- können die runden Klammern wegfallen.

Lange Schreibweise	Typen inferred	Vollständig abgekürzt
<code>(String s) -&gt; s.length()</code>	<code>(s) -&gt; s.length()</code>	<code>s -&gt; s.length()</code>
<code>(int i) -&gt; Math.abs(i)</code>	<code>(i) -&gt; Math.abs(i)</code>	<code>i -&gt; Math.abs(i)</code>

Tabelle 12.3 Unterschiedlicher Grad von Abkürzungen

<sup>3</sup> Wohl aber gibt es wie bei `{ throw new RuntimeException(); }` Ausnahmen, bei denen Lambda-Ausdrücke beides sind.

Kommen alle Abkürzungen zusammen, lässt sich etwa die Hälfte an Code einsparen. Aus `(int i) -> { return Math.abs(i); }` wird einfach `i -> Math.abs(i)`.

### Syntax-Hinweis

Nur bei genau einem Lambda-Parameter können die runden Klammern weggelassen werden, da es sonst Mehrdeutigkeiten gibt, für die es wieder komplexe Regeln zur Auflösung geben müsste. Heißt es etwa `foo( k, v -> { ... } )`, ist unklar, ob `foo` zwei Parameter deklariert. Ist das zweite Argument ein Lambda-Ausdruck, oder handelt es sich um nur genau einen Parameter, wobei dann ein Lambda-Ausdruck übergeben wird, der selbst zwei Parameter deklariert? Um Problemen wie diesen aus dem Weg zu gehen, können Entwickler auf den ersten Blick sehen, dass `foo( k, v -> { ... } )` eindeutig für zwei Argumente steht und `foo( (k, v) -> { ... } )` nur ein Argument übergibt.



### Unbenutzte Parameter in Lambda-Ausdrücken

Es kommt vor, dass ein Lambda-Ausdruck eine funktionale Schnittstelle implementiert, aber nicht jeder Parameter von Interesse ist. Als Beispiel schauen wir uns an:

```
interface Consumer<A> { void apply( A a ); }
```

Ein Konsument, der das Argument in Hochkommata ausgibt, sieht so aus:

```
Consumer<String> printQuoted = s -> System.out.printf( "%s", s );
printQuoted.accept( "Chris" ); // 'Chris'
```

Was ist nun, wenn ein Konsument auf das Argument gar nicht zugreifen möchte, weil zum Beispiel die aktuelle Zeit ausgegeben wird, aber der Code als `Consumer` vorliegen muss?

```
Consumer<String> printNow =
    s -> System.out.println( System.currentTimeMillis() );
```

Die Variable `s` in der Lambda-Parameterliste ist ungenutzt und wird vom Compiler auch als »unused« bemängelt.

Es gibt für unbenutzte Parameter keine spezielle Schreibweise und keine Möglichkeit, den Variablennamen wegzulassen und nur den Typ anzugeben.

Es gibt drei Ansätze, mit der Situation umzugehen. Der eine ist, für den Lambda-Parameter den Typnamen anzugeben – der wird dann als Variablenname verwendet, aber das ist vom Compiler her in Ordnung:

```
Consumer<String> printNow =
    String -> System.out.println( System.currentTimeMillis() );
```

Die Schreibweise ist ungewohnt, denn großgeschriebene Variablennamen zu verwenden, die zudem so heißen wie ein Typ, ist ein Bruch gegen die Konvention, funktioniert aber. Ge nutzt werden sollte der Lambda-Parameter aber nicht, `String -> System.out.println(String)` sieht einfach nur falsch aus.

Die zweite Möglichkeit kann so aussehen, zwei Unterstriche zu verwenden – die Nutzung eines Unterstrichs für einen Bezeichner ist seit Java 9 verboten, könnte aber in späteren Java-Versionen wieder eingeführt werden:

```
Consumer<String> printNow = __ -> System.out.println( System.currentTimeMillis() );
```

Die dritte Variante kann nicht immer eingesetzt werden, sondern nur dann, wenn es eine Methode gibt, die über eine Methodenreferenz angesprochen werden kann; in dieser Schreibweise gibt es keine Lambda-Parameter. Methodenreferenzen schauen wir in [Abschnitt 12.3](#) genauer an.

### 12.2.5 Die Umgebung der Lambda-Ausdrücke und Variabenzugriffe

Ein Lambda-Ausdruck »sieht« seine Umgebung genauso wie der Code, der vor oder nach dem Lambda-Ausdruck steht. Insbesondere hat ein Lambda-Ausdruck vollständigen Zugriff auf alle Eigenschaften der Klasse, genauso wie auch der einschließende äußere Block sie hat. Es gibt keinen besonderen Namensraum, sondern nur neue und vielleicht überdeckte Variablen durch die Parameter. Das ist einer der grundlegenden Unterschiede zwischen Lambda-Ausdrücken und inneren Klassen. Somit ist auch die Bedeutung von `this` und `super` bei Lambda-Ausdrücken und inneren Klassen unterschiedlich.

#### Lesender Zugriff auf finale, lokale Variablen/Parametervariablen

Lambda-Ausdrücke können problemlos auf Objektvariablen und Klassenvariablen lesend und schreibend zugreifen. Auf lokale Variablen sowie Methoden- oder Exception-Parameter hat ein Lambda-Ausdruck jedoch nur lesenden Zugriff, und die Variablen müssen final sein. Liegt ein Lambda-Ausdruck zum Beispiel in einer Schleife, kann er nicht auf den Schleifen zähler zugreifen, da sich dieser bei jeder Iteration ändert. (Anders sieht das bei der Variablen in der erweiterten `for`-Schleife aus, darauf kann ein Lambda-Ausdruck zugreifen.)

Dass eine Variable `final` ist, muss nicht extra mit einem Modifizierer geschrieben werden, aber sie muss *effektiv final* (engl. *effectively final*) sein. Effektiv final ist eine Variable, wenn sie nach der Initialisierung nicht mehr beschrieben wird, der Modifizierer `final` kann entfallen.

Ein Beispiel: Der Benutzer soll über eine Eingabe die Möglichkeit bekommen, zu bestimmen, ob String-Vergleiche mit unserem trimmenden `Comparator` unabhängig von der Groß-/Kleinschreibung stattfinden sollen:

**Listing 12.2** src/main/java/com/tutego/insel/lambda/CompareIgnoreCase.java, main

```
public static void main( String[] args ) {
    /*final*/ boolean compareIgnoreCase = new Scanner( System.in ).nextBoolean();
    Comparator<String> c = (s1, s2) -> compareIgnoreCase ?
        s1.trim().compareToIgnoreCase( s2.trim() ) :
        s1.trim().compareTo( s2.trim() );
    String[] words = { "M", "\nSkyfall", "Q", "\t\tAdele\t" };
    Arrays.sort( words, c );
    System.out.println( Arrays.toString( words ) );
}
```

Ob `compareIgnoreCase` von uns `final` gesetzt wird oder nicht, ist egal, denn die Variable wird hier effektiv `final` verwendet. Natürlich kann es nicht schaden, `final` als Modifizierer immer davorzusetzen, um dem Leser des Codes diese Tatsache bewusst zu machen.

Neu eingeschobene Lambda-Ausdrücke, die auf lokale Variablen oder Parametervariablen zugreifen, können also im Nachhinein zu Compilerfehlern führen. Folgendes Segment ist ohne Lambda-Ausdruck korrekt:

```
boolean compareIgnoreCase = new Scanner( System.in ).nextBoolean(); // 1
...
compareIgnoreCase = true; // 3
```

Schiebt sich zwischen Zeile 1 und 3 nachträglich ein Lambda-Ausdruck, der auf `compareIgnoreCase` zugreift, gibt es anschließend einen Compilerfehler. Allerdings liegt der Fehler nicht in Zeile 3, sondern beim Lambda-Ausdruck, denn die Variable `compareIgnoreCase` ist nach der Änderung nicht mehr effektiv `final`, was sie aber sein müsste, um in dem Lambda-Ausdruck verwendet zu werden.

### Schreibender Zugriff auf lokale Variablen/Parametervariablen? \*

Lambda-Ausdrücke können lokale Variablen nur lesen und nicht beschreiben – das Gleiche gilt übrigens für innere anonyme Klassen. Der Grund hat etwas damit zu tun, wo Variablen gespeichert werden: Objekt- und statische Variablen »leben« auf dem Heap, lokale Variablen und Parameter »leben« auf dem Stack. Wenn nun Threads ins Spiel kommen, ist es nicht unüblich, dass unterschiedliche Threads die Variablen vom Heap nutzen; dafür gibt es Synchronisationsmöglichkeiten. Allerdings kann ein Thread nicht auf lokale Variablen eines anderen Threads zugreifen, denn ein Thread hat erst einmal keinen Zugriff auf den Stack-Speicher eines anderen Threads. Grundsätzlich wäre das möglich, allerdings wollten die Oracle-Entwickler diesen Pfad nicht gehen. Beim Lesezugriff wird tatsächlich eine Kopie angelegt, so dass sie für einen anderen Thread sichtbar ist.

Die Einschränkung, dass äußere lokale Variablen von Lambda-Ausdrücken nur gelesen werden können, ist an sich etwas Gutes, denn die Beschränkung minimiert Fehler bei nebenläu-

figer Ausführung von Lambda-Ausdrücken: Arbeiten mehrere Threads Lambda-Ausdrücke ab und beschreiben diese eine lokale Variable, müsste andernfalls eine Thread-Synchronisation her.

Grundsätzlich verbietet nicht jede Programmiersprache das Schreiben von lokalen Variablen aus Lambda-Ausdrücken heraus. In C# kann ein Lambda-Ausdruck lokale Variablen beschreiben, sie »leben« dann nicht mehr auf dem Stack.

### Implementierungsdetail

Lambda-Ausdrücke müssen im Bytecode abgebildet werden, und der Compiler macht das mit Methoden. Wenn wir das JDK-Werkzeug `javap` auf den Bytecode mit dem Aufruf `javap -p CompareIgnoreCase` anwenden, folgt:

```
public class CompareIgnoreCase {
    public CompareIgnoreCase();
    public static void main(java.lang.String[]);
    private static int lambda$0(boolean, java.lang.String, java.lang.String);
}
```

Der Rumpf der privaten statischen Methode `lambda$0(...)` enthält den Codeblock `compareIgnoreCase ? s1.trim() ...`. Die JVM als Aufrufer der Methode übergibt den Inhalt der `compareIgnoreCase`-Variablen, und da es in Java nur als Parameterübergabe-Mechanismus Call by Value gibt, eine Kopie. Selbst wenn die Methode die Parametervariable ändern würde, käme die neue Belegung niemals aus der Methode heraus. Mit Behältern wie einem Array oder den speziellen `AtomicXXX`-Klassen aus dem `java.util.concurrent.atomic`-Paket lässt sich das Problem im Prinzip lösen. Denn greift ein Lambda-Ausdruck etwa auf das Array `boolean[] compareIgnoreCase = new boolean[1];` zu, so ist die Variable `compareIgnoreCase` selbst final, aber `compareIgnoreCase[0] = true;` ist erlaubt, denn es ist ein Schreibzugriff auf das Array, nicht auf die Variable `compareIgnoreCase`. Je nach Code besteht jedoch die Gefahr, dass Lambda-Ausdrücke parallel ausgeführt werden. Wird etwa ein Lambda-Ausdruck mit Veränderung auf diesem Array-Inhalt parallel ausgeführt, so ist der Zugriff nicht synchronisiert, und das Ergebnis kann »kaputt« sein, denn paralleler Zugriff auf Variablen muss immer koordiniert vorgenommen werden.

### Namensräume

Deklariert eine innere anonyme Klasse Variablen innerhalb der Methode, so sind diese immer »neu«, das heißt, die neuen Variablen überlagern vorhandene lokale Variablen aus dem äußeren Kontext. Die Variable `compareIgnoreCase` kann im Rumpf von `compare(...)` zum Beispiel problemlos neu deklariert werden:

```
boolean compareIgnoreCase = true;
Comparator<String> c = new Comparator<String>() {
```

```
@Override public int compare( String s1, String s2 ) {
    boolean compareIgnoreCase = false;           // völlig ok
    return ...
}
};
```

In einem Lambda-Ausdruck ist das nicht möglich, und Folgendes führt zu einer Fehlermeldung »variable compareIgnoreCase ist already defined« des Compilers:

```
boolean compareIgnoreCase = true;
Comparator<String> c = (s1, s2) -> {
    boolean compareIgnoreCase = false; // ☠ Compilerfehler
    return ...
}
```

### this-Referenz

Ein Lambda-Ausdruck unterscheidet sich von einer inneren (anonymen) Klasse auch darin, worauf die this-Referenz verweist:

- ▶ Beim Lambda-Ausdruck zeigt this immer auf das Objekt, in dem der Lambda-Ausdruck eingebettet ist.
- ▶ Bei einer inneren Klasse referenziert this die innere Klasse, und die ist ein komplett neuer Typ.

Folgendes Beispiel macht das deutlich:

**Listing 12.3** src/main/java/com/tutego/insel/lambda/InnerVsLambdaThis.java, Ausschnitt

```
class InnerVsLambdaThis {
    InnerVsLambdaThis() {
        Runnable lambdaRun = () -> System.out.println( this.getClass().getName() );
        Runnable innerRun = new Runnable() {
            @Override public void run() { System.out.println( this.getClass().getName()); }
        };

        lambdaRun.run();      // InnerVsLambdaThis
        innerRun.run();       // InnerVsLambdaThis$1
    }
    public static void main( String[] args ) {
        new InnerVsLambdaThis();
    }
}
```

Als Erstes nutzen wir `this` in einem Lambda-Ausdruck im Konstruktor der Klasse `InnerVsLambdaThis`. Damit referenziert `this` das neu gebaute `InnerVsLambdaThis`-Objekt. Bei der inneren Klasse referenziert `this` ein anderes Exemplar, und zwar vom Typ `Runnable`. Da es bei anonymen Klassen keinen Namen hat, trägt es lediglich die Kennung `InnerVsLambdaThis$1`.

### Rekursive Lambda-Ausdrücke

Lambda-Ausdrücke können auf sich selbst verweisen. Da aber ein `this` zur Selbstreferenz nicht funktioniert, ist ein kleiner Umweg nötig. Erst muss eine Objekt- oder eine Klassenvariable deklariert werden, dann muss dieser Variablen ein Lambda-Ausdruck zugewiesen werden, und dann kann der Lambda-Ausdruck auf diese Variable zugreifen und einen rekursiven Aufruf starten. Für den Klassiker der Fakultät sieht das so aus:

**Listing 12.4** src/main/java/com/tutego/insel/lambda/RecursiveFactLambda.java, Ausschnitt

```
public class RecursiveFactLambda {

    public static IntFunction<Integer> fact =
        n -> (n == 0) ? 1 : n * RecursiveFactLambda.fact.apply( n - 1 );

    public static void main( String[] args ) {
        System.out.println( fact.apply( 5 ) ); // 120
    }
}
```

`IntFunction` ist eine funktionale Schnittstelle aus dem Paket `java.util.function` mit einer Operation `T apply(int i)`. Dabei ist `T` ein generischer Rückgabetyp, den wir hier mit `Integer` belegt haben. Es funktioniert übrigens nicht, `n * fact.apply(n - 1)` zu schreiben, da der Compiler dann meldet: »Cannot reference a field before it is defined«.

`fact` hätte genauso gut als normale Methode deklariert werden können. Großartige Vorteile bietet die Schreibweise mit Lambda-Ausdrücken hier nicht. Zumal jetzt auch der Begriff *anonyme Methode* nicht mehr so richtig passt, da der Lambda-Ausdruck ja doch einen Namen hat, nämlich `fact`.

#### 12.2.6 Ausnahmen in Lambda-Ausdrücken

Lambda-Ausdrücke sind Implementierungen von funktionalen Schnittstellen, und bisher haben wir noch nicht die Frage betrachtet, was passiert, wenn der Codeblock vom Lambda-Ausdruck eine Ausnahme auslöst, und wer diese auffangen muss.

### Ausnahmen im Codeblock eines Lambda-Ausdrucks

In `java.util.function` gibt es eine funktionale Schnittstelle `Predicate`, deren Deklaration im Kern wie folgt ist:

```
public interface Predicate<T> { boolean test( T t ); }
```

Ein `Predicate` führt einen Test durch und liefert wahr oder falsch als Ergebnis. Ein Lambda-Ausdruck kann diese Schnittstelle implementieren. Nehmen wir an, wir wollten testen, ob eine Datei die Länge 0 hat, um etwa Dateileichen zu finden. In einer ersten Idee greifen wir auf die existierende `Files`-Klasse zurück, die `size(...)` anbietet:

```
Predicate<Path> isEmptyFile = path -> Files.size( path ) == 0; // ☠ Compilerfehler
```

Das Problem dabei ist, dass `Files.size(...)` eine `IOException` auslöst, die behandelt werden muss, und zwar *nicht* vom Block, in dem der Lambda-Ausdruck als Ganzes steht, sondern vom Code im Lambda-Ausdruck selbst. Das schreibt der Compiler so vor. Folgendes ist keine Lösung:

```
try {
    Predicate<Path> isEmptyFile = path -> Files.size( path ) == 0; // ☠
} catch ( IOException e ) { ... }
```

sondern nur:

```
Predicate<Path> isEmptyFile = path -> {
    try {
        return Files.size( path ) == 0;
    } catch ( IOException e ) { return false; }
};
```

Die Eigenschaft, die Java fehlt, nennt sich *Exception-Transparenz*, und hier ist deutlich der Unterschied zwischen geprüften und ungeprüften Ausnahmen zu sehen. Bei der Exception-Transparenz wäre keine Ausnahmebehandlung im Lambda-Ausdruck nötig und an einer übergeordneten Stelle möglich. Doch da diese Möglichkeit in Java fehlt, bleibt uns nur übrig, geprüfte Ausnahmen in Lambda-Ausdrücken direkt zu behandeln.

### Funktionale Schnittstellen mit throws-Klausel

Ungeprüfte Ausnahmen können immer auftreten und führen (nicht abgefangen) wie üblich zum Abbruch des Threads. Eine `throws`-Klausel an den Methoden/Konstruktoren ist dafür nicht nötig. Doch können funktionale Schnittstellen eine `throws`-Klausel mit geprüften Ausnahmen deklarieren, und die Implementierung einer funktionalen Schnittstelle kann logischerweise geprüfte Ausnahmen auslösen.

Eine Deklaration wie Callable aus dem Paket `java.util.concurrent` macht das deutlich:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Das könnte durch folgenden Lambda-Ausdruck realisiert werden:

```
Callable<Integer> randomDice = () -> (int)(Math.random() * 6) + 1;
```

Der Aufruf von `call()` auf einem `randomDice` muss mit einer Ausnahmebehandlung einhergehen, da `call()` eine `Exception` auslöst, etwa so:

```
try {
    System.out.println( randomDice.call() );
    System.out.println( randomDice.call() );
}
catch ( Exception e ) { ... }
```

Dass der Aufrufer die Ausnahme behandeln muss, ist klar. Die Deklaration des Lambda-Ausdrucks enthält keinen Hinweis auf die Ausnahme, das ist ein Unterschied zum vorangegangenen Abschnitt.



### Designtipp

Ausnahmen in Methoden funktionaler Schnittstellen schränken den Nutzen stark ein, und daher löst keine der funktionalen Schnittstellen aus etwa `java.util.function` eine geprüfte Ausnahme aus. Der Grund ist einfach, denn jeder Methodenaufrufer müsste sonst entweder die Ausnahme weiterleiten oder behandeln.<sup>4</sup>

Um die Einschränkungen und Probleme mit einer `throws`-Klausel noch etwas deutlicher zu machen, stellen wir uns vor, dass die funktionale Schnittstelle `Predicate` an der Operation `ein throws Exception` (vom Sinn des Typs `Exception` an sich einmal abgesehen) enthält:

```
interface Predicate<T> { boolean test( T t ) throws Exception; } // Was wäre, wenn?
```

Die Konsequenz wäre, dass jeder Aufrufer von `test(...)` nun seinerseits die `Exception` in die Hände bekäme und sie auffangen oder weiterleiten müsste. Leitet der `test(...)`-Aufrufer mit `throws Exception` die Ausnahme weiter nach oben, bekommen wir plötzlich an allen Stellen `ein throws Exception` in die Methodensignatur, was auf keinen Fall gewünscht ist. So enthält

---

<sup>4</sup> Von `Callable` gibt es zwar Nutzer, die mit Nebenläufigkeit (daher das Paket `java.util.concurrent`) in Zusammenhang stehen, aber keine weiteren Verwendungen in der Java-Bibliothek, von zwei Beispielen aus `javax.tools` abgesehen. Mit `java.util.function.Supplier` existiert eine entsprechende Alternative ohne `throws`-Klausel.

jetzt etwa ArrayList eine Deklaration von removeIf(Predicate filter); hier müsste sich dann removeIf(...) – das letztendlich filter.test(...) aufruft – mit der Testausnahme herumärgern, und removeIf(Predicate filter) throws Exception ist keine gute Sache.

### Von geprüft nach ungeprüft

Geprüfte Ausnahmen sind in Lambda-Ausdrücken nicht schön. Eine Lösung ist, Code, der geprüfte Ausnahmen auslöst, zu verpacken und die geprüfte Ausnahme in einer ungeprüften zu manteln. Das kann etwa so aussehen:

**Listing 12.5** src/main/java/com/tutego/insel/lambda/PredicateWithException.java, Ausschnitt

```
public class PredicateWithException {
    @FunctionalInterface
    public interface ExceptionalPredicate<T,E extends Exception> {
        boolean test( T t ) throws E;
    }
    public static <T> Predicate<T> asUncheckedPredicate(
            ExceptionalPredicate<T,Exception> predicate ) {
        return t -> {
            try {
                return predicate.test( t );
            }
            catch ( Exception e ) {
                throw new RuntimeException( e.getMessage(), e );
            }
        };
    }
    public static void main( String[] args ) {
        Predicate<Path> isEmptyFile =
            asUncheckedPredicate( path -> Files.size( path ) == 0 );
        System.out.println( isEmptyFile.test( Paths.get( "c://" ) ) );
    }
}
```

Die Schnittstelle ExceptionalPredicate ist ein Prädikat mit optionaler Ausnahme. In der eigenen Hilfsmethode asUncheckedPredicate(ExceptionalPredicate) nehmen wir so ein ExceptionalPredicate an und packen es in ein Predicate, das die Methode zurückgibt. Geprüfte Ausnahmen werden in eine ungeprüfte Ausnahme vom Typ RuntimeException gesetzt. Somit muss Predicate keine geprüfte Ausnahme weiterleiten, was es ja laut Deklaration auch nicht kann.

Die Java-Bibliothek selbst bringt keine Ummantelungen dieser Art mit. Es gibt nur eine interne Methode, die etwas Vergleichbares tut:

**Listing 12.6** java.nio.file.Files.java, asUncheckedRunnable(...)

```
/*
 * Convert a Closeable to a Runnable by converting checked IOException
 * to UncheckedIOException
 */
private static Runnable asUncheckedRunnable( Closeable c ) {
    return () -> {
        try {
            c.close();
        }
        catch ( IOException e ) {
            throw new UncheckedIOException( e );
        }
    };
}
```

Hier kommt die Klasse `UncheckedIOException` zum Einsatz. Diese ist eine ungeprüfte Ausnahme, die als Wrapper-Klasse für Ein-/Ausgabefehler genutzt wird. Wir finden die `UncheckedIOException` etwa bei `lines()` von `BufferedReader` bzw. `Files`, die einen `Stream<String>` mit Zeilen liefert – geprüfte Ausnahmen sind hier nur im Weg.

### 12.2.7 Klassen mit einer abstrakten Methode als funktionale Schnittstelle? \*

Als die Entwickler der Sprache Java die Lambda-Ausdrücke diskutierten, stand auch die Frage im Raum, ob abstrakte Klassen, die nur über eine abstrakte Methode verfügen, ebenfalls für Lambda-Ausdrücke genutzt werden können.<sup>5</sup> Sie entschieden sich dagegen, unter anderem deswegen, weil bei der Implementierung von Schnittstellen die JVM weitreichende Optimierungen vornehmen kann. Und bei Klassen wird das schwierig. Das liegt auch daran, dass ein Konstruktor umfangreiche Initialisierungen mit Seiteneffekten vornimmt (die Konstruktoren aller Oberklassen nicht zu vergessen) sowie Ausnahmen auslösen könnte. Gewünscht ist aber nur die Ausführung einer Implementierung der funktionalen Schnittstelle und kein anderer Code.

Es gibt nun im JDK einige abstrakte Klassen, die genau eine abstrakte Methode vorschreiben, etwa `java.util.TimerTask`. Solche Klassen können nicht über einen Lambda-Ausdruck realisiert werden; hier müssen Entwickler weiterhin zu Klassenimplementierungen greifen, und die kürzeste Lösung ist eine innere anonyme Klasse. Eigene Hilfsklassen können natürlich den Code etwas abkürzen, aber eben nur mithilfe einer eigenen Implementierung.

Wer abstrakte Methoden mit Lambda-Ausdrücken implementieren möchte, kann mit Hilfsklassen arbeiten. Denn wenn eine Hilfsklasse funktionale Schnittstellen einsetzt, so können

---

<sup>5</sup> Früher wurde hier die Abkürzung *SAM* (*Single Abstract Method*) genutzt.

Lambda-Ausdrücke wieder ins Spiel kommen, indem die Implementierung der abstrakten Methode an den Lambda-Ausdruck weiterleitet. Nehmen wir das Beispiel für TimerTask und gehen zwei unterschiedliche Strategien der Implementierung durch. Mit Delegation sieht das so aus:

**Listing 12.7** src/main/java/com/tutego/insel/lambda/TimerTaskLambda.java, Ausschnitt

```
class TimerTaskLambda {
    public static TimerTask createTimerTask( Runnable runnable ) {
        return new TimerTask() {
            @Override public void run() { runnable.run(); }
        };
    }
    public static void main( String[] args ) {
        new Timer().schedule( createTimerTask( () -> System.out.println("Hi") ), 500 );
    }
}
```

Mit Vererbung erhalten wir:

```
public class LambdaTimerTask extends TimerTask {
    private final Runnable runnable;

    public LambdaTimerTask( Runnable runnable ) {
        this.runnable = runnable;
    }

    @Override public void run() { runnable.run(); }
}
```

Der Aufruf erfolgt dann statt über createTimerTask(...) mit dem Konstruktor:

```
new Timer().schedule( new LambdaTimerTask( () -> System.out.println("Hi") ), 500 );
```

## 12.3 Methodenreferenz

Je größer Softwaresysteme werden, desto wichtiger werden Dinge wie Klarheit, Wiederverwendbarkeit und Dokumentation. Wir haben für unseren String-Comparator eine Implementierung geschrieben, anfangs über eine innere Klasse, später über einen Lambda-Ausdruck. In jedem Fall haben wir Code geschrieben. Doch was wäre, wenn eine Utility-Klasse schon eine Implementierung mitbringen würde? Dann könnte der Lambda-Ausdruck natürlich an die vorhandene Implementierung delegieren, und wir sparen Code. Schauen wir uns das mal an einem Beispiel an:

```

class StringUtils {
    public static int compareTrimmed( String s1, String s2 ) {
        return s1.trim().compareTo( s2.trim() );
    }
}
public class CompareIgnoreCase {
    public static void main( String[] args ) {
        String[] words = { "A", "B", "a" };
        Arrays.sort( words,
                    (String s1, String s2) -> StringUtils.compareTrimmed(s1, s2) );
        System.out.println( Arrays.toString( words ) );
    }
}

```

Auffällig ist hier, dass die referenzierte Methode `compareTrimmed(String, String)` von den Parametertypen und vom Rückgabetyp genau auf die `compare(..)-Methode` eines `Comparator` passt. Für genau solche Fälle gibt es eine weitere syntaktische Verkürzung, sodass im Code kein Lambda-Ausdruck, sondern nur noch ein Methodenverweis notwendig ist.

### Definition

Eine *Methodenreferenz* ist ein Verweis auf eine Methode, ohne diese jedoch aufzurufen. Syntaktisch trennen zwei Doppelpunkte den Typnamen oder die Referenz auf der linken Seite von dem Methodennamen auf der rechten.

Die Zeile

```
Arrays.sort( words, (String s1, String s2) -> StringUtils.compareTrimmed(s1, s2) );
```

lässt sich mit einer Methodenreferenz abkürzen zu:

```
Arrays.sort( words, StringUtils::compareTrimmed );
```

Die Sortiermethode erwartet vom `Comparator` eine Methode, die zwei `Strings` annimmt und eine Ganzzahl zurückgibt. Der Name der Klasse und der Name der Methode sind unerheblich, weshalb an dieser Stelle eine Methodenreferenz eingesetzt werden kann.



### Hinweis

Es ist nicht möglich, eine spezielle Methode über die Methodenreferenz auszuwählen. Eine Angabe wie `String::valueOf` oder `Arrays::sort` ist relativ breit – bei Letzterem wählt der Compiler eine der 18 passenden überladenen Methoden aus. Da kann es passieren, dass der Compiler eine falsche Methode auswählt. In dem Fall muss ein expliziter Lambda-Ausdruck eine Mehrdeutigkeit auflösen. Bei generischen Typen kann zum Beispiel `List<String>::length` oder auch `List::length` stehen, auch hier erkennt der Compiler wieder alles selbst.



Eine Methodenreferenz ist wie ein Lambda-Ausdruck ein Exemplar einer funktionalen Schnittstelle, jedoch für eine existierende Methode einer bekannten Klasse. Wie üblich bestimmt der Kontext, von welchem Typ genau der Ausdruck ist.

### Hinweis

Gleicher Code für eine Methodenreferenz kann zu komplett unterschiedlichen Typen führen – der Kontext macht den Unterschied:

```
Comparator<String> c1 = StringUtils::compareTrimmed;
BiFunction<String, String, Integer> c2 = StringUtils::compareTrimmed;
```

### 12.3.1 Varianten von Methodenreferenzen

Im Beispiel ist die Methode `compareTrimmed(...)` statisch, und links vom Doppelpunkt steht der Name eines Typs. Allerdings kann beim Einsatz eines Typnamens die Methode auch nichtstatisch sein, `String::length` ist so ein Beispiel. Das wäre eine Funktion, die ein `String` auf ein `int` abbildet, in Code:

```
Function<String, Integer> len = String::length;
```

Insgesamt gibt es drei Schreibweisen für Methodenreferenzen:

Methodenreferenz auf eine ...	Lambda-Ausdruck	Syntax für Methodenreferenz
... statische Methode	(args) -> Klasse.staticMethode(args)	Klasse::staticMethode
... Objektmethode	(args) -> obj.objektMethode(args)	obj::objektMethode
... Objektmethode eines Typs	(obj, args) -> obj.objektMethode(args)	Typvonobj::objektMethode

Tabelle 12.4 Unterschiedliche Methodenreferenzen

### Beispiele

Während `String::length` eine Funktion ist, wäre `string::length` ein Supplier, unter der Annahme, dass `string` eine Referenzvariable ist:

```
String string = "Goll";
Supplier<Integer> len = string::length;
System.out.println( len.get() ); // 4
```

System.out ist eine Referenz, und eine Methode wie println(...) kann an einen Consumer gebunden werden. Es ist aber auch ein Runnable, weil es println() auch ohne Parameterliste gibt:

```
Consumer<String> out = System.out::println;
out.accept( "Kates kurze Kleider" );
Runnable out = System.out::println;
out.run();
```

Ist eine Hauptmethode mit main(String... args) deklariert, so ist das auch ein Runnable:

```
Runnable r = JavaApplication1::main;
```

Anders wäre das bei main(String[]), hier ist ein Parameter zwingend, doch ein Vararg kann auch leer sein.

Besonders bei Comparatoren benötigen wir Schlüssel-Extraktoren, die im Kern die Getter aufrufen. Statt p -> p.getName() passt hier die Methodenreferenz Person::getName.

### this und super sind möglich

Anstatt den Namen einer Referenzvariablen zu wählen, kann auch this das Objekt beschreiben, und auch super ist möglich. this ist praktisch, wenn die Implementierung einer funktionalen Schnittstelle auf eine Methode der eigenen Klasse delegieren möchte. Wenn zum Beispiel eine lokale Methode compareTrimmed(...) in der Klasse existieren würde, in der auch der Lambda-Ausdruck steht, und wenn diese Methode als Comparator in Arrays.sort(...) verwendet werden sollte, könnte es heißen: Arrays.sort(words, this::compareTrimmed).

### Was soll das alles?

Einem Einsteiger in die Sprache Java wird dieses Sprache-Feature wie der größte Zauber auf Erden vorkommen, und auch Java-Profis bekommen hier zittrige Finger, entweder vor Furcht oder Aufregung ... In der Vergangenheit musste in Java sehr viel Code explizit geschrieben werden, aber mit diesen neuen Methodenreferenzen erkennt und macht der Compiler vieles von selbst.

Nützlich wird diese Eigenschaft mit den funktionalen Bibliotheken bei der Stream-API, die ein eigenes Kapitel im zweiten Band einnehmen. Hier nur ein kurzer Vorgesmack:

```
Object[] words = { " ", '3', null, "2", 1, "" };
Arrays.stream( words ) // " ", '3', null, "2", 1, ""
    .filter( Objects::nonNull ) // " ", '3', "2", 1, ""
    .map( Objects::toString ) // " ", "3", "2", "1", ""
    .map( String::trim ) // "", "3", "2", "1", ""
```

```
.filter( s -> ! s.isEmpty() )      // "3", "2", "1"
.map( Integer::parseInt )          // 3, 2, 1
.sorted()                          // 1, 2, 3
.forEach( System.out::println );   // 1 2 3
```

## 12.4 Konstruktorreferenz

Um ein Objekt aufzubauen, nutzen wir das Schlüsselwort `new`. Das führt zum Aufruf eines Konstruktors, dem sich optional Argumente übergeben lassen. Die Java-API deklariert aber auch Typen, von denen sich keine direkten Exemplare mit `new` aufbauen lassen. Stattdessen gibt es Erzeuger, deren Aufgabe es ist, Objekte aufzubauen. Die Erzeuger können statische oder auch nichtstatische Methoden sein:

Konstruktor ...	... erzeugt:	Erzeuger ...	... baut:
<code>new Integer( "1" )</code>	<code>Integer</code>	<code>Integer.valueOf( "1" )</code>	<code>Integer</code>
<code>new File( "dir" )</code>	<code>File</code>	<code>Paths.get( "dir" )</code>	<code>Path</code>
<code>new BigInteger( val )</code>	<code>BigInteger</code>	<code>BigInteger.valueOf( val )</code>	<code>BigInteger</code>

Tabelle 12.5 Beispiele für Konstruktoren und Erzeugermethoden

Beide, Konstruktoren und Erzeuger, lassen sich als spezielle Funktionen sehen, die von einem Typ in einen anderen Typ konvertieren. Damit eignen sie sich perfekt für Transformationen, und in einem Beispiel haben wir das schon eingesetzt:

```
Arrays.stream( words )
    . . .
    .map( Integer::parseInt )
    . . .
```

`Integer.parseInt(string)` ist eine Methode, die sich einfach mit einer Methodenreferenz fassen lässt, und zwar als `Integer::parseInt`. Aber was ist mit Konstruktoren? Auch sie transformieren! Statt `Integer.parseInt(string)` hätte ja auch `new Integer(string)` eingesetzt werden können.

Wo Methodenreferenzen statische Methoden und Objektmethoden angeben können, bieten *Konstruktorreferenzen* die Möglichkeit, Konstruktoren anzugeben, sodass diese als Erzeuger an anderer Stelle übergeben werden können. Damit lassen sich elegant Konstruktoren als Erzeuger angeben, und zwar auch von einer Klasse, die nicht über Erzeugermethoden verfügt. Wie auch bei Methodenreferenzen spielt eine funktionale Schnittstelle eine entscheidende

Rolle, doch dieses Mal ist es die Methode der funktionalen Schnittstelle, die mit ihrem Aufruf zum Konstruktorauftruf führt. Wo syntaktisch bei Methodenreferenzen rechts vom Doppelpunkt ein Methodenname steht, ist dies bei Konstruktorreferenzen ein `new`.<sup>6</sup> Also ergibt sich alternativ zu

```
.map( Integer::parseInt )           // Methode Integer.parseInt(String)
```

in unserem Beispiel das Ergebnis mittels:

```
.map( Integer::new )               // Konstruktor Integer(String)
```

Mit der Konstruktorreferenz gibt es vier Möglichkeiten, funktionale Schnittstellen zu implementieren; die drei verbleibenden Varianten sind Lambda-Ausdrücke, Methodenreferenzen und klassische Implementierung über eine Klasse.



### Beispiel

Die funktionale Schnittstelle sei:

```
interface DateFactory { Date create(); }
```

Die folgende Konstruktorreferenz bindet den Konstruktor an die Methode `create()` der funktionalen Schnittstelle:

```
DateFactory factory = Date::new;
System.out.print( factory.create() ); // zum Beispiel Fri Oct 06 22:34:24 CET 2017
```

Beziehungsweise die letzten beiden Zeilen zusammengefasst:

```
System.out.println( ((DateFactory)Date::new).create() );
```

Soll nur der parameterlose Konstruktor aufgerufen werden, muss die funktionale Schnittstelle nur eine Methode besitzen, die keinen Parameter besitzt und etwas zurückliefert. Der Rückgabetyp der Methode muss natürlich mit dem Klassentyp zusammenpassen. Das gilt für den Typ `DateFactory` aus unserem Beispiel. Doch es geht noch etwas generischer, zum Beispiel mit der vorhandenen funktionalen Schnittstelle `Supplier`, wie wir gleich sehen werden.

In der API finden sich oftmals Parameter vom Typ `Class`, die als Typangabe dazu verwendet werden, dass über den `Constructor` der `Class` mit der Methode `newInstance()` Exemplare gebildet werden. Der Einsatz von `Class` lässt sich durch eine funktionale Schnittstelle ersetzen, und Konstruktorreferenzen lassen sich an Stelle von `Class`-Objekten übergeben.

---

<sup>6</sup> Da `new` ein Schlüsselwort ist, kann keine Methode so heißen; der Identifizierer ist also sicher.

#### 12.4.1 Parameterlose und parametrisierte Konstruktoren

Beim parameterlosen Konstruktor hat die Methode nur eine Rückgabe, bei einem parametrisierten Konstruktor muss die Methode der funktionalen Schnittstelle natürlich über eine kompatible Parameterliste verfügen:

Konstruktor	Date()	Date( <b>long</b> t)
Kompatible funktionale Schnittstelle	interface DateFactory { Date create(); }	interface DateFactory { Date create( <b>long</b> t); }
Konstruktorreferenz	DateFactory factory = Date::new;	DateFactory factory = Date::new;
Aufruf	factory.create();	factory.create(1);

Tabelle 12.6 Standard- und parametrisierter Konstruktor mit korrespondierenden funktionalen Schnittstellen

#### Hinweis

Kommt die Typ-Inferenz des Compilers an ihre Grenzen, sind zusätzliche Typinformationen gefordert. In diesem Fall werden hinter dem Doppelpunkt in eckigen Klammern weitere Angaben gemacht, etwa Klasse::<Typ1, Typ2>new.



#### 12.4.2 Nützliche vordefinierte Schnittstellen für Konstruktorreferenzen

Die für einen parameterlosen Konstruktor passende funktionale Schnittstelle muss eine Rückgabe besitzen und keinen Parameter annehmen; die funktionale Schnittstelle für einen parametrisierten Konstruktor muss eine entsprechende Parameterliste haben. Es kommt nun häufig vor, dass der Konstruktor ein parameterloser ist oder genau einen Parameter annimmt. Hier ist es vorteilhaft, dass für diese beiden Fälle die Java-API zwei praktische (generisch deklarierte) funktionale Schnittstellen mitbringt:

Funktionale Schnittstelle	Funktions-Deskriptor	Abbildung	Passt auf
Supplier<T>	T get()	() → T	parameterloser Konstruktor
Function<T,R>	R apply(T t)	(T) → R	einfacher parametrisierter Konstruktor

Tabelle 12.7 Vorhandene funktionale Schnittstellen als Erzeuger



### Beispiel

Die funktionale Schnittstelle `Supplier<T>` hat eine `T get()`-Methode, die wir mit dem parameterlosen Konstruktor von `Date` verbinden können:

```
Supplier<Date> factory = Date:::new;
System.out.print( factory.get() );
```

Wir nutzen `Supplier` mit dem Typparameter `Date`, was den parametrisierten Typ `Supplier<Date>` ergibt, und `get()` liefert folglich den Typ `Date`. Der Aufruf `factory.get()` führt zum Aufruf des Konstruktors.

### Ausblick \*

Besonders interessant werden die Konstruktorreferenzen mit den neuen Bibliotheksmethoden der Stream-API. Nehmen wir eine Liste vom Typ `Zeitstempel` an. Der Konstruktor `Date(long)` nimmt einen solchen Zeitstempel entgegen, und mit einem `Date`-Objekt können wir Vergleiche vornehmen, etwa ob ein Datum hinter einem anderen Datum liegt. Folgendes Beispiel listet alle Datumswerte auf, die nach dem 1.1.2012 liegen:

```
Long[] timestamps = { 2432558632L, // Thu Jan 29 04:42:38 CET 1970
                     1575975986345L }; // Tue Dec 10 12:06:26 CET 2019
Date givenYear = new GregorianCalendar( 2018, Calendar.JANUARY, 1 ).getTime();
Arrays.stream( timestamps )
    .map( Date:::new )
    .filter( givenYear::before )
    .forEach( System.out::println ); // Tue Dec 10 12:06:26 CET 2019
```

Die Konstruktorreferenz `Date:::new` hilft dabei, das `long` mit dem Zeitstempel in ein `Date`-Objekt zu konvertieren.

### Denksportaufgabe

Ein Konstruktor kann als `Supplier` oder `Function` gelten. Problematisch sind mal wieder geprüfte Ausnahmen. Der Leser soll überlegen, ob der Konstruktor `URI(String str)` throws `URIException` über `URI:::new` angesprochen werden kann.

## 12.5 Implementierung von Lambda-Ausdrücken \*

Als die Compilerentwickler einen Prototyp für Lambda-Ausdrücke bauten, setzten sie diese technisch mit geschachtelten Klassen um. Doch das war nur in der Testphase, denn geschachtelte Klassen sind für die JVM komplett Klassen und schwergewichtig. Das Laden und Initialisieren ist relativ teuer und würde bei den vielen kleinen Lambda-Ausdrücken einen

großen Overhead darstellen. Daher nutzt die Implementierung ein *invokedynamic* genanntes Konstrukt. Das hat den großen Vorteil, dass die Laufzeitumgebung viel Gestaltungsraum in der Optimierung hat. Geschachtelte Klassen sind nur eine mögliche technische Umsetzung für Lambda-Ausdrücke, *invokedynamic* ist sozusagen die deklarative Variante, und geschachtelte Klassen sind die imperative. Letztendlich ist der Overhead mit *invokedynamic* gering, und Programmcode von geschachtelten Klassen hin zu Lambda-Ausdrücken zu refaktorieren führt zu kleinen Bytecodedateien. Von der Performance her unterscheiden sich Lambda-Ausdrücke und die Implementierung funktionaler Schnittstellen und Klassen nicht, eher ist die Optimierung auf der Seite der JVM zu finden, die es mit weniger Klassendateien zu tun hat. Umgekehrt bedeutet das auch: Wenn Entwickler ihre alte vorhandene Implementierung von funktionalen Schnittstellen durch Lambda-Ausdrücke ersetzen, wird der Bytecode kompakter, da ein kleines *invokedynamic* viel kürzer ist als komplexe neue Klassendateien.

## 12.6 Funktionale Programmierung mit Java

### 12.6.1 Programmierparadigmen: imperativ oder deklarativ

In irgendeiner Weise muss ein Entwickler sein Problem in Programmform beschreiben, damit der Computer es letztendlich ausführen kann. Hier gibt es verschiedene Beschreibungsformen, die wir *Programmierparadigmen* nennen. Bisher haben wir uns immer mit der imperativen Programmierung beschäftigt, bei der Anweisungen im Mittelpunkt stehen. Wir haben im Deutschen den Imperativ, also die Befehlsform, die sehr gut mit dem Programmierstil vergleichbar ist, denn es handelt sich in beiden Fällen um Anweisungen der Art »tue dies, tue das«. Diese »Befehle« mit Variablen, Fallunterscheidungen, Sprüngen beschreiben das Programm und den Lösungsweg.

Zwar ist imperativer Programmierung die technisch älteste, aber nicht die einzige Form, Programme zu beschreiben; es gibt daneben die deklarative Programmierung, die nicht das Wie zur Problemlösung beschreibt, sondern das Was, also was eigentlich gefordert ist, ohne sich in genauen Abläufen zu verstricken. Auf den ersten Blick klingt das abstrakt, aber jeder, der schon einmal

- ▶ eine Selektion wie »\*.html« auf der Kommandozeile/im Explorer-Suchfeld getätigt,
- ▶ eine Datenbankabfrage mit SQL geschrieben,
- ▶ eine XML-Selektion mit XQuery genutzt,
- ▶ ein Build-Skript mit Ant oder make formuliert oder
- ▶ eine XML-Transformation mit XSLT beschrieben hat,

wird das Prinzip kennen.

Bleiben wir kurz bei SQL, um einen Punkt deutlich zu machen. Natürlich führt im Endeffekt die CPU die Abarbeitung der Tabellen und die Auswertungen der Ergebnisse rein imperativ aus, doch es geht um die Programmbeschreibung auf einem höheren Abstraktionsniveau. Deklarative Programme sind üblicherweise wesentlich kürzer, und damit kommen weitere Vorteile wie leichtere Erweiterbarkeit, Verständlichkeit ins Spiel. Da deklarative Programme oftmals einen mathematischen Hintergrund haben, lassen sich die Beschreibungen leichter formal in ihrer Korrektheit beweisen.

Deklarative Programmierung ist ein Programmierstil, und eine deklarative Beschreibung braucht eine Art »Ablaufumgebung«, denn SQL kann zum Beispiel keine CPU direkt ausführen. Aber anstatt nur spezielle Anwendungsfälle wie Datenbank- oder XML-Abfragen zu behandeln, können auch typische Algorithmen deklarativ formuliert werden, und zwar mit funktionaler Programmierung. Damit sind imperative Programme und funktionale Programme gleich mächtig in ihren Möglichkeiten.

### 12.6.2 Funktionale Programmierung und funktionale Programmiersprachen

Bei der funktionalen Programmierung stehen Funktionen im Mittelpunkt und ein im Idealfall zustandsloses Verhalten, in dem viel mit Rekursion gearbeitet wird. Ein typisches Beispiel ist die Berechnung der Fakultät. Es ist  $n! = 1 \times 2 \times 3 \times \dots \times n$ , und mit Schleifen und Variablen, dem imperativen Weg, sieht sie so aus:

```
public static int factorial( int n ) {
    int result = 1;
    for ( int i = 1; i <= n; i++ )
        result *= i;
    return result;
}
```

Deutlich sind die vielen Zuweisungen und die Fallunterscheidung durch die Schleife abzulesen, die typischen Indikatoren für imperative Programme. Der Schleifenzähler erhöht sich, damit kommt Zustand in das Programm, denn der aktuelle Index muss ja irgendwo im Speicher gehalten werden. Bei der rekursiven Variante ist das ganz anders, hier gibt es keine Zuweisungen im Programm, und die Schreibweise erinnert an die mathematische Definition:

```
public static int factorial( int n ) {
    return n == 0 ? 1 : n * factorial( n - 1 );
}
```

Mit der funktionalen Programmierung haben wir eine echte Alternative zur imperativen Programmierung. Die Frage ist nur: Mit welcher Programmiersprache lassen sich funktionale Programme schreiben? Im Grunde mit jeder höheren Programmiersprache! Denn funktional zu programmieren ist ja ein Programmierstil, und Java unterstützt funktionale Programmierung, wie wir am Beispiel mit der Fakultät ablesen können. Da das im Prinzip schon

alles ist, stellt sich die Frage, warum funktionale Programmierung einen so schweren Stand hat und bei den Entwicklern gefürchtet ist. Das hat mehrere Gründe:

- ▶ **Lesbarkeit:** Am Anfang der funktionalen Programmiersprachen steht historisch LISP aus dem Jahr 1958, eine sehr flexible, aber ungewohnt zu lesende Programmiersprache. Unser Fakultät sieht in LISP so aus: (defun factorial (n) (if (= n 1) 1 (\* n (factorial (- n 1))))). Die ganzen Klammern machen die Programme nicht einfach lesbar, und die Ausdrücke stehen in der Präfix-Notation  $- n 1$  statt der üblichen Infix-Notation  $n - 1$ . Bei anderen funktionalen Programmiersprachen ist es anders, dennoch führt das zu einem gewissen Vorurteil, dass alle funktionalen Programmiersprachen schlecht lesbar seien.
- ▶ **Performance und Speicherverbrauch:** Ohne clevere Optimierungen von Seiten des Compilers und der Laufzeitumgebung führen insbesondere rekursive Aufrufe zu prall gefüllten Stacks und schlechter Laufzeit.
- ▶ **Rein funktional:** Es gibt funktionale Programmiersprachen, die als »rein« oder »pur« bezeichnet werden und keine Zustandsänderungen erlauben. Die Entwicklung von Ein-/Ausgabeoperationen oder simplen Zufallszahlen ist ein großer Akt, der für normale Entwickler nicht mehr nachvollziehbar ist. Die Konzepte sind kompliziert, doch zum Glück sind die meisten funktionalen Sprachen nicht so rein und erlauben Zustandsänderungen, nur Programmierer versuchen genau diese Zustandsänderungen zu vermeiden, um sich nicht die Nachteile damit einzuhandeln.
- ▶ **Funktional mit Java:** Wenn es darum geht, nur mit Funktionen zu arbeiten, kommen Entwickler schnell an einen Punkt, an dem Funktionen andere Funktionen als Argumente übergeben oder Funktionen zurückgeben. So etwas lässt sich in Java in der traditionellen Syntax nur sehr umständlich schreiben. Dies führt dazu, dass alles so unlesbar wird, dass der ganze Vorteil der kompakten deklarativen Schreibweise verloren geht.

Aus heutiger Sicht stellt sich eine Kombination aus beiden Konzepten als zukunftsweisend dar. Mit Lambda-Ausdrücken sind funktionale Programme kompakt und relativ gut lesbar, und die JVM hat gute Optimierungsmöglichkeiten. Java ermöglicht beide Programmierparadigmen, und Entwickler können den Weg wählen, der für eine Problemlösung gerade am besten ist. Diese Mehrdeutigkeit schafft natürlich auch Probleme, denn immer wenn es mehrere Lösungswege gibt, entstehen Auseinandersetzungen um die beste der Varianten – und hier kann von Entwickler zu Entwickler eine konträre Meinung herrschen. Funktionale Programmierung hat unbestrittene Vorteile, und das wollen wir uns genau anschauen.

### 12.6.3 Funktionale Programmierung in Java am Beispiel vom Comparator

Funktionale Programmierung hat auch daher etwas Akademisches, weil in den Köpfen der Entwickler oftmals dieses Programmierparadigma nur mit mathematischen Funktionen in Verbindung gebracht wird. Und die wenigsten werden tatsächlich Fakultät oder Fibonacci-Zahlen in Programmen benötigen und daher schnell funktionale Programmierung beisei-

telegen. Doch diese Vorurteile sind unbegründet, und es ist hilfreich, funktionale Programmierung gedanklich von der Mathematik zu lösen, denn die allermeisten Programme haben nichts mit mathematischen Funktionen im eigentlichen Sinne zu tun, wohl aber viel stärker mit formal beschriebenen Methoden.

Betrachten wir erneut unser Beispiel aus der Einleitung, die Sortierung von Strings, diesmal aus der Sicht eines funktionalen Programmierers. Ein `Comparator` ist eine einfache »Funktion«, mit zwei Parametern und einer Rückgabe. Diese »Funktion« (realisiert als Methode) wiederum wird an die `sort(...)`-Methode übergeben. Alles das ist funktionale Programmierung, denn wir programmieren Funktionen und übergeben sie. Drei Beispiele (Generics ausgelassen):

Code	Bedeutung
<code>Comparator c = (c1, c2) -&gt; ...</code>	Implementiert eine Funktion über einen Lambda-Ausdruck.
<code>Arrays.sort(T[] a, Comparator c)</code>	Nimmt eine Funktion als Argument an.
<code>Collections.reverseOrder(Comparator cmp)</code>	Nimmt eine Funktion an und liefert auch eine zurück.

Tabelle 12.8 Beispiele für Funktionen in der Übergabe und als Rückgabe

Funktionen selbst können in Java nicht übergeben werden, also helfen sich Java-Entwickler mit der Möglichkeit, die Funktionalität in eine Methode zu setzen, sodass die Funktion zum Objekt mit einer Methode wird, was die Logik realisiert. Lambda-Ausdrücke bzw. Methoden/Konstruktorreferenzen geben eine kompakte Syntax ohne den Ballast, extra eine Klasse mit einer Methode schreiben zu müssen.

Der Typ `Comparator` ist eine funktionale Schnittstelle und steht für eine besondere Funktion mit zwei Parametern gleichen Typs und einer Ganzzahl-Rückgabe. Es gibt weitere funktionale Schnittstellen, die etwas flexibler sind als `Comparator`, in der Weise, dass etwa die Rückgabe statt `int` auch `double` oder etwas anderes sein kann.

#### 12.6.4 Lambda-Ausdrücke als Funktionen sehen

Wir haben gesehen, dass sich Lambda-Ausdrücke in einer Syntax formulieren lassen, die folgende allgemeine Form hat:

`( LambdaParameter ) -> { Anweisungen }`

Der Pfeil macht gut deutlich, dass wir es bei Lambda-Ausdrücken mit Funktionen zu tun haben, die etwas abbilden. Im Fall vom `Comparator` ist es eine Abbildung von zwei Strings auf eine Ganzzahl – in eine etwas mathematischere Notation gepackt: `(String, String) -> int`.



### Beispiel

Methoden gibt es mit und ohne Rückgabe und mit und ohne Parameter. Genauso ist das mit Lambda-Ausdrücken. Ein paar Beispiele in Java-Code mit ihren Abbildungen:

Lambda-Ausdruck	Abbildung
(int a, int b) → a + b	(int, int) → int
(int a) → Math.abs(a)	(int) → int
(String s) → s.isEmpty()	(String) → boolean
(Collection c) → c.size()	(Collection) → int
() → Math.random()	() → double
(String s) → { System.out.print(s); }	(String) → void
() → {}	() → void

Tabelle 12.9 Lambda-Ausdrücke und was sie als Funktionen abbilden



### Begriff: Funktion versus Methode

Die Java-Sprachdefinition kennt den Begriff »Funktion« nicht, sondern spricht nur von Methoden. Methoden hängen immer an Klassen, und das heißt, dass Methoden immer an einem Kontext hängen. Das ist zentral bei der Objektorientierung, da Methoden auf Attribute lesend und schreibend zugreifen können. Lambda-Ausdrücke wiederum realisieren Funktionen, die erst einmal ihre Arbeitswerte rein aus den Parametern beziehen, sie hängen nicht an Klassen und Objekten. Der Gedanke bei funktionalen Programmiersprachen ist der, ohne Zustände auszukommen, also Funktionen so clever anzuwenden, dass sie ein Ergebnis liefern. Funktionen geben für eine spezifische Parameterkombination immer dasselbe Ergebnis zurück, unabhängig vom Zustand des umgebenden Gesamtprogramms.

## 12.7 Funktionale Schnittstelle aus dem java.util.function-Paket

Funktionen realisieren Abbildungen, und da es verschiedene Arten von Abbildungen geben kann, bietet die Java-Standardbibliothek im Paket `java.util.function` für die häufigsten Fälle funktionale Schnittstellen an. Ein erster Überblick:

Schnittstelle	Abbildung
Consumer<T>	(T) → void
DoubleConsumer	(double) → void
BiConsumer<T,U>	(T, U) → void
Supplier<T>	() → T
BooleanSupplier	() → boolean
Predicate<T>	(T) → boolean
LongPredicate	(long) → boolean
BiPredicate<T,U>	(T, U) → boolean
Function<T,R>	(T) → R
LongToDoubleFunction	(long) → double
BiFunction<T,U,R>	(T, U) → R
UnaryOperator<T>	(T) → T
DoubleBinaryOperator	(double, double) → double

Tabelle 12.10 Beispiele einiger vordefinierter funktionaler Schnittstellen

### 12.7.1 Blöcke mit Code und die funktionale Schnittstelle Consumer

Anweisungen von Code lassen sich in eine Methode eines Objekts setzen und auf diese Weise weitergeben. Das ist eine häufige Notwendigkeit, für die das Paket `java.util.function` eine einfache funktionale Schnittstelle `Consumer` vorgibt, die einen Konsumenten repräsentiert, der Daten annimmt und dann »verbraucht« (konsumiert) und nichts zurückgibt.

```
interface java.util.function.Consumer<T>
```

- `void accept(T t)`  
Führt Operationen mit der Übergabe `t` durch.
- `default Consumer<T> andThen(Consumer<? super T> after)`  
Liefert einen neuen `Consumer`, der erst den aktuellen `Consumer` ausführt und danach `after`.

Die `accept(...)`-Methode bekommt ein Argument – wobei die Implementierung natürlich nicht zwingend darauf zurückgreifen muss – und liefert keine Rückgabe. Transformationen sind damit nicht möglich, denn nur über Umwege kann der Konsument die Ergebnisse spei-

chern, und dafür ist die Schnittstelle nicht gedacht. Consumer-Typen sind eher gedacht als Endglied einer Kette, in der zum Beispiel Dateien in eine Datei geschrieben werden, die vorher verarbeitet wurden. Diese Seiteneffekte sind beabsichtigt, da sie nach einer Kette von seiteneffektfreien Operationen stehen.

### Typ Consumer in der API

In der Java-API zeigt sich der Typ Consumer in der Regel als Argument einer Methode `forEach(Consumer)`, die Datenquellen abläuft und für jedes Element `accept(...)` aufruft. Interessant ist die Methode am Typ Iterable, denn die wichtigen Collection-Datenstrukturen wie `ArrayList` implementieren diese Schnittstelle. So lässt sich einfach über alle Daten laufen und ein Stück Code für jedes Element ausführen. Auch Iterator hat eine vergleichbare Methode, da heißt sie `forEachRemaining(Consumer)` – das »Remaining« macht deutlich, dass der Iterator schon ein paar `next()`-Aufrufe erlebt haben könnte und die Konsumenten daher nicht zwingend die ersten Elemente mitbekommen.

#### Beispiel

Gib jedes Element einer Liste auf der Konsole aus:

```
Arrays.asList( 1, 2, 3, 4 ).forEach( System.out::println );
```

[zB]

Gegenüber einem normalen Durchiterieren ist die funktionale Variante ein wenig kürzer im Code, aber sonst gibt es keinen Unterschied. Auch `forEach(...)` macht auf dem Iterable nichts anderes, als alle Elemente über den Iterator zu holen.

### Einen eigenen Wrapper-Consumer schreiben

Immer repräsentieren Konsumenten Code, und eine API kann nun einfach einen Codeblock nach der Art `doSomethingWith(myConsumer)` annehmen, um ihn etwa in einem Hintergrund-Thread abzuarbeiten oder wiederholend auszuführen, oder kann ihn nach einer erlaubten Maximaldauer abbrechen oder die Zeit messen oder, oder, oder ...

Schreiben wir einen Consumer-Wrapper, der die Ausführungszeit eines anderen Konsumenten loggt:

**Listing 12.8** src/main/java/com/tutego/insel/lambda/Consumers.java, Ausschnitt

```
class Consumers {
    public static <T> Consumer<T> measuringConsumer( Consumer<T> block ) {
        return t -> {
            long start = System.nanoTime();
            block.accept( t );
            long duration = System.nanoTime() - start;
            Logger.getAnonymousLogger().info( "Ausführungszeit (ns): " + duration );
        };
    }
}
```

```

    };
}
}
```

Folgender Aufruf zeigt die Nutzung:

```
Arrays.asList( 1, 2, 3, 4 )
    .forEach( executionTimeLogger( System.out::println ) );
```

Was wir hier implementiert haben, ist ein Beispiel für das Execute-around-Method-Muster, bei dem wir um einen Block Code noch etwas anderes legen.

### 12.7.2 Supplier

Ein *Supplier* (auch *Provider* genannt) ist eine Fabrik und sorgt für Objekte. In Java deklariert das Paket `java.util.function` die funktionale Schnittstelle *Supplier* für Objektgeber:

```
interface java.util.function.Supplier<T>
```

- `T get()`  
Führt Operationen mit der Übergabe `t` durch.

Weitere statische oder Default-Methoden deklariert *Supplier* nicht. Was `get()` nun genau liefert, ist Aufgabe der Implementierung und ein Internum. Es können neue Objekte sein, immer die gleichen Objekte (Singleton) oder Objekte aus einem Cache.

### 12.7.3 Prädikate und `java.util.function.Predicate`

Ein Prädikat ist eine Aussage über einen Gegenstand, die wahr oder falsch ist. Die Frage "tutego".`isEmpty()`, ob also die Zeichenfolge »tutego« leer ist oder nicht, wird mit falsch beantwortet – `isEmpty` ist also ein Prädikat, weil es über einen Gegenstand, ein Zeichenfolge, eine Wahrheitsaussage fällen kann.

Prädikate als Objekte sind flexibel, denn Objekte lassen sich an unterschiedliche Stellen weitergeben. Wenn etwa ein Prädikat bestimmt, was aus einer Sammlung gelöscht werden soll oder ob mindestens ein Element in einer Sammlung ist, das ein Prädikat erfüllt.

Das `java.util.function`-Paket<sup>7</sup> deklariert eine flexible funktionale Schnittstelle *Predicate* auf folgende Weise:

```
interface java.util.function.Predicate<T>
```

---

<sup>7</sup> Achtung, in `javax.sql.rowset` gibt es ebenfalls eine Schnittstelle *Predicate*.

- `boolean test(T t)`

Führt einen Test auf `t` durch und liefert `true`, wenn das Kriterium erfüllt ist, sonst `false`.

### Beispiel

[zB]

Der Test, ob ein Zeichen eine Ziffer ist, kann durch Prädikat-Objekte nun auch anders durchgeführt werden:

```
Predicate<Character> isDigit =
    c -> Character.isDigit( c ); // kurz: Character::isDigit
System.out.println( isDigit.test('a') ); // false
```

Hätte es die Schnittstelle `Predicate` schon früher in Java 1.0 gegeben, hätte es der Methode `Character.isDigit(...)` gar nicht bedurft, es hätte auch ein `Predicate<Character>` als statische Variable in der Klasse `Character` geben können, sodass ein Test dann geschrieben würde als `Character.IS_DIGIT.test(...)` oder als Rückgabe von einer Methode `Predicate<Character> isDigit()` mit der Nutzung `Character.isDigit().test(...)`. Es ist daher gut möglich, dass sich in Zukunft die API dahingehend verändert, dass Aussagen auf Gegenständen mit Wahrheitsrückgabe nicht mehr als Methoden bei den Klassen realisiert werden, sondern als Prädikat-Objekte angeboten werden. Aber Methodenreferenzen geben zum Glück die Flexibilität, dass existierende Methoden problemlos als Lambda-Ausdrücke genutzt werden können, und so kommen wir wieder von Methoden zu Funktionen.

### Typ `Predicate` in der API

Es gibt in der Java-API einige Stellen, an denen `Predicate`-Objekte genutzt werden:

- ▶ als Argument für Löschenmethoden, um in Sammlungen Elemente zu spezifizieren, die gelöscht werden sollen oder nach denen gefiltert werden soll
- ▶ bei den Default-Methoden der `Predicate`-Schnittstelle selbst, um Prädikate zu verknüpfen
- ▶ bei regulären Ausdrücken; auf einem Pattern liefern `asPredicate()` und `asMatchPredicate()` (ab Java 11) ein `Predicate` für Tests.
- ▶ in der Stream-API, bei der Objekte beim Durchlaufen des Stroms über ein Prädikat identifiziert werden, um sie etwa auszufiltern

### Beispiel

[zB]

Lösche aus einer Liste mit beliebigen Zeichen alle heraus, die Ziffern sind:

```
Predicate<Character> isDigit = Character::isDigit;
List<Character> list = new ArrayList<>( Arrays.asList( 'a', '1' ) );
list.removeIf( isDigit );
```

Auf diese Weise steht nicht die Schleife, sondern das Löschen im Vordergrund.

### Default-Methoden von Predicate

Es gibt eine Reihe von Default-Methoden, die die funktionale Schnittstelle `Predicate` anbietet. Zusammenfassend:

```
interface java.util.function.Predicate<T>
```

- `default Predicate<T> negate()`  
Liefert vom aktuellen Prädikat eine Negation. Implementiert als `return t -> ! test(t);`.
- `static <T> Predicate<T> not(Predicate<? super T> target)`  
Liefert `target.negate()`. Neu in Java 11.
- `default Predicate<T> and(Predicate<? super T> p)`
- `default Predicate<T> or(Predicate<? super T> p)`  
Verknüpft das aktuelle Prädikat mit einem anderen Prädikat mit einem logischen Und/Oder.
- `static <T> Predicate<T> isEqual(Object targetRef)`  
Liefert ein neues Prädikat, das einen Gleichheitstest mit `targetRef` vornimmt, im Grunde `return ref -> Objects.equals(ref, targetRef)`.

[zB]

#### Beispiel

Lösche aus einer Liste mit Zeichen alle die, die *keine* Ziffern sind:

```
Predicate<Character> isDigit = Character::isDigit;
Predicate<Character> isNotDigit = isDigit.negate();
List<Character> list = new ArrayList<>( Arrays.asList( 'a', '1' ) );
list.removeIf( isNotDigit );
// alternativ: list.removeIf( Predicate.not( isDigit ) );
```

### 12.7.4 Funktionen über die funktionale Schnittstelle `java.util.function.Function`

Funktionen im Sinne der funktionalen Programmierung können in verschiedenen Bauarten vorkommen: mit Parameterliste/Rückgabe oder ohne. Doch im Grunde sind es Spezialformen, und die funktionale Schnittstelle `java.util.function.Function` ist die allgemeinste, die zu einem Argument ein Ergebnis liefert.

```
interface java.util.function.Function<T,R>
```

- `R apply(T t)`  
Wendet eine Funktion an, und liefert zur Eingabe `t` eine Rückgabe.



## Beispiel

Eine Funktion zur Bestimmung des Absolutwerts:

```
Function<Double,Double> abs = a -> Math.abs( a ); // alternativ Math::abs
System.out.println( abs.apply( -12. ) );           // 12.0
```

Auch bei Funktionen ergibt sich für das API-Design ein Spannungsfeld, denn im Grunde müssen »Funktionen« nun gar nicht mehr als Methoden angeboten werden, sondern Klassen könnten sie auch als Function-Objekte anbieten. Doch da Methodenreferenzen problemlos die Brücke von Methodennamen zu Objekten schlagen, fahren Entwickler mit klassischen Methoden ganz gut.

## Typ Function in der API

Die Stream-API ist der größte Nutznießer des Function-Typs. Es finden sich einige wenige Beispiele bei Objektvergleichen (Comparator), im Paket für Nebenläufigkeiten und bei Assoziativspeichern. Im Abschnitt über die Stream-API werden wir daher viele weitere Beispiele kennenlernen.



## Beispiel

Ein Assoziativspeicher soll als Cache realisiert werden, der zu Dateinamen den Inhalt assoziiert. Ist zu dem Schlüssel (dem Dateinamen) noch kein Inhalt vorhanden, soll der Dateinhalt gelesen und in den Assoziativspeicher gelegt werden.

**Listing 12.9** src/main/java/com/tutego/insel/lambda/FileCache.java, Ausschnitt

```
class FileCache {
    private Map<String,byte[]> map = new HashMap<>();
    public byte[] getContent( String filename ) {
        return map.computeIfAbsent( filename, file -> {
            try {
                return Files.readAllBytes( Paths.get( file ) );
            } catch ( IOException e ) { throw new UncheckedIOException( e ); }
        } );
    }
}
```

Auf die Methode kommt [Kapitel 17](#), »Einführung in Datenstrukturen und Algorithmen«, noch einmal zurück, das Beispiel soll nur eine Vorstellung geben, dass Funktionen an andere Funktionen übergeben werden – hier eine Function<String,byte[]> an computeIfAbsent(...). Sobald an den Datei-Cache der Aufruf getContent(String) geht, wird dieser die Map fragen, und wenn diese zu dem Schlüssel keinen Wert hat, wird sie den Lambda-Ausdruck auswerten, um zu dem Dateinamen den Inhalt zu liefern.

### Getter-Methoden als Function über Methodenreferenzen

Methodenreferenzen gehören zu den syntaktisch knappsten Sprachmitteln von Java. In Kombination mit Gettern ist ein Muster abzulesen, das oft in Code zu sehen ist. Zunächst noch einmal zur Wiederholung von Function und der Nutzung bei Methodenreferenzen:

```
Function<String, String> func1a = (String s) -> s.toUpperCase();
Function<String, String> func1b = String::toUpperCase;
Function<Point, Double> func2a = (Point p) -> p.getX();
Function<Point, Double> func2b = Point::getX;
System.out.println( func1b.apply( "jocelyn" ) );           // JOCELYN
System.out.println( func2b.apply( new Point( 9, 0 ) ) ); // 9.0
```

Dass Function auf die gegebene Methodenreferenz passt, ist auf den ersten Blick unverständlich, da die Signaturen von `toUpperCase()` und `getX()` keinen Parameter deklarieren, also im üblichen Sinne keine Funktionen sind, wo etwas reinkommt und wieder rauskommt. Wir haben es hier aber mit einem speziellen Fall zu tun, denn die in der Methodenreferenz genannten Methoden sind a) nicht statisch – wie `Math::max` –, und b) ist auch keine Referenz – wie `System.out::print` – im Spiel, sondern hier wird der Compiler eine Objektmethode auf genau dem Objekt aufrufen, das als erstes Argument der funktionalen Schnittstelle übergeben wurde. (Diesen Satz bitte zweimal lesen.)

Damit ist Function ein praktischer Typ bei allen Szenarien, bei denen irgendwie über Getter Zustände erfragt werden, wie es etwa bei einem Comparator öfter vorkommt. Hier ist eine statische Methode – die Generics einmal ausgelassen – `Comparator.comparing(Function<...> keyExtractor)` sehr nützlich.



#### Beispiel

Besorge eine Liste von Paketen, die vom Klassenlader zugänglich sind, und sortiere sie nach Namen:

```
List<Package> list = Arrays.asList( Package.get_packages() );
Collections.sort( list, Comparator.comparing( Package::get_name() ) );
System.out.println( list ); // [package java.io, ... sun.util.locale ...]
```

### Default-Methoden in Function

Die funktionale Schnittstelle schreibt nur eine Methode `apply(...)` vor, deklariert jedoch noch drei zusätzliche Default-Methoden:

```
interface java.util.function.Function<T,R>
```

- static <T> Function<T,T> identity()
 Liefert eine Funktion, die immer die Eingabe als Ergebnis liefert, implementiert als return t -> t;.
- default <V> Function<T,V> andThen(Function<? super R,> ? extends V> after)
 Entspricht t -> after.apply(apply(t)).
- default <V> Function<V,R> compose(Function<? super V,> ? extends T> before)
 Entspricht v -> apply(before.apply(v)).

identity() scheint auf den ersten Blick sinnlos zu sein, erfüllt aber einen Zweck, wenn in der API eine Function erforderlich ist, bei der sich nichts verändern, die Eingabe also die Ausgabe sein soll.

Die Methoden andThen(...) und compose(...) unterscheiden sich darin, in welcher Reihenfolge die Funktionen aufgerufen werden. Das Gute ist, dass die Parameternamen (before, after) klarmachen, was hier in welcher Reihenfolge aufgerufen wird.

### Beispiel

[zB]

```
Function<String, String> f1 = s -> "~" + s + "~";
Function<String, String> f2 = s -> "<" + s + ">";
System.out.println( f1.andThen( f2 ).apply( ":" ) ); // <:~)
System.out.println( f2.andThen( f1 ).apply( ":" ) ); // ~:>~
System.out.println( f1.compose( f2 ).apply( ":" ) ); // ~:>~
System.out.println( f2.compose( f1 ).apply( ":" ) ); // <:~)
```

### Function versus Consumer/Predicate

Im Grunde lässt sich alles als Function darstellen, denn

- ▶ ein Consumer<T> lässt sich auch als Function<T,Void> verstehen  
(es geht etwas rein, aber nichts raus),
- ▶ ein Predicate<T> als Function<T,Boolean> und
- ▶ ein Supplier<T> als Function<Void,T>.

Dennoch erfüllen diese speziellen Typen ihren Zweck, denn je genauer der Typ, desto besser.

Function ist kein Basistyp von Consumer oder Supplier, da sich »keine Rückgabe« oder »kein Parameter« nicht durch einen generischen Typ ausdrücken kann, der bei Function<T,R> möglich wäre.

### UnaryOperator

Es gibt auch eine weitere Schnittstelle im `java.util.function`-Paket, die `Function` spezialisiert, und zwar `UnaryOperator`. Ein `UnaryOperator` ist eine spezielle `Function`, bei der die Typen für »Eingang« und »Ausgang« gleich sind.

```
interface java.util.function.UnaryOperator<T>
extends Function<T,T>
```

- `static <T> UnaryOperator<T> identity()`  
Liefert den Identitäts-Operator, der alle Eingaben auf die Ausgaben abbildet.

Die generischen Typen machen deutlich, dass der Typ des Methodenparameters gleich dem Ergebnistyp ist. Bis auf `identity()` gibt es keine weitere Funktionalität, die Schnittstelle dient lediglich zur Typdeklaration.

An einigen Stellen der Java-Bibliothek kommt dieser Typ auch vor, etwa bei der Methode `replaceAll(UnaryOperator)` der `List`-Typen.



### Beispiel

Verdopple jeden Eintrag in der Liste:

```
List<Integer> list = Arrays.asList( 1, 2, 3 );
list.replaceAll( e -> e * 2 );
System.out.println( list ); // [2, 4, 6]
```

### 12.7.5 Ein bisschen Bi ...

Bi ist eine bekannte lateinische Vorsilbe für »zwei«, was übertragen auf die Typen aus dem Paket `java.util.function` bedeutet, dass statt eines Arguments zwei übergeben werden können.

Typ	Schnittstelle	Operation
Konsument	<code>Consumer&lt;T&gt;</code>	<code>void accept(T t)</code>
	<code>BiConsumer&lt;T,U&gt;</code>	<code>void accept(T t, U u)</code>
Funktion	<code>Function&lt;T,R&gt;</code>	<code>R apply(T t)</code>
	<code>BiFunction&lt;T,U,R&gt;</code>	<code>R apply(T t, U u)</code>

Tabelle 12.11 Ein-/Zwei-Argument-Methoden im Vergleich

Typ	Schnittstelle	Operation
Prädikat	Predicate<T>	boolean test(T t)
	BiPredicate<T,U>	boolean test(T t, U u)

Tabelle 12.11 Ein-/Zwei-Argument-Methoden im Vergleich (Forts.)

Die Bi-Typen haben mit den Nicht-Bi-Typen keine Typbeziehung.

### BiConsumer

Der BiConsumer deklariert die Methode accept(T, U) mit zwei Parametern, die jeweils unterschiedliche Typen tragen können. Haupteinsatzpunkt des Typs in der Java-Standardbibliothek sind Assoziativspeicher, die Schlüssel und Werte an accept(...) übergeben. So deklariert Map die Methode:

```
interface java.util.Map<K,V>
```

- default void forEach(BiConsumer<? super K, ? super V> action)  
Läuft den Assoziativspeicher ab und ruft auf jedem Schlüssel-Wert-Paar die accept(...) -Methode vom übergebenen BiConsumer auf.

### Beispiel

Gib die Temperaturen der Städte aus:

```
Map<String, Integer> map = new HashMap<>();
map.put( "Manila", 25 );
map.put( "Leipzig", -5 );
map.forEach( (k, v) -> System.out.printf("%s hat %d Grad%n", k, v) );
```

[zB]

Ein BiConsumer besitzt eine Default-Methode andThen(...), wie auch der Consumer sie zur Verkettung deklariert.

```
interface java.util.function.BiConsumer<T,U>
```

- default BiConsumer<T,U> andThen(BiConsumer<? super T, ? super U> after)  
Verknüpft den aktuellen BiConsumer mit after zu einem neuen BiConsumer.

### BiFunction und BinaryOperator

Eine BiFunction ist eine Funktion mit zwei Argumenten, während eine normale Function nur ein Argument annimmt.



### Beispiel

Nutzung von Function und BiFunction mit Methodenreferenzen:

```
Function<Double,Double> sign = Math::abs;
BiFunction<Double,Double,Double> max = Math::max;
```

Die Java-Bibliothek greift viel öfter auf Function zurück als auf BiFunction. Der häufigste Einsatz findet sich in der Standardbibliothek rund um Assoziativspeicher, bei denen Schlüssel und Wert an eine BiFunction übergeben werden.



### Beispiel

Konvertiere alle assoziierten Werte einer HashMap in Großbuchstaben:

```
Map<Integer,String> map = new HashMap<>();
map.put( 1, "eins" ); map.put( 2, "zwei" );
System.out.println( map ); // {1=eins, 2=zwei}
BiFunction<Integer,String,String> func = (k, v) -> v.toUpperCase();
map.replaceAll( func );
System.out.println( map ); // {1=EINS, 2=ZWEI}
```

Ist bei einer Function der Typ derselbe, bietet die Java-API dafür den spezielleren Typ UnaryOperator. Sind bei einer BiFunction alle drei Typen gleich, bietet sich BinaryOperator an – zum Vergleich:

- ▶ interface UnaryOperator<T> extends Function<T,T>
- ▶ interface BinaryOperator<T> extends BiFunction<T,T,T>



### Beispiel

BiFunction und BinaryOperator:

```
BiFunction<Double,Double,Double> max1 = Math::max;
BinaryOperator<Double> max2 = Math::max;
```

BinaryOperator spielt bei so genannten Reduktionen eine große Rolle, wenn zum Beispiel aus zwei Werten einer wird, wie bei Math.max(...); auch das beleuchtet der Abschnitt über die Stream-API später genauer.

Die Schnittstelle BiFunction deklariert genau eine Default-Methode:

```
interface java.util.function.BiFunction<T,U,R>
extends Function<T,T>
```

- default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)

BinaryOperator dagegen wartet mit zwei statischen Methoden auf:

```
public interface java.util.function.BinaryOperator<T>
extends BiFunction<T,T,T>
```

- static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
  - static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
- Liefert einen BinaryOperator, der das Maximum/Minimum bezüglich eines gegebenen comparator liefert.

### BiPredicate

Ein BiPredicate testet zwei Argumente und verdichtet sie zu einem Wahrheitswert. Wie Predicate deklariert auch BiPredicate drei Default-Methoden and(...), or(...) und negate(...), wobei natürlich eine statische isEqual(...)-Methode wie bei Predicate in BiPredicate fehlt. Für BiPredicate gibt es in der Java-Standardbibliothek nur eine Verwendung bei einer Methode zum Finden von Dateien – der Gebrauch ist selten, zudem ja auch ein Prädikat immer eine Funktion mit boolean-Rückgabe ist, sodass es eigentlich für diese Schnittstelle keine zwingende Notwendigkeit gibt.

### Beispiel

Bei BiXXX und zwei Argumenten hört im Übrigen die Spezialisierung auf, es gibt keine Typen TriXXX, QuaddXXX ... Das ist in der Praxis auch nicht nötig, denn zum einen kann oftmals eine Reduktion stattfinden – so ist etwa `max(1, 2, 3)` gleich `max(1, max(2, 3))` –, und zum anderen kann auch der Parametertyp eine Sammlung sein, wie in `Function<List<Integer>, Integer> max`.

[zB]

### 12.7.6 Funktionale Schnittstellen mit Primitiven

Die bisher vorgestellten funktionalen Schnittstellen sind durch die generischen Typparameter sehr flexibel, aber was fehlt, sind Signaturen mit Primitiven – Java hat das »Problem«, dass Generics nur mit Referenztypen funktionieren, nicht aber mit primitiven Typen. Aus diesem Grund gibt es von fast allen Schnittstellen aus dem java.util.function-Paket vier Versionen: eine generische für beliebige Referenzen sowie Versionen für die Typen int, long und double. Die API-Designer wollten gerne die Wrapper-Typen außen vor lassen und gewisse primitive Typen unterstützen, auch aus Performance-Gründen, um nicht immer ein Boxing durchführen zu müssen.

Tabelle 12.12 gibt einen Überblick über die funktionalen Schnittstellen, die alle keinerlei Vererbungsbeziehungen zu anderen Schnittstellen haben.

Funktionale Schnittstelle	Funktionsdeskriptor
<b>XXXSupplier</b>	
BooleanSupplier	boolean getAsBoolean()
IntSupplier	int getAsInt()
LongSupplier	long getAsLong()
DoubleSupplier	double getAsDouble()
<b>XXXConsumer</b>	
IntConsumer	void accept(int value)
LongConsumer	void accept(long value)
DoubleConsumer	void accept(double value)
ObjIntConsumer<T>	void accept(T t, int value)
ObjLongConsumer<T>	void accept(T t, long value)
ObjDoubleConsumer<T>	void accept(T t, double value)
<b>XXXPredicate</b>	
IntPredicate	boolean test(int value)
LongPredicate	boolean test(long value)
DoublePredicate	boolean test(double value)
<b>XXXFunction</b>	
DoubleToIntFunction	int applyAsInt(double value)
IntToDoubleFunction	double applyAsDouble(int value)
LongToIntFunction	int applyAsInt(long value)
IntToLongFunction	long applyAsLong(int value)
DoubleToLongFunction	long applyAsLong(double value)
LongToDoubleFunction	double applyAsDouble(long value)
IntFunction<R>	R apply(int value)
LongFunction<R>	R apply(long value)

Tabelle 12.12 Spezielle funktionale Schnittstellen für primitive Werte

Funktionale Schnittstelle	Funktionsdeskriptor
DoubleFunction<R>	R apply(double value)
ToIntFunction<T>	int applyAsInt(T t)
ToLongFunction<T>	long applyAsLong(T t)
ToDoubleFunction<T>	double applyAsDouble(T t)
ToIntBiFunction<T,U>	int applyAsInt(T t, U u)
ToLongBiFunction<T,U>	long applyAsLong(T t, U u)
ToDoubleBiFunction<T,U>	double applyAsDouble(T t, U u)
<b>XXXOperator</b>	
IntUnaryOperator	int applyAsInt(int operand)
LongUnaryOperator	long applyAsLong(long operand)
DoubleUnaryOperator	double applyAsDouble(double operand)
IntBinaryOperator	int applyAsInt(int left, int right)
LongBinaryOperator	long applyAsLong(long left, long right)
DoubleBinaryOperator	double applyAsDouble(double left, double right)

Tabelle 12.12 Spezielle funktionale Schnittstellen für primitive Werte (Forts.)

### Statische und Default-Methoden

Einige generisch deklarierte funktionale Schnittstellen-Typen besitzen Default-Methoden bzw. statische Methoden, und Ähnliches findet sich auch bei den primitiven funktionalen Schnittstellen wieder:

- ▶ Die XXXConsumer-Schnittstellen deklarieren default XXXConsumer andThen(XXXConsumer after), aber nicht die ObjXXXConsumer-Typen, sie besitzen keine Default-Methode.
- ▶ Die XXXPredicate-Schnittstellen deklarieren:
  - default XXXPredicate negate()
  - default XXXPredicate and(XXXPredicate other)
  - default XXXPredicate or(XXXPredicate other)
- ▶ Jeder XXXUnaryOperator besitzt:
  - default XXXUnaryOperator andThen(XXXUnaryOperator after)
  - default XXXUnaryOperator compose(XXXUnaryOperator before)
  - static XXXUnaryOperator identity()

- ▶ BinaryOperator hat zwei statische Methoden `maxBy(...)` und `minBy(...)`, die es nicht in der primitiven Version `XXXBinaryOperator` gibt, da kein Comparator bei primitiven Vergleichen nötig ist.
- ▶ Die `XXXSupplier`-Schnittstellen deklarieren keine statischen Methoden oder Default-Methoden, genauso wie es auch Supplier nicht tut.

## 12.8 Optional ist keine Nullnummer

Java hat eine besondere Referenz, die Entwicklern die Haare zu Berge stehen lässt und die Grund für lange Debug-Stunden ist: die `null`-Referenz. Eigentlich sagt `null` nur aus: »nicht initialisiert«. Doch was `null` so problematisch macht, ist die `NullPointerException`, die durch referenzierte `null`-Ausdrücke ausgelöst wird.



### Beispiel

Entwickler haben vergessen, das Attribut `location` mit einem Objekt zu initialisieren, sodass `setLocation(...)` fehlschlagen wird:

```
class Place {
    private Point2D location;
    public void setLocation( double longitude, double latitude ) {
        location.setLocation( longitude, latitude ); // ☠ NullPointerException
    }
}
```

### Einsatz von `null`

Fehler dieser Art sind durch Tests relativ leicht aufzuspüren. Aber hier liegt nicht das Problem. Das eigentliche Problem ist, dass Entwickler allzu gerne die typenlose `null`<sup>8</sup> als magischen Sonderwert sehen, sodass sie neben »nicht initialisiert« noch etwas anderes bedeutet:

- ▶ Erlaubt die API in Argumenten für Methoden/Konstruktoren `null`, heißt das meistens »nutze einen Default-Wert« oder »nichts gegeben, ignorieren«.
- ▶ In Rückgaben von Methoden steht `null` oftmals für »nichts gemacht« oder »keine Rückgabe«. Im Gegensatz dazu kodieren andere Methoden wiederum mit der Rückgabe `null`, dass eine Operation erfolgreich durchlaufen wurde, und würden sonst zum Beispiel Fehlerobjekte zurückgeben.<sup>9</sup>

<sup>8</sup> `null instanceof Typ` ist immer `false`.

<sup>9</sup> Zum Glück wird `null` selten als Fehler-Identifikator genutzt, die Zeiten sind vorbei. Hier sind Ausnahmen die bessere Wahl, denn Fehler sind Ausnahmen im Programm.



### Beispiel 1

Die mit Javadoc dokumentierte Methode `getTask(out, fileManager, diagnosticListener, options, classes, compilationUnits)` in der Schnittstelle `JavaCompiler` ist so ein Beispiel:

- ▶ `out`: »a writer for additional output from the compiler; use `System.err` if `null`«
- ▶ `fileManager`: »a file manager; if `null` use the compiler's standard filemanager«
- ▶ `diagnosticListener`: »a diagnostic listener; if `null` use the compiler's default method for reporting diagnostics«
- ▶ `options`: »compiler options, `null` means no options«
- ▶ `classes`: »names of classes to be processed by annotation processing, `null` means no class names«
- ▶ `compilationUnits`: »the compilation units to compile, `null` means no compilation units«

Alle Argumente können `null` sein, `getTask(null, null, null, null, null, null)` ist ein korrekter Aufruf. Schön ist die API nicht, und für so lange Parameterlisten gibt es mit dem Builder-Pattern eine feine Alternative. Frei erfunden könnte das so aussehen: `new CompilationTask.Builder().out(...).fileManager(...).build()`.



### Beispiel 2

Der `BufferedReader` erlaubt das zeilenweise Einlesen aus Datenquellen, und `readLine()` liefert `null`, wenn es keine Zeile mehr zu lesen gibt.



### Beispiel 3

Viel Irritation gibt es mit der API vom Assoziativspeicher. Eine gewöhnliche `HashMap` kann als assoziierten Wert `null` bekommen, doch `get(key)` liefert auch dann `null`, wenn es keinen assoziierten Wert gibt. Das führt zu einer Mehrdeutigkeit, da die Rückgabe von `get(...)` nicht verrät, ob es eine Abbildung auf `null` gibt oder ob der Schlüssel nicht vorhanden ist.

```
Map<Integer, String> map = new HashMap<>();
map.put(0, null);
System.out.println( map.containsKey(0) );      // true
System.out.println( map.containsValue(null) ); // true
System.out.println( map.get(0) );                // null
System.out.println( map.get(1) );                // null
```

Kann die Map `null`-Werte enthalten, muss es immer ein Paar der Art `if(map.containsKey(key))`, gefolgt von `map.get(key)`, geben. Am besten verzichten Entwickler auf `null` in Datenstrukturen.

Da `null` so viele Einsatzfälle hat und das Lesen der API-Dokumentation gerne übersprungen wird, sollte es zu einigen `null`-Einsätzen Alternativen geben. Manches Mal ist das einfach, etwa wenn die Rückgabe Sammlungen sind. Dann gibt es mit einer leeren Sammlung eine gute Alternative zu `null`. Das ist ein Spezialfall des so genannten Null-Object-Patterns und wird in [Kapitel 17, »Einführung in Datenstrukturen und Algorithmen«](#), näher beschrieben.

Fehler, die aufgrund einer `NullPointerException` entstehen, ließen sich natürlich komplett vermeiden, wenn immer ordentlich auf `null`-Referenzen getestet würde. Aber gerade die `null`-Prüfungen werden von Entwicklern gerne vergessen, da ihnen nicht bewusst ist oder sie nicht erwarten, dass eine Rückgabe `null` sein kann. Gewünscht ist ein Programmkonstrukt, bei dem explizit wird, dass ein Wert nicht vorhanden sein kann, sodass nicht `null` diese Rolle übernehmen muss. Wenn im Code lesbar ist, dass ein Wert optional ist, also vorhanden sein kann oder nicht, reduziert das Fehler.

### Geschichte

Tony Hoare gilt als »Erfinder« der `null`-Referenz. Heute bereut er es und nennt die Entscheidung »my billion-dollar mistake«.<sup>10</sup>

#### 12.8.1 Optional-Typ

Die Java-Bibliothek bietet eine Art Container, der ein Element enthalten kann oder nicht. Wenn der Container ein Element enthält, ist es nie `null`. Dieser Container kann befragt werden, ob er ein Element enthält oder nicht. Eine `null` als Kennung ist somit überflüssig.

### zB Beispiel

```
Optional<String> opt1 = Optional.of( "Aitzaz Hassan Bangash" );
System.out.println( opt1.isPresent() ); // true
System.out.println( opt1.isEmpty() ); // false
System.out.println( opt1.get() ); // Aitzaz Hassan Bangash
Optional<String> opt2 = Optional.empty();
System.out.println( opt2.isPresent() ); // false
System.out.println( opt2.isEmpty() ); // true
```

<sup>10</sup> Er sagt dazu: »*It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*« Unter <http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare> gibt es ein Video mit ihm und Erklärungen.

```
// opt2.get() -> java.util.NoSuchElementException: No value present
Optional<String> opt3 = Optional.ofNullable( "Malala" );
System.out.println( opt3.isPresent() ); // true
System.out.println( opt3.isEmpty() ); // false
System.out.println( opt3.get() ); // Malala
Optional<String> opt4 = Optional.ofNullable( null );
System.out.println( opt4.isPresent() ); // false
// opt4.get() -> java.util.NoSuchElementException: No value present
```

### final class java.util.Optional<T>

- static <T> Optional<T> empty()
 Liefert ein leeres Optional-Objekt.
- boolean isPresent()
 Liefert wahr, wenn dieses Optional einen Wert hat, sonst ist wie im Fall von empty() die Rückgabe false.
- boolean isEmpty()
 Gegenteil von isPresent(). Neu in Java 11.
- static <T> Optional<T> of(T value)
 Baut ein neues Optional mit einem Wert auf, der nicht null sein darf; andernfalls gibt es eine NullPointerException. null in das Optional hineinzubekommen, geht also nicht.
- static <T> Optional<T> ofNullable(T value)
 Liefert ein Optional mit dem Wert, wenn dieser ungleich null ist, bei null ist die Rückgabe ein Optional.empty().
- T get()
 Liefert den Wert. Enthält das Optional keinen Wert, weil es isEmpty() ist, folgt eine NoSuchElementException.
- TorElse(T other)
 Ist ein Wert isPresent(), liefere den Wert; ist es isEmpty(), liefere other.
- Stream<T> stream()
 Konvertiere das Optional in den Datentyp Stream.

Des Weiteren überschreibt Optional die Methoden equals(...), toString() und hashCode() – 0, wenn kein Wert gegeben ist, sonst Hashcode vom Element – und ein paar weitere Methoden, die wir uns später anschauen.



### Hinweis

Intern `null` zu verwenden hat zum Beispiel den Vorteil, dass die Objekte serialisiert werden können. `Optional` implementiert `Serializable` *nicht*, daher sind `Optional`-Attribute nicht serialisierbar, können also etwa nicht im Fall von Remote-Aufrufen mit RMI übertragen werden. Auch die Abbildung auf XML oder auf Datenbanken ist umständlicher, wenn nicht JavaBean-Properties herangezogen werden, sondern die internen Attribute.

### Ehepartner oder nicht?

`Optional` wird also dazu verwendet, im Code explizit auszudrücken, ob ein Wert vorhanden ist oder nicht. Das gilt auf beiden Seiten: Der Erzeuger muss explizit `ofXXX(...)` aufrufen und der Nutzer explizit `isPresent()`, `isEmpty()` oder `get()`. Beide Seiten sind sich bewusst, dass sie es mit einem Wert zu tun haben, der optional ist, also existieren kann oder nicht. Wir wollen das in einem Beispiel nutzen, und zwar für eine Person, die einen Ehepartner haben kann:

**Listing 12.10** src/main/java/com/tutego/insel/lang/Person.java, Ausschnitt

```
public class Person {
    private Person spouse;
    public void setSpouse( Person spouse ) {
        this.spouse = Objects.requireNonNull( spouse );
    }
    public void removeSpouse() {
        spouse = null;
    }
    public Optional<Person> getSpouse() {
        return Optional.ofNullable( spouse );
    }
}
```

In diesem Beispiel ist `null` für die interne Referenz auf den Partner möglich; diese Kodierung soll aber nicht nach außen gelangen. Daher liefert `getSpouse()` nicht direkt die Referenz, sondern es kommt `Optional` zum Einsatz und drückt aus, ob eine Person einen Ehepartner hat oder nicht. Auch bei `setSpouse(...)` akzeptieren wir kein `null`, denn `null`-Argumente sollten so weit wie möglich vermieden werden. Ein `Optional` ist hier nicht angemessen, weil es ein Fehler ist, `null` zu übergeben. Zusätzlich sollte natürlich die Javadoc an `setSpouse(...)` dokumentieren, dass ein `null`-Argument zu einer `NullPointerException` führt. Daher passt `Optional` als ParameterTyp nicht.

**Listing 12.11** src/main/java/com/tutego/insel/lang/OptionalDemo.java, main()

```
Person heinz = new Person();
System.out.println( heinz.getSpouse().isEmpty() ); // true
```

```

Person eva = new Person();
heinz.setSpouse( eva );
System.out.println( heinz.getSpouse().isPresent() ); // true
System.out.println( heinz.getSpouse().get() ); // com/.../Person
heinz.removeSpouse();
System.out.println( heinz.getSpouse().isEmpty() ); // true

```

### 12.8.2 Primitive optionale Typen

Während Referenzen `null` sein können und auf diese Weise das Nichtvorhandensein anzeigen, ist das bei primitiven Datentypen nicht so einfach. Wenn eine Methode ein `boolean` zurückgibt, bleibt neben `true` und `false` nicht viel übrig, und ein »nicht zugewiesen« wird dann doch gerne wieder über einen `Boolean` verpackt und auf `null` getestet. Gerade bei Ganzzahlen gibt es immer wieder Rückgaben wie `-1`.<sup>11</sup> Das ist bei den folgenden Beispielen der Fall:

- ▶ Wenn bei `InputStreams.read(...)` keine Eingaben mehr kommen, wird »`-1`« zurückgegeben.
- ▶ `indexOf(Object)` von `List` liefert »`-1`«, wenn das gesuchte Objekt nicht in der Liste ist und folglich auch keine Position vorhanden ist.
- ▶ Bei einer unbekannten Bytelänge einer MIDI-Datei (Typ `MidiFileFormat`) hat `getByteLength()` als Rückgabe »`-1`«.

Diese magischen Werte sollten vermieden werden, und daher kann auch der optionale Typ wieder erscheinen.

Als generischer Typ kann `Optional` beliebige Typen kapseln, und primitive Werte könnten in Wrapper verpackt werden. Allerdings bietet Java für drei primitive Typen spezielle `Optional`-Typen an: `OptionalInt`, `OptionalLong`, `OptionalDouble`:

<code>Optional&lt;T&gt;</code>	<code>OptionalInt</code>	<code>OptionalLong</code>	<code>OptionalDouble</code>
<code>static &lt;T&gt; Optional&lt;T&gt; empty()</code>	<code>static OptionalInt empty()</code>	<code>static OptionalLong empty()</code>	<code>static OptionalDouble empty()</code>
<code>T get()</code>	<code>int getAsInt()</code>	<code>long getAsLong()</code>	<code>double getAsDouble()</code>
<code>boolean isPresent() boolean isEmpty()</code>			

Tabelle 12.13 Methodenvergleich zwischen den vier `OptionalXXX`-Klassen

<sup>11</sup> Unter <https://docs.oracle.com/en/java/javase/11/docs/api/constant-values.html> lassen sich alle Konstantendeklarationen einsehen.

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
static <T> Optional<T> of(T value)	static OptionalInt of(int value)	static OptionalLong of(long value)	static OptionalDouble of(double value)
static <T> Optional<T> ofNullable(T value)	nicht übertragbar		
T orElse(T other)	int orElse(int other)	long orElse(long other)	double orElse( double other)
boolean equals(Object obj)			
int hashCode()			
String toString()			

Tabelle 12.13 Methodenvergleich zwischen den vier OptionalXXX-Klassen (Forts.)

Die Optional-Methode ofNullable(...) fällt in den primitiven Optional-Klassen natürlich raus. Die optionalen Typen für die drei primitiven Typen haben insgesamt weniger Methoden, und die Tabelle oben ist nicht ganz vollständig. Wir kommen im Rahmen der funktionalen Programmierung in Java noch auf die verbleibenden Methoden wie isPresent(...) zurück.

#### Best Practice

OptionalXXX-Typen eignen sich hervorragend als Rückgabetyp, sind als Parametertyp denkbar, doch wenig attraktiv für interne Attribute. Intern ist null eine akzeptable Wahl, der »Typ« ist schnell und speicherschonend.

### 12.8.3 Erstmal funktional mit Optional

Neben den vorgestellten Methoden wie ofXXX(...) und isPresent(), isEmpty() gibt es weitere, die auf funktionale Schnittstellen zurückgreifen:

```
final class java.lang.Optional<T>
```

- void ifPresent(Consumer<? super T> consumer)  
Repräsentiert das Optional einen Wert, rufe den Consumer mit diesem Wert auf, andernfalls mache nichts.

- `void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`  
Repräsentiert das Optional einen Wert, rufe den Consumer mit diesem Wert auf, andernfalls führe emptyAction aus. Das Runnable muss hier als Typ aus java.lang herhalten, weil es im java.util.function-Paket keine Schnittstelle gibt, die keine Parameter hat und auch keine Rückgabe liefert.
- `Optional<T> filter(Predicate<? super T> predicate)`  
Enthält das Optional einen Wert und ist das Prädikat predicate auf dem Wert wahr, ist die Rückgabe das eigene Optional (also this), sonst ist die Rückgabe Optional.empty().
- `<U> Optional<U> map(Function<? super T, ? extends U> mapper)`  
Repräsentiert das Optional einen Wert, dann wende die Funktion an und verpacke das Ergebnis (wenn es ungleich null ist) wieder in ein Optional. Ist das Optional ohne Wert, dann ist die Rückgabe Optional.empty(), genauso, wenn die Funktion null liefert.
- `<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)`  
Wie map(...), nur dass die Funktion ein Optional statt eines direkten Werts gibt. Liefert die Funktion mapper ein leeres Optional, so ist das Ergebnis von flatMap(...) auch Optional.empty().
- `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)`  
Repräsentiert das Optional einen Wert, so liefere ihn. Ist das Optional leer, beziehe den Wert aus dem anderen Optional.
- `TorElseGet(Supplier<? extends T> other)`  
Repräsentiert das Optional einen Wert, so liefere ihn; ist das Optional leer, so beziehe den Alternativwert aus dem Supplier.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`  
Repräsentiert das Optional einen Wert, so liefere ihn, andernfalls hole mit Supplier das Ausnahmeobjekt, und löse es aus.
- `T orElseThrow()`  
Wie get(). Seit Java 10.

### Beispiel

Wenn das Optional keinen Wert hat, soll eine NullPointerException statt der NoSuchElementException ausgelöst werden.

```
String s = optionalString.orElseThrow( NullPointerException::new );
```



### Beispiel für NullPointerException-sichere Kaskadierung von Aufrufen mit Optional

Die beiden XXXmap(...)-Methoden sind besonders interessant und ermöglichen einen ganz neuen Programmierstil. Warum, soll ein Beispiel zeigen.

Der folgende Zweizeiler gibt auf meinem System »MICROSOFT KERNELDEBUGGER-NETZWERKADAPTER« aus:

```
String s = NetworkInterface.getByIndex( 2 ).getDisplayName().toUpperCase();
System.out.println( s );
```

Allerdings ist der Programmcode alles andere als gut, denn `NetworkInterface.getByIndex(int)` kann `null` zurückgeben und `getDisplayName()` auch. Um ohne eine `NullPointerException` um die Klippen zu schiffen, müssen wir schreiben:

```
NetworkInterface networkInterface = NetworkInterface.getByIndex( 2 );
if ( networkInterface != null ) {
    String displayName = networkInterface.getDisplayName();
    if ( displayName != null )
        System.out.println( displayName.toUpperCase() );
}
```

Von der Eleganz des Zweizeilers ist nicht mehr viel geblieben. Integrieren wir `Optional` (das ja eigentlich ein toller Rückgabetyp für `getByIndex()` und `getDisplayName()` wäre):

```
Optional<NetworkInterface> networkInterface =
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) );
if ( networkInterface.isPresent() ) {
    Optional<String> name =
Optional.ofNullable( networkInterface.get().getDisplayName() );
    if ( name.isPresent() )
        System.out.println( name.get().toUpperCase() );
}
```

Mit `Optional` wird es nicht sofort besser, doch statt `if` können wir einen Lambda-Ausdruck nehmen und bei `ifPresent(...)` einsetzen:

```
Optional<NetworkInterface> networkInterface =
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) );
networkInterface.ifPresent( ni -> {
    Optional<String> displayName = Optional.ofNullable( ni.getDisplayName() );
    displayName.ifPresent( name -> {
        System.out.println( name.toUpperCase() );
    } );
});
```

Wenn wir die lokalen Variablen `networkInterface` und `displayName` entfernen, landen wir bei:

```
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) ).ifPresent( ni -> {
    Optional.ofNullable( ni.getDisplayName() ).ifPresent( name -> {
        System.out.println( name.toUpperCase() );
    } );
} );
```

Von der Struktur her ist das mit der `if`-Abfrage identisch und über die Einrückungen auch zu erkennen. Fallunterscheidungen mit `Optional` und `ifPresent(...)` umzuschreiben bringt keinen Vorteil.

In Fallunterscheidungen zu denken hilft hier nicht weiter. Was wir uns bei `NetworkInterface.getByIndex( 2 ).getDisplayName().toUpperCase()` vor Augen halten müssen, ist eine Kette von Abbildungen. `NetworkInterface.getByIndex(int)` bildet auf `NetworkInterface` ab, `getDisplayName()` von `NetworkInterface` bildet auf `String` ab, und `toUpperCase()` bildet von einem `String` auf einen anderen `String` ab. Wir verketten drei Abbildungen und müssten ausdrücken können: Wenn eine Abbildung fehlschlägt, dann höre mit der Abbildung auf. Und genau hier kommen `Optional` und `map(...)` ins Spiel. In Code:

```
Optional<String> s = Optional.ofNullable( NetworkInterface.getByIndex( 2 ) )
    .map( ni -> ni.getDisplayName() )
    .map( name -> name.toUpperCase() );
s.ifPresent( System.out::println );
```

Die Klasse `Optional` hilft uns bei zwei Dingen: Erstens wird `map(...)` beim Empfangen einer null-Referenz auf ein `Optional.empty()` abbilden, und zweitens ist das Verketten von leeren Optionals kein Problem, es passiert einfach nichts – `Optional.empty().map(...)` führt nichts aus, und die Rückgabe ist einfach nur ein leeres `Optional`. Am Ende der Kette steht nicht mehr `String` (wie am Anfang des Beispiels), sondern `Optional<String>`.

Umgeschrieben mit Methodenreferenzen und weiter verkürzt ist der Code sehr gut lesbar und sicher vor einer `NullPointerException`:

```
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) )
    .map( NetworkInterface::getDisplayName )
    .map( String::toUpperCase )
    .ifPresent( System.out::println );
```

Die Logik kommt ohne externe Fallunterscheidungen aus und arbeitet nur mit optionalen Abbildungen. Das ist ein schönes Beispiel für funktionale Programmierung.

### Primitiv-Optionales mit speziellen OptionalXXX-Klassen

Die eigentliche Optional-Klasse ist generisch und kapselt jeden Referenztyp. Auch für die primitiven Typen int, long und double gibt es in drei speziellen Klassen OptionalInt, OptionalLong, OptionalDouble Methoden zur funktionalen Programmierung. Stellen wir die Methoden der vier OptionalXXX-Klassen gegenüber:

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
static <T> Optional<T> empty()	static OptionalInt empty()	static OptionalLong empty()	static OptionalDouble empty()
T get()	int getAsInt()	long getAsLong()	double getAsDouble()
<code>boolean isPresent() boolean isEmpty()</code>			
static <T> Optional<T> of(T value)	static OptionalInt of(int value)	static OptionalLong of(long value)	static Optional Double of(double value)
static <T> Optional<T> ofNullable(T value)	nicht übertragbar		
T orElse(T other)	int orElse(int other)	long orElse(long other)	double orElse (double other)
Stream<T> stream()	IntStream stream()	LongStream stream()	DoubleStream stream()
<code>boolean equals(Object obj)</code>			
<code>int hashCode()</code>			
<code>String toString()</code>			
void ifPresent( Consumer<? super T> consumer)	void ifPresent( IntConsumer consumer)	void ifPresent( LongConsumer consumer)	void ifPresent( DoubleConsumer consumer)
void ifPresent- OrElse(Consumer<? super T> action, Runnable empty- Action)	void ifPresent- OrElse( IntConsu- mer action, Runn- able emptyAction)	void ifPresent- OrElse( LongConsu- mer action, Runn- able emptyAction)	void ifPresent- OrElse( Double- Consumer action, Runnable empty- Action)

Tabelle 12.14 Vergleich von Optional mit den primitiven OptionalXXX-Klassen

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
T orElseGet( Supplier<? extends T> other)	int orElseGet( IntSupplier other)	long orElseGet( LongSupplier other)	double orElseGet( DoubleSupplier other)
<X extends Throwable> T orElseThrow( Supplier<? extends X> exceptionSupplier)	<X extends Throwable> int orElseThrow( Supplier<? extends X> exceptionSupplier)	<X extends Throwable> long orElseThrow( Supplier<? extends X> exceptionSupplier)	<X extends Throwable> double orElseThrow( Supplier<? extends X> exceptionSupplier)
Optional<T> filter(Predicate<? super T> predicate)	nicht vorhanden		
<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)			
<U> Optional<U> map(Function<? super T, ? extends U> mapper)			

Tabelle 12.14 Vergleich von Optional mit den primitiven OptionalXXX-Klassen (Forts.)

## 12.9 Was ist jetzt so funktional?

Bisher hat dieser Abschnitt einen Großteil darauf verwendet, die Typen aus dem `java.util.function`-Paket vorzustellen, also die funktionalen Schnittstellen, mit denen Entwickler Abbildungen in Java ausdrücken können. Wenig war von funktionaler Programmierung und den Vorteilen die Rede, das holen wir jetzt nach.

### Wiederverwertbarkeit

Zunächst einmal bieten Funktionen eine zusätzliche Ebene der Wiederverwertbarkeit von Code. Nehmen wir ein Prädikat wie

```
Predicate<Path> exists = path -> Files.exists( path );
```

Dieses `exists`-Prädikat ist relativ einfach und lässt auch noch die Ausnahmebehandlung aus, könnte aber natürlich komplexer sein. Der Punkt ist, dass diese Prädikate an allen möglichen Stellen wiederverwendet werden können, etwa zum Filtern in Listen oder zum Löschen von

Elementen aus Listen. Das Prädikat kann als Funktion weitergereicht oder zu neuen Prädikaten verbunden werden, etwa zu:

```
Predicate<Path> exists    = path -> Files.exists( path );
Predicate<Path> directory = path -> Files.isDirectory( path );
Predicate<Path> existsAndDirectory = exists.and( directory );
```

Methoden wie `ifPresent(Predicate)` oder `removeIf(Predicate)` nehmen dann dieses Prädikat und führen Operationen durch. Diese kleinen Mini-Objekte lassen sich sehr gut testen, und das minimiert insgesamt Fehler im Code.

Während aktuelle Bibliotheken wenig davon Gebrauch machen, Typen wie `Supplier`, `Consumer`, `Function`, `Predicate` anzunehmen und zurückzugeben, wird sich dieses im Laufe der nächsten Jahre ändern.

### Zustandslos, immutable

Bei der funktionalen Programmierung geht es darum, ohne externe Zustände auszukommen.

#### Definition

Funktionen heißen pur, wenn sie ohne einen Zustand auskommen und keine Seiteneffekte haben. `Math.max(3, 4)` ist eine pure Funktion, `System.out.println()` oder `Math.random()` sind es nicht. Einen aus puren Funktionen aufgebauten Ausdruck nennen wir *puren Ausdruck*. Er hat eine Eigenschaft, die sich in der Informatik *referenzielle Transparenz* nennt, dass nämlich das Ergebnis eines Ausdrucks an Stelle des Ausdrucks selbst gesetzt werden kann, ohne dass das Programm ein anderes Verhalten zeigt. Statt `Math.max(3, 4)` kann jederzeit 4 gesetzt werden, das Ergebnis wäre das gleiche. Ein Compiler kann bei referenzieller Transparenz diverse Optimierungen durchführen. Die API-Dokumentation sollte so gut sein, dass sie Seiteneffekte benennt.

Pure funktionale Programmiersprachen basieren auf puren Funktionen, und auch in Java muss nicht jede Methode einen äußeren Zustand verändern. Allerdings sind es Java-Entwickler gewohnt, in Zuständen zu denken, und daran ist an sich nichts Falsches: Ein Textdokument im Speicher ist eben ein Objektgraph genauso wie eine grafische Anwendung mit Eingabefeldern. Worauf funktionale Programmierung abzielt, sind die Operationen auf den Datenstrukturen und Berechnungen, die ohne Seiteneffekte sind.

Pure Funktionen ohne Zustand haben den Vorteil, dass sie

- ▶ beliebig oft ausgeführt werden können, ohne dass sich Systemzustände ändern,
- ▶ in beliebiger Reihenfolge ausgeführt werden können, ohne dass das Ergebnis ein anderes wird.

Diese Vorteile sind reizvoll unter dem Gesichtspunkt der Parallelisierung, denn die Prozessoren werden nicht wirklich schneller, aber wir haben mehr Prozessorkerne zur Verfügung. Pure Funktionen erlauben es Bibliotheken, Aufgaben wie Suchen und Filtern auf Kerne zu verteilen und so zu parallelisieren. Je weniger Zustand dabei im Spiel ist, desto besser, denn je weniger Zustand, desto weniger Synchronisation und Warteeffekte gibt es.

Aufpassen müssen Entwickler natürlich trotzdem, denn ein Lambda-Ausdruck muss nicht pur sein und kann Seiteneffekte haben. Daher ist es wichtig zu wissen, wann diese Lambda-Ausdrücke vielleicht nebenläufig sind und eine Synchronisation nötig ist.

### Beispiel



Die Schnittstelle `Iterable` deklariert eine Methode `forEach(...)`, mit einem Parameter vom Typ einer funktionalen Schnittstelle. Hier ist ein Lambda-Ausdruck möglich. Es wäre natürlich grundlegend falsch, wenn dieser Lambda-Ausdruck selbst in die Sammlung eingreift:

```
List<Integer> ints = new ArrayList<>( Arrays.asList( 1, 99, 2 ) );
ints.forEach( v -> { System.out.println( ints + ", " + v); ints.set( v, 0 ); } );
```

Die Ausgabe ist weit von dem, was erwartet wurde, aber kein Wunder, wenn Lambda-Ausdrücke illegale Seiteneffekte hervorrufen:

```
[1, 99, 2], 1
[1, 0, 2], 0
[0, 0, 2], 2
```

Die Vermeidung von Zuständen gekoppelt an die Unveränderbarkeit von Werten (engl. *immutability*) erhöht das Verständnis des Programms, da Entwickler es schwer haben, im Kopf das System mit den ganzen Änderungen »nachzuspielen«, insbesondere wenn diese Änderungen noch nebenläufig sind. Das zeigt das vorangehende Beispiel recht gut; solche Systeme zu verstehen und zu debuggen ist schwer. Je weniger Seiteneffekte es gibt, desto einfacher ist das Programm zu verstehen. Zustände machen ein Programm komplex, nicht nur in nebenläufigen Umgebungen. Wenn die Methode pur ist, muss ein Entwickler nichts anderes tun, als den Code der Methode zu verstehen. Wenn die Methode von Zuständen des Objekts abhängt, muss ein Entwickler den Code der gesamten Klasse verstehen. Und wenn das Objekt von Zuständen im Gesamtprogramm abhängt, ufert das Ganze aus, denn dann ist noch viel mehr vom System zu verstehen.

## 12.10 Zum Weiterlesen

Funktional zu programmieren ändert grundlegend das Design von Java-Programmen: weg von Methoden mit Seiteneffekten hin zu kleinen Funktionen, die Objekte mit neuen Zuständen liefern. Die Zukunft wird uns Muster und Best Practices an die Hand geben, wie in Java

entwickelt wird. Auch wird sich zeigen, ob weitere Konzepte der funktionalen Programmierung in Java bzw. die JVM einfließen werden. Bisher ist zum Beispiel Immutability kein Sprachkonstrukt, sondern durch die API gewährleistet, wenn es keine Setter oder Schreibzugriffe auf Variablen gibt; Reflection kann aber auch hier einen bösen Strich durch die Rechnung machen. Ansätze für Java 10 im Bereich Pattern Matching stellt Brian Goetz im Video [https://www.youtube.com/watch?v=n3\\_8YcYKScw](https://www.youtube.com/watch?v=n3_8YcYKScw) vor; unter <http://openjdk.java.net/jeps/305> gibt es die Projektbeschreibung. Die Optimierung von Endrekursion (engl. *tail call optimization*) auf Seiten der JVM, etwas, was in anderen funktionalen Programmiersprachen immer hochgehalten wird, ist aktuell nicht in der HotSpot JVM implementiert.

Entwickler, die noch tiefer in die Denkweise funktionaler Programmierung eintauchen möchten, können sich mit puren funktionalen Programmiersprachen wie Haskell beschäftigen und müssen dort ohne Seiteneffekte auskommen. Etwas einfacher für Java-Programmierer ist die Sprachfamilie ML, die auch imperative Elemente wie `while`-Schleifen bietet. Für Java-Programmierer wirkt das meist fremd, die hippe Programmiersprache Scala vereint objektorientierte und funktionale Programmierung nahezu perfekt. Java-Entwickler profitieren am meisten von den funktionalen Ansätzen bei der Stream-API, die schon kurz angerissen wurde und im zweiten Insel-Band »Java SE 11 Standard-Bibliothek« intensiv diskutiert wird.

# Kapitel 13

## 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)

Während die Beispiele aus [Kapitel 2 bis Kapitel 12](#) 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.

### 13.1 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 *Softwarearchitektur* 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 Architekturnuster 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 Ahnung 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 Möglichkeiten; 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 sie 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.

## 13.2 Design-Patterns (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?

### 13.2.1 Motivation für Design-Patterns

Die Gedanken über grundlegende Designkriterien gehen weit zurück. 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 setzten ja auch Unterprogramme nur deswegen ein, 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 zeich-

net 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.

### 13.2.2 Singleton

Ein *Singleton* ist eine Klasse, von der es in einer Anwendung nur ein Exemplar gibt.<sup>1</sup> Nützlich ist das für Dinge, die es nur genau einmal in einer Applikation geben soll, und davon gibt es einige Beispiele:

- ▶ Eine grafische Anwendung hat nur ein Fenster.
- ▶ Eine Konsolenanwendung hat nur je einen Eingabe-/Ausgabestrom.
- ▶ Alle Druckaufträge wandern in eine Drucker-Warteschlage.

Unbestreitbar ist, dass es einmalige Objekte gibt, variantenreich ist jedoch der Weg dahin. Im Prinzip lässt sich unterscheiden zwischen einem Ansatz, bei dem

1. sich ein Framework um den einmaligen Aufbau des Objekts kümmert und dann auf Anfrage das Objekt liefert oder
2. bei dem wir selbst in Java-Code ein Singleton realisieren.

Die bessere Lösung ist, ein Framework zu nutzen, namentlich CDI, Guice, Spring, Java EE; doch die Java SE enthält keines davon, weswegen wir zur Demonstration den expliziten Weg gehen.

---

<sup>1</sup> Pro Klassenlader, um das etwas genauer auszudrücken

Die technischen Realisierungen sind vielseitig; in Java bieten sich zur Realisierung von Singletons Aufzählungen (`enum`) und normale Klassen an. Im Folgenden wollen wir ein Szenario annehmen, bei dem eine Anwendung zentral auf Konfigurationsdaten zurückgreifen möchte.

### Singletons über Aufzählungen

Einen guten Weg für Singletons bieten Aufzählungen – auf den ersten Blick scheint ein Aufzählungstyp nicht dafür gemacht, denn eine Aufzählung impliziert ja irgendwie mehr als ein Element; doch die Eigenschaften vom `enum` sind perfekt für ein Singleton, und die Bibliothek implementiert einige Tricks, das Objekt auch möglichst nur einmal zu erzeugen, etwa dann, wenn die Aufzählung serialisiert über die Leitung geht. Die Idee dabei ist, genau ein Element anzubieten – gerne `INSTANCE` genannt –, das letztendlich ein Exemplar der Aufzählungsklasse wird, und die normalen Methoden:

**Listing 13.1** com/tutego/insel/pattern/singleton/Configuration.java, Configuration

```
public enum Configuration {
    INSTANCE;
    private Properties props = new Properties( System.getProperties() );
    public String getVersion() {
        return "1.2";
    }

    public String getUserDir() {
        return props.getProperty( "user.dir" );
    }
}
```

Der Typ `Configuration` deklariert neben der später öffentlichen statischen Variable `INSTANCE` auch noch eine interne Variable `props`, die von der Aufzählung genutzt werden kann, um dort Zustände abzulegen oder zu erfragen. Wir machen das im Beispiel nur lesend über `getUserDir()`.

Ein Nutzer greift wie üblich auf die `enum`-Eigenschaften zu:

**Listing 13.2** com/tutego/insel/pattern/singleton/ConfigurationDemo.java, main()

```
System.out.println( Configuration.INSTANCE.getVersion() ); // 1.2
System.out.println( Configuration.INSTANCE.getUserDir() ); // C:\Users\...
```

### 13.2.3 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 ist jedoch,

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 zum Beispiel

- ▶ 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 zu Tage. 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:

**Listing 13.3** java.util.Calendar, getInstance()

```
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.

### 13.2.4 Das Beobachter-Pattern mit Listener realisieren

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.

Eine Implementierung des Beobachter-Musters erlauben *Listener*. Es gibt Ereignisauslöser, die spezielle Ereignisobjekte aussenden, und Interessenten, die sich bei den Auslösern anmelden und abmelden. Die beteiligten Klassen und Schnittstellen folgen einer bestimmten Namenskonvention; XXX steht im Folgenden stellvertretend für einen Ereignisnamen, wie Window, Click ...:

- ▶ 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. Diese Schnittstelle kann 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 Ereignisobjekte sollen den Werbespruch (Slogan) speichern:

**Listing 13.4** 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 traditionell alle Ereignisklassen 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. Zwingend ist der Einsatz der Basisklasse nicht unbedingt. Sie wurde auch nicht im Laufe der Jahre generisch angepasst, sodass source lediglich Object ist.

Der zweite Parameter des AdEvent-Konstruktors ist unsere Werbung.

Der AdListener ist die Schnittstelle, die Interessenten implementieren:

**Listing 13.5** com/tutego/insel/pattern/listener/AdListener.java

```

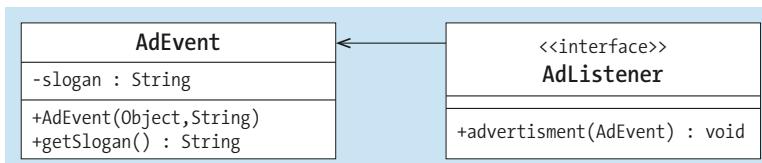
package com.tutego.insel.pattern.listener;

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(AdEvent) vor. Ob die Schnittstelle @FunctionalInterface tragen soll, muss sorgsam abgewogen werden, denn es wäre ja möglich, dass in Zukunft der Ereignisauslöser eine weitere Methode aufrufen möchte, um zum Beispiel etwas anderes zu melden.



**Abbildung 13.1** UML-Klassendiagramm von AdListener, das AdEvent referenziert

### Bemerkung

Falls wir die Callback-Methode nicht besonders benennen möchten, lässt sich auch eine allgemeine Listener-Schnittstelle der folgenden Art einsetzen:

```
public interface Listener<T extends Object, U extends Object> {
    void notifyObserver( T source, U data );
}
```

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 13.6** 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, einmal über eine innere anonyme Klasse und einmal über einen Lambda-Ausdruck:

**Listing 13.7** com/tutego/insel/pattern/listener/RadioDemo.java, main()

```

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() );
r.addAdListener( e -> System.out.println( "Ich höre nichts" ) );

```

Die Java-API-Dokumentation enthält einige generische Typen:

`class javax.swing.event.EventListenerList`

- `EventListenerList()`  
Erzeugt einen Container für Horcher.
- `<T extends EventListener> void add(Class<T> t, T l)`  
Fügt einen Listener `l` vom Typ `T` hinzu.
- `Object[] getListenerList()`  
Liefert ein Array aller Listener.
- `<T extends EventListener> T[] getListeners(Class<T> t)`  
Liefert ein Array 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.

### 13.3 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://shop.oreilly.com/product/9780596007126.do>, ISBN 978-0596007126, auf Deutsch übersetzt unter dem Titel »Entwurfsmuster von Kopf bis Fuß«.

# Kapitel 14

## Komponenten, JavaBeans und Module

»Das Ganze ist mehr als die Summe seiner Teile.«

– Aristoteles (384 v. u. Z.–322 v. u. Z.)

Nachdem wir uns das Zusammenspiel von Klassen angeschaut haben, wollen wir uns in diesem Kapitel mit der Frage beschäftigen, wie einzelne Typen oder Verbände von anderen Programmen gut wiederverwendet werden können. Wir wollen uns das bei einzelnen Komponenten anschauen und bei einer Sammlung von Klassen, die zu Archiven zusammengebounden werden.

### 14.1 JavaBeans

Die Architektur von *JavaBeans* ist ein einfaches Komponentenmodell. 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 deren 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 bei Zustandsänderungen Ereignisse auslösen.
- ▶ *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.

#### 14.1.1 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 »Age«, so bietet die JavaBean die Methoden `getAge()` und `setAge(...)` an.
- ▶ *Indizierte/Indexierte Eigenschaften* (engl. *indexed properties*). Sie werden eingesetzt, falls mehrere gleiche Eigenschaften aus einem Array verwaltet werden. So lassen sich Arrays gleichen Datentyps verwalten.
- ▶ *Gebundene Eigenschaften* (engl. *bound properties*). Ändert eine JavaBean ihren Zustand, kann sie angemeldete Interessenten (Listener) informieren.
- ▶ *Eigenschaft mit Votorecht* (engl. *veto properties*, auch *constraint properties* bzw. *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.

### 14.1.2 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 )`

### 14.1.3 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 Array von Werten intern gespeichert ist, müssen wir Zugriff auf bestimmte Werte bekommen. Daher erwarten die `setXXX(...)/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 )`

### 14.1.4 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 viel-

leicht 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 Ereignisobjekt 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 eine Klasse an, `PropertyChangeSupport`, die die JavaBeans nutzen können, um die Listener zu verwalten. Die Interessenten lassen sich mit `addPropertyChangeListener(...)` als Zuhörer einführen 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 14.1 src/main/java/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`.

```

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 Wertänderung. 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 Ganzzahl- und Wahrheitswerten
- `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 14.2** src/main/java/com/tutego/insel/bean/bound/PersonWatcher.java, main()

```
PropertyChangeListener psl = (PropertyChangeEvent e) ->
    System.out.printf("Property '%s': '%s' -> '%s'%n",
        e.getPropertyName(), e.getOldValue(), e.getNewValue());
Person p = new Person();
p.addPropertyChangeListener(psl);
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.

```
interface java.beans.PropertyChangeEvent
extends java.util.EventListener
```

- `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.

### 14.1.5 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 Wertänderung 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 ist. Das bricht den Setter ab, und es kommt nicht zum `firePropertyChange(...)`. Wegen der `PropertyVetoException` muss auch die Setter-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 14.3** src/main/java/com/tutego/insel/bean/veto/Person.java

```
package com.tutego.insel.bean.veto;

import java.beans.*;

public class Person {

    private boolean isBigamist;

    private PropertyChangeSupport changes = new PropertyChangeSupport( this );
    private VetoableChangeSupport vetos = new VetoableChangeSupport( this );

    public void setBigamist( boolean isBigamist ) throws PropertyVetoException {
        boolean oldValue = this.isBigamist;
        vetos.fireVetoableChange( "bigamist", oldValue, isBigamist );
        this.isBigamist = isBigamist;
        changes.firePropertyChange( "bigamist", oldValue, isBigamist );
    }

    public boolean isBigamist() {
        return isBigamist;
    }

    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 wie im ersten Beispiel an, um alle gültigen Zustandswechsel auszugeben:

**Listing 14.4** src/main/java/com/tutego/insel/bean/veto/PersonWatcher.java, main(), Teil 1

```

Person p = new Person();
p.addPropertyChangeListener( e ->
    System.out.printf( "Property '%s': '%s' -> '%s'%n",
        e.getPropertyName(), e.getOldValue(), e.getNewValue() ) );

```

Ohne ein Veto gehen alle Zustandsänderungen durch:

**Listing 14.5** src/main/java/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 vetoeinlegenden VetoableChangeListener eine Änderung nicht mehr erlaubt:

**Listing 14.6** src/main/java/com/tutego/insel/bean/veto/PersonWatcher.java, main(), Teil 3

```
VetoableChangeListener vetoableChangeListener = (PropertyChangeEvent e) -> {
    if ( "bigamist".equals( e.getPropertyName() ) && (Boolean) e.getNewValue() )
        throw new PropertyVetoException( "Nimm zwei ist nichts für mich!", e );
};

p.addVetoableChangeListener( vetoableChangeListener );
```

Durch Auslösung der Exception mit `throw new PropertyVetoException` im Veto-Fall wird eine unerwünschte Wertänderung verhindert:

**Listing 14.7** src/main/java/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.lambda$1(PersonWatcher.java:24)
at java.desktop/>
java.beans.VetoableChangeSupport.fireVetoableChange(VetoableChangeSupport.java:381)
at java.desktop/>
java.beans.VetoableChangeSupport.fireVetoableChange(VetoableChangeSupport.java:274)
at java.desktop/>
java.beans.VetoableChangeSupport.fireVetoableChange(VetoableChangeSupport.java:332)
at com.tutego.insel.bean.veto.Person.setBigamist(Person.java:18)
at com.tutego.insel.bean.veto.PersonWatcher.main(PersonWatcher.java:29)
```

Obwohl es mit `PropertyChangeListener` und `VetoableChangeListener` jeweils zwei Listener gibt, empfangen beide Listener in ihren Methoden `propertyChange(PropertyChangeEvent evt)` und `vetoableChange(PropertyChangeEvent evt)` ein Ereignisobjekt vom Typ `PropertyChange-`

Event. 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 `VetoableChangeListener` hinzu, der alle gewünschten Änderungen meldet.
- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener)`  
Fügt einen `VetoableChangeListener` 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`.

```
class java.beans.VetoableChangeChangeListener
implements Serializable
```

- `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.

## 14.2 Klassenlader (Class Loader) und Modul-/Klassenpfad

Ein Klassenlader ist dafür verantwortlich, die Binärrepräsentation einer Klasse aus einem Hintergrundspeicher oder Hauptspeicher zu laden. Aus der Datenquelle (im Allgemeinen die `.class`-Datei) liefert der Klassenlader ein Byte-Array mit den Informationen, die im zweiten Schritt dazu verwendet werden, die Klasse ins Laufzeitsystem einzubringen; das ist *Linking*. Es gibt vordefinierte Klassenlader und die Möglichkeit, eigene Klassenlader zu schreiben, um etwa verschlüsselte Klassendateien vom Netzwerk zu beziehen oder komprimierte `.class`-Dateien aus Datenbanken zu laden.

### 14.2.1 Klassenladen auf Abruf

Nehmen wir zu Beginn ein einfaches Programm mit drei Klassen:

```
package com.tutego.insel.tool;

public class HambachForest {
    public static void main( String[] args ) {
        boolean rweWantsToCutTrees = true;
        Forrest hambachForest = new Forrest();
        if ( rweWantsToCutTrees ) {
            Protest<Forrest> p1 = new Protest<>();
            p1.believeIn = hambachForest;
        }
    }
}

class Forrest { }

class Protest<T> {
    T believeIn;
    java.time.LocalDate since;
}
```

Wenn die Laufzeitumgebung das Programm `HambachForest` startet, muss sie eine Reihe von Klassen laden. Das tut sie dynamisch zur Laufzeit. Sofort wird klar, dass es zumindest `HambachForest` sein muss. Und da die JVM die statische `main(String[])`-Methode aufruft und Optionen übergibt, muss auch `String` geladen sein. Unsichtbar stecken noch andere referenzierte Klassen dahinter, die nicht direkt sichtbar sind. So wird zum Beispiel `Object` geladen, da implizit in der Klassendeklaration von `HambachForest` steht: `class HambachForest extends Object`. Intern ziehen die Typen viele weitere Typen nach sich. `String` implementiert `Serializable`, `CharSequence` und `Comparable`, also müssen diese drei Schnittstellen auch geladen werden. Und so geht das weiter, je nachdem, welche Programmfpade abgelaufen werden. Wichtig ist aber, zu verstehen, dass diese Klassendateien so spät wie möglich geladen werden.

### 14.2.2 Klassenlader bei der Arbeit zusehen

Im Beispiel lädt die Laufzeitumgebung selbstständig die Klassen (*implizites Klassenladen*). Klassen lassen sich auch mit `Class.forName(String)` über ihren Namen laden (*explizites Klassenladen*).

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 Ma-

schine beim Lauf alle Typen aus, die sie lädt. Nehmen wir das Beispiel von eben, so ist die Ausgabe mit dem aktivierten Schalter unter Java 11 fast 500 Zeilen lang; ein Ausschnitt:

```
$ java -verbose:class com.tutego.insel.tool.HambachForest
[0.010s][info][class,load] opened: C:\Program Files\Java\jdk-11\lib\modules
[0.032s][info][class,load] java.lang.Object source: jrt:/java.base
[0.032s][info][class,load] java.io.Serializable source: jrt:/java.base
[0.033s][info][class,load] java.lang.Comparable source: jrt:/java.base
[0.036s][info][class,load] java.lang.CharSequence source: jrt:/java.base
[0.037s][info][class,load] java.lang.String source: jrt:/java.base
...
[0.684s][info][class,load] sun.security.util.Debug source: jrt:/java.base
[0.685s][info][class,load] com.tutego.insel.tool.HambachForest source: file:/C:/>
Inselprogramme/target/classes/
[0.687s][info][class,load] java.lang.PublicMethods$MethodList source: jrt:/java.base
[0.687s][info][class,load] java.lang.PublicMethods$Key source: jrt:/java.base
[0.689s][info][class,load] java.lang.Void source: jrt:/java.base
[0.690s][info][class,load] com.tutego.insel.tool.Forrest source: file:/C:/>
Inselprogramme/target/classes/
[0.691s][info][class,load] jdk.internal.misc.TerminatingThreadLocal$1 source: jrt:/>
java.base
[0.692s][info][class,load] java.lang.Shutdown source: jrt:/java.base
[0.692s][info][class,load] java.lang.Shutdown$Lock source: jrt:/java.base
```

Ändern wir die Variable `rweWantsToCutTrees` in `true`, so wird unsere Klasse `Protest` geladen, und in der Ausgabe kommt nur eine Zeile hinzu! Das wundert auf den ersten Blick, denn die Klasse referenziert `LocalDate`. Doch ein `LocalDate` wird nicht benötigt, also auch nicht geladen. Der Klassenlader bezieht nur Klassen, wenn sie für den Programmablauf benötigt werden, nicht aber durch die reine Deklaration als Attribut. Wenn wir `LocalDate` mit zum Beispiel `LocalDate.now()` initialisieren, kommen stattliche 200 Klassendateien hinzu.

### 14.2.3 JMOD-Dateien und JAR-Dateien

Der Klassenlader bezieht `.class`-Dateien nicht nur aus Verzeichnissen, sondern in der Regel aus Containern. So müssen keine Verzeichnisse ausgetauscht werden, sondern nur einzelne Dateien. Als Containerformate finden wir JMOD (neu in Java 9) und JAR. Wenn Java-Software ausgeliefert wird, bieten sich JAR- oder JMOD-Dateien an, denn es ist einfacher und platzsparender, nur ein komprimiertes Archiv weiterzugehen als einen großen Dateibaum.

#### JAR-Dateien

Sammlungen von Java-Klassendateien und Ressourcen werden in der Regel in *Java-Archiven*, kurz *JAR-Dateien*, zusammengefasst. Diese Dateien sind im Grunde ganz normale ZIP-Archi-

ve mit einem besonderen Verzeichnis *META-INF* für Metadateien. Das JDK bringt im *bin*-Verzeichnis das Werkzeug *jar* zum Aufbau und Extrahieren von JAR-Dateien mit.

JAR-Dateien behandelt die Laufzeitumgebung wie Verzeichnisse von Klassendateien und Ressourcen. Zudem haben Java-Archive den Vorteil, dass sie signiert werden können und illegale Änderungen auffallen. JAR-Dateien können Modulinformationen beinhalten, dann heißen sie englisch *modular JAR*.

### JMOD-Dateien

Das Format JMOD ist speziell für Module, und es organisiert Typen und Ressourcen. Zum Auslesen und Packen gibt es im *bin*-Verzeichnis des JDK das Werkzeug *jmod*.

#### Hinweis

Die JVM greift selbst nicht auf diese Module zurück. Achten wir auf die Ausgabe vom letzten Programm, dann steht in der ersten Zeile:

```
[0.007s][info][class,load] opened: C:\Program Files\Java\jdk-11\lib\modules
```

Die Datei *module* ist ca. 170 MiB groß und in einem proprietären Dateiformat.



### JAR vs. JMOD

Module können in JMOD- und JAR-Container gepackt werden. Wenn ein JAR kein Modular JAR ist, also keine Modulinformationen enthält, so fehlen zentrale Informationen, wie Abhängigkeiten oder eine Version; ein JMOD ist immer ein benanntes Modul.

JMOD-Dateien sind nicht so flexibel wie JAR-Dateien, denn sie können nur zur Übersetzungszeit und zum Linken eines Runtime-Images – dafür gibt es das Kommandozeilenwerkzeug *jlink* – genutzt werden. JMOD-Dateien können nicht wie JAR-Dateien zur Laufzeit verwendet werden. Das Dateiformat ist proprietär und kann sich jederzeit ändern, es ist nichts Genaues spezifiziert.<sup>1</sup> Einziger Vorteil von JMOD: Native Bibliotheken lassen sich standardisiert einbinden.

#### 14.2.4 Woher die kleinen Klassen kommen: die Suchorte und spezielle Klassenlader

Die Laufzeitumgebung nutzt zum Laden nicht nur einen Klassenlader, sondern mehrere. Das ermöglicht, unterschiedliche Orte für die Klassendateien festzulegen. Ein festes Schema bestimmt die Suche nach den Klassen:

1. Klassentypen 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

<sup>1</sup> Die <http://openjdk.java.net/jeps/261> macht die Aussage, dass es ein ZIP ist.

eines Systems gehören, werden sie *Bootstrap-Klassen* genannt. Die Implementierung dieses *Bootstrap-Klassenladers* ist Teil der Laufzeitumgebung.

2. Ist eine Klasse keine Bootstrap-Klasse, beginnt der *System-Klassenlader* bzw. *Applikations-Klassenlader* die Suche im *Modulpfad* (ehemals *Klassenpfad/Classpath*). Diese Pfadangabe besteht aus einer Aufzählung von Modulen, in denen die Laufzeitumgebung nach den Klassendateien und Ressourcen sucht.

#### 14.2.5 Setzen des Modulpfades

Wo die JVM die Klassen findet, muss ihr mitgeteilt werden, und das ist in der Praxis elementar für die Auslieferung, auch englisch *deployment* genannt. Java wartet mit dem Laden der Klassen so lange, bis sie benötigt werden. Es gibt zum Beispiel Programmabläufe nur zu besonderen Bedingungen, und wenn dann erst spät ein neuer Typ referenziert wird, der nicht vorhanden ist, fällt dieser Fehler erst sehr spät auf. Dem Compiler müssen folglich nicht nur die Quellen für Klassen und Ressourcen der eigenen Applikation mitgeteilt werden, sondern alle vom Programm referenzierten Typen aus zum Beispiel quelloffenen und kommerziellen Bibliotheken.

Sollen in einem Java-Projekt Dateien aus einem Verzeichnis oder einem externen Modul geholt werden, so ist der übliche Weg, diese Dateien im Modulpfad anzugeben. Diese Angabe ist für alle SDK-Werkzeuge notwendig – am häufigsten ist sie beim Compiler und bei der Laufzeitumgebung zu sehen.

#### Setzen des Klassenpfades

Vor Java 9 gab es nur JAR-Dateien und Verzeichnisse im Klassenpfad. Auch wenn es ab Java 9 weiterhin den Klassenpfad gibt, sollte er auf lange Sicht leer sein.

Es gibt zwei Möglichkeiten zur Aufnahme von Verzeichnissen und JAR-Dateien in den Klassenpfad:

- ein Schalter
- eine Umgebungsvariable

#### Schalter -classpath

Die Suchorte lassen sich flexibel angeben, wobei die erste Variante einem SDK-Werkzeug über den Schalter **-classpath** (kurz **-cp**) die Klassendateien bzw. Archive liefert:

```
$ java -classpath classpath1;classpath2 mein.paket.MainClass
```

Der Klassenpfad enthält Wurzelverzeichnisse der Pakete und JAR-Dateien, also Archive von Klassendateien und Ressourcen.



### Beispiel

Nimm ein Java-Archiv *library.jar* im aktuellen Verzeichnis, die Ressourcen unter dem *bin*-Verzeichnis und alle JAR-Dateien im Verzeichnis *lib* in den Klassenpfad mit auf:

```
$ java -cp "library.jar;bin/.;lib/*" mein.paket.MainClass
```

Unter Windows ist der Trenner ein Semikolon, unter Unix ein Doppelpunkt. Das Sternchen steht für *alle* JAR-Dateien, es ist *keine* übliche Wildcard, wie z. B. *parser\*.jar*.<sup>2</sup> Sehen Kommandozeilen der Betriebssysteme ein \*, beginnen sie in der Regel eine eigene Verarbeitung; daher muss die gesamte Pfadangabe in doppelten Anführungszeichen stehen.

### Umgebungsvariable CLASSPATH

Eine Alternative zum Schalter *-cp* ist das Setzen der Umgebungsvariablen *CLASSPATH* mit einer Zeichenfolge, die Pfadangaben spezifiziert:

```
$ SET CLASSPATH=classpath1;classpath2
$ java mein.paket.MeinClass
```

Problematisch ist der globale Charakter der Variablen, sodass lokale *-cp*-Angaben besser sind. Außerdem »überschreiben« die *-cp*-Optionen die Einträge in *CLASSPATH*. Zu guter Letzt: Ist weder *CLASSPATH* noch eine *-cp*-Option gesetzt, besteht der Klassenpfad für die JVM nur aus dem aktuellen Verzeichnis, also ».«.

Um in Eclipse den Klassenpfad zu erweitern, damit etwa die Klassendateien von Java-Archiven berücksichtigt werden, ist Folgendes zu tun: Im Projekt das Kontaktmenü öffnen und PROPERTIES aufrufen, dann links unter JAVA BUILD PATH gehen und anschließend im Reiter LIBRARIES entweder ADD JARS... (JARs sind im Projekt) oder ADD EXTERNAL JARS... (JAR-Daten liegen nicht im Projekt, sondern irgendwo anders im Dateisystem) nutzen.



### Hinweis

Die so genannten Bootstrap-Klassen aus den Paketen *java(x).\** (wie *Object*, *String*) stehen nicht im *CLASSPATH*.



### Classpath-Hell

Java-Klassen in JAR-Dateien auszuliefern, ist der übliche Weg, es gibt aber zwei Probleme:

1. Aus Versehen können zwei JAR-Dateien mit unterschiedlichen Versionen im Klassenpfad liegen. Nehmen wir an, es sind *parser-1.2.jar* und *parser-2.0.jar*, wobei sich bei der neuen Version API und Implementierung leicht geändert haben. Das fällt vielleicht am Anfang

<sup>2</sup> Weitere Details unter <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html>

nicht auf, denn einen Ladefehler gibt es für den Typ nicht, er ist ja da – die JVM nimmt den ersten Typ, den sie findet. Nur wenn ein Programm auf die neue API zurückgreift, aber die geladene Klasse vom alten JAR stammt, knallt es zur Laufzeit. Bei doppelten JARs mit unterschiedlichen Versionen führt eine Umsortierung im Klassenpfad zu einem ganz anderen Ergebnis. Zum Glück lässt sich das Problem relativ schnell lösen.

2. Zwei Java-Bibliotheken – nennen wir sie *vw.jar* und *audi.jar* – benötigen je eine Neben-JAR zum Arbeiten. Doch während *vw.jar* die Version *bosch-1.jar* benötigt, benötigt *audi.jar* die Version *bosch-2.jar*. Das ist ein Problem, denn JARs sind im Standard-Klassenpfad immer global, aber nicht hierarchisch, es kann also kein JAR ein »lokales« Unter-JAR haben.

Lösungen für das zweite Problem gibt es einige, wobei zu neuen Klassenladern gegriffen wird. Bekannt ist OSGi, das in der Java-Welt aber etwas an Fahrt verloren hat.

## 14.3 Module entwickeln und einbinden

Das *JPMS (Java Platform Module System)*, auch unter dem Projektnamen *Jigsaw* bekannt, ist eine der größten Neuerungen seit Java 9. Im Mittelpunkt steht die starke Kapselung: Implementierungsdetails kann ein Modul geheim gehalten. Selbst Hilfscode innerhalb des Moduls, auch wenn er öffentlich ist, darf nicht nach außen dringen. Zweitens kommt eine Abstraktion von Verhalten über Schnittstellen hinzu, die interne Klassen aus dem Modul implementieren können, wobei dem Nutzer die konkreten Klassen nicht bekannt sind. Als dritten Punkt machen explizite Abhängigkeiten die Interaktion mit anderen Modulen klar. Eine grafische Darstellung hilft auch bei großen Architekturen, die Übersicht über Nutzungsbeziehungen zu behalten.

### 14.3.1 Wer sieht wen

Klassen, Pakete und Module lassen sich als Container mit unterschiedlichen Sichtbarkeiten sehen:

- Ein Typ, sei es Klasse oder Schnittstelle, enthält Attribute und Methoden.
- Ein Paket enthält Typen.
- Ein Modul enthält Pakete.
- Private Eigenschaften in einem Typ sind nicht in anderen Typen sichtbar.
- Nicht öffentliche Typen sind in anderen Paketen nicht sichtbar.
- Nicht exportierte Pakete sind außerhalb eines Moduls nicht sichtbar.

Ein Modul ist definiert

1. durch einen Namen,

2. durch die Angabe, was es exportiert und
3. welches Modul es zur Arbeit selbst benötigt.

Interessant ist der zweite Aspekt, also dass ein Modul etwas exportiert. Wenn nichts exportiert wird, ist auch nichts sichtbar nach außen. Alles, was Außenstehende sehen sollen, muss in der Modulbeschreibung aufgeführt sein – nicht alle öffentlichen Typen des Moduls sind standardmäßig öffentlich, dann wäre das kein Fortschritt zu JAR-Dateien. Mit dem neuen Modulsystem haben wir also eine ganz andere Sichtbarkeit. Aus der Viererbande `public`, `private`, `paketsichtbar`, `protected` bekommt `public` eine viel feinere Abstufung. Denn was `public` ist, bestimmt das Modul, und das sind:

- ▶ Typen, die das Modul für alle exportiert
- ▶ Typen für explizit aufgezählte Module
- ▶ alle Typen im gleichen Modul

Der Compiler und die JVM achten auf die Einhaltung der Sichtbarkeit, und auch Tricks mit Reflection sind nicht mehr möglich, wenn ein Modul keine Freigabe erteilt hat.

## Modultypen

Wir wollen uns in dem Abschnitt intensiver mit drei Modultypen beschäftigen. Wenn wir neue Module schreiben, dann sind das *benannte Module*. Daneben gibt es aus Kompatibilitätsgründen *automatische Module* und *unbenannte Module*, mit denen wir vorhandene JAR-Dateien einbringen können. Die Bibliothek der Java SE ist selbst in Module unterteilt, wir nennen sie *Plattform-Module*.

Die Laufzeitumgebung zeigt mit einem Schalter `--list-modules` alle Plattform-Module an. Der Einsatz des Schalters vor Java 9 führt zu einem Fehler.

### Beispiel

Liste die ca. 70 Module auf:

```
$ java --list-modules
java.base@11
java.compiler@11
java.datatransfer@11
...
jdk.unsupported.desktop@11
jdk.xml.dom@11
jdk.zipfs@11
```

[zB]

Im Ordner `C:\Program Files\Java\jdk-11\jmods` liegen JMOD-Dateien.

### 14.3.2 Plattform-Module und JMOD-Beispiel

Das Kommandozeilenwerkzeug `jmod` zeigt an, was ein Modul exportiert und benötigt. Nehmen wir die JDBC-API für Datenbankverbindungen als Beispiel; die Typen sind in einem eigenen Modul mit dem Namen `java.sql`.

```
C:\Program Files\Java\jdk-11\jmods>jmod describe java.sql.jmod
java.sql@11
exports java.sql
exports javax.sql
requires java.base mandated
requires java.logging transitive
requires java.transaction.xa transitive
requires java.xml transitive
uses java.sql.Driver
platform windows-amd64
```

Wir können ablesen:

- ▶ den Namen
- ▶ die Pakete, die das Modul exportiert: `java.sql` und `javax.sql`
- ▶ die Module, die `java.sql` benötigt: `java.base` ist hier immer drin, dazu kommen weitere.
- ▶ Die Meldung mit »uses« steht im Zusammenhang mit dem Service-Locator – wir können das vorerst ignorieren.
- ▶ Die Kennung über die Plattform (`windows-amd64`) schreibt `jmod` mit hinein, es ist die Belegung der System-Property `os.arch` auf dem Build-Server.

### 14.3.3 Interne Plattformeigenschaften nutzen, `--add-exports`

Als Sun im letzten Jahrhundert mit der Entwicklung der Java-Bibliotheken begann, kamen eine Reihe interner Hilfsklassen mit in die Bibliothek. Viele beginnen mit den Paketpräfixen `com.sun` und `sun`. Die Typen wurden immer als interne Typen kommuniziert, doch bei einigen Entwicklern waren die Neugierde und das Interesse so groß, dass die Warnungen von Sun/Oracle ignoriert wurden. In Java 9 gab es den großen Knall, da `public` nicht mehr automatisch `public` für alle Klassen außerhalb des Moduls ist; die internen Klassen werden nicht mehr exportiert, sind also nicht mehr benutzbar.

#### Akt 1, der Quellcode

Es kommt zu einem Compilerfehler, wie in folgendem Beispiel:

**Listing 14.8** com/tutego/insel/tools>ShowRuntimeArguments.java

```
package com.tutego.insel.tool;

public class ShowRuntimeArguments {
    public static void main( String[] args ) throws Exception {
        System.out.println( java.util.Arrays.toString(
            jdk.internal.misc.VM.getRuntimeArguments() ) );
    }
}
```

Unser Programm greift auf die `VM`-Klasse zurück, um die eigentliche Belegung der Kommandozeile zu erfragen. Was wir in der `main(String[] args)`-Methode über `args` empfangen, enthält keine JVM-Argumente.

**Akt 2: der Compilerfehler**

Ab Java 9 lässt sich das Programm nicht mehr ohne Compilerfehler übersetzen.

```
$ javac com/tutego/insel/tool>ShowRuntimeArguments.java
com\tutego\insel\tool\
ShowRuntimeArguments.java:6: error: package jdk.internal.misc is not visible
    jdk.internal.misc.VM.getRuntimeArguments() );
                           ^
(package jdk.internal.misc is declared in module java.base, which does not export
it to the unnamed module)
1 error
```

Das Problem dokumentiert der Compiler: `jdk.internal.misc` ist nicht für unser Programm zugänglich.

**Akt 3: der magische Compiler-Schalter**

Zwar ist die Klasse `VM` selbst `public` und die Methode `getRuntimeArguments()` ebenfalls `public`, aber `jdk.internal.misc` wurde nicht exportiert, also ist der Zugriff von unserem Programm nicht möglich, denn die JVM realisiert eine Zugriffskontrolle. Allerdings können wir diese abschalten. Mit dem Schalter `--add-exports` stellen wir aus dem Modul `java.base` das Paket `jdk.internal.misc` unserer Klasse bereit. Die allgemeine Syntax ist so:

```
--add-exports <source-module>/<package>=<target-module>(,<target-module>)*
```

Die Angabe ist für den Compiler und für die Laufzeitumgebung zu setzen; für unser Beispiel:

```
$ javac --add-exports java.base/jdk.internal.misc=ALL-UNNAMED com/tutego/insel/tool/
ShowRuntimeArguments.java
```

### Akt 4: die zickige JVM

Das Programm ist compiliert, führen wir es aus:

```
$ java com/tutego/insel/tool>ShowRuntimeArguments
Exception in thread "main" java.lang.IllegalAccessError: class com.tutego.insel.tool.ShowRuntimeArguments (in unnamed module @0x4d591d15) cannot access class jdk.internal.misc.VM (in module java.base) because module java.base does not export jdk.internal.misc to unnamed module @0x4d591d15
at com.tutego.insel.tool>ShowRuntimeArguments.main>ShowRuntimeArguments.java:6)
```

Es funktioniert nicht! Aber die Fehlermeldung kommt uns bekannt vor, und wir wissen warum ...

### Akt 4: Die JVM will das, was der Compiler will. Schluss

Wir müssen den gleichen Schalter wie für den Compiler setzen:

```
$ java --add-exports java.base/jdk.internal.misc=ALL-UNNAMED ShowRuntimeArguments
[--add-exports=java.base/jdk.internal.misc=ALL-UNNAMED]
```

Wir sehen die Ausgabe, das Programm funktioniert.

Eine Angabe wie `java.base/jdk.internal.misc`, bei der vorne das Modul steht und hinter dem / der Paketname, ist oft ab Java 9 anzutreffen. Hinter dem Gleichheitszeichen steht entweder unser Paket, welches die Typen in `jdk.internal.misc` sehen kann, oder – wie in unserem Fall – `ALL-UNNAMED`.

### jdeps

Hätten wir das Programm schon erfolgreich ab Java 9 übersetzt, würde es zur Laufzeit ebenfalls knallen. Da es nun sehr viel Programmcode gibt, haben die Java-Entwickler bei Oracle das Kommandozeilenprogramm `jdeps` entwickelt. Es meldet, wenn interne Typen im Programm vorkommen:

```
$ jdeps com/tutego/insel/tool>ShowRuntimeArguments.class
ShowRuntimeArguments.class -> java.base
com.tutego.insel.tool -> java.io          java.base
com.tutego.insel.tool -> java.lang        java.base
com.tutego.insel.tool -> java.util        java.base
com.tutego.insel.tool -> jdk.internal.misc JDK internal API (java.base)
```

Anders als beim Java-Compiler ist der volle Dateiname, also mit `.class`, nötig. Die Meldung »JDK internal API« bereitet uns darauf vor, dass es gleich Ärger geben wird.

So kann relativ leicht eine große Codebasis untersucht werden, und Entwickler können proaktiv den Stellen auf den Grund gehen, die problematische Abhängigkeiten haben.

#### 14.3.4 Neue Module einbinden, --add-modules und --add-opens

Jedes Java SE-Projekt basiert auf dem Modul `java.se`, was diverse Modulabhängigkeiten nach sich zieht.

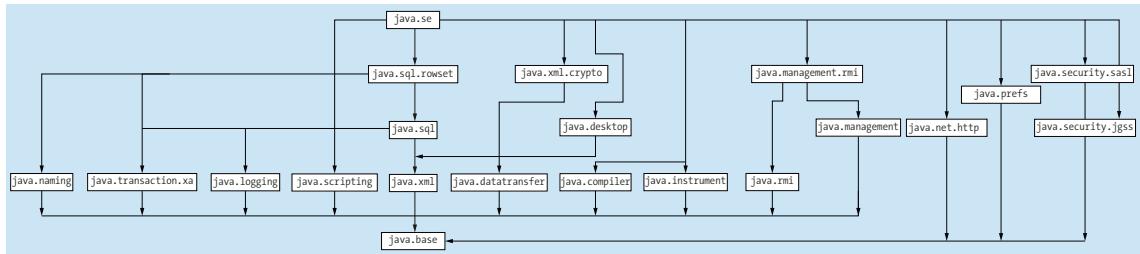


Abbildung 14.1 Modulabhängigkeiten von `java.se`

#### Hinweis

Nicht alle installierten Module sind im `java.se`-Modul enthalten, dazu zählen die JDK-Module und `java.smartcardio`. Sie befinden sich jedoch standardmäßig im Modulpfad, wie folgender »Einzeiler« zeigt:

```
ModuleLayer.boot().modules().stream()
    .map( Module::getName )
    .sorted()
    .reduce( ( s1, s2 ) -> s1 + ", " + s2 )
    .ifPresent( System.out::println );
```

Es listet auf:

```
java.base, java.compiler, java.datatransfer, java.desktop, java.instrument,
java.logging, java.management, java.management.rmi, java.naming, java.net.http,
java.prefs, java.rmi, java.scripting, java.security.jgss, java.security.sasl,
java.smartcardio, java.sql, java.sql.rowset, java.transaction.xa, java.xml,
java.xml.crypto, jdk.accessibility, jdk.attach, jdk.charsets, jdk.compiler,
jdk.crypto.cryptoki, jdk.crypto.ec, jdk.crypto.mscapi, jdk.dynalink, jdk.editpad,
jdk.httpserver, jdk.internal.ed, jdk.internal.jvmstat, jdk.internal.le,
jdk.internal.opt, jdk.jartool, jdk.javadoc, jdk.jconsole, jdk.jdeps, jdk.jdi,
jdk.jdwp.agent, jdk.jfr, jdk.jlink, jdk.jshell, jdk.jsobject, jdk.jstated,
jdk.localedata, jdk.management, jdk.management.agent, jdk.management.jfr,
jdk.naming.dns, jdk.naming.rmi, jdk.net, jdk.scripting.nashorn, jdk.sctp,
jdk.security.auth, jdk.security.jgss, jdk.unsupported, jdk.unsupported.desktop,
jdk.xml.dom, jdk.zipfs
```

Alle diese Module können ohne Schalter direkt verwendet werden.



## Neue Module zu den Kernmodulen hinzufügen und öffnen

Sollen externe Module zu den Kernmodulen hinzugenommen werden, so geschieht das mit dem Schalter `--add-modules`. Ein weiterer Schalter ist `--add-opens`, der ein Paket für Reflection öffnet. Neben `--add-opens` gibt es das ähnliche `--add-exports`, das alle öffentlichen Typen und Eigenschaften zur Übersetzungs-/Laufzeit öffnet; `--add-opens` geht für Reflection einen Schritt weiter.

### 14.3.5 Projektabhängigkeiten in Eclipse

Um Module praktisch umzusetzen, wollen wir in Eclipse zwei neue Java-Projekte aufbauen: `com.tutego.greeter` und `com.tutego.main`. Wir legen im Projekt `com.tutego.greeter` eine Klasse `com.tutego.insel.greeter.Greeter` an und in `com.tutego.main` die Klasse `com.tutego.insel.main.Main`. Im Package-Explorer sieht das so aus:

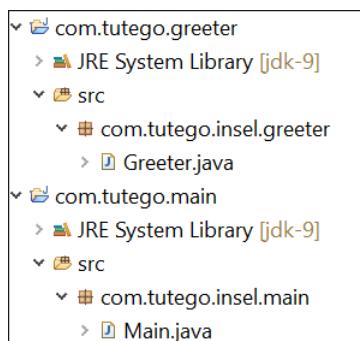


Abbildung 14.2 Java-Projekte `com.tutego.greeter` und `com.tutego.main` im Package-Explorer

Jetzt ist eine wichtige Vorbereitung in Eclipse nötig: Wir müssen einstellen, dass `com.tutego.main` das Java-Projekt `com.tutego.greeter` benötigt. Dazu gehen wir auf das Projekt `com.tutego.main` und rufen im Kontextmenü **PROJECT** auf, alternativ im Menüpunkt **PROJECT • PROPERTIES** oder über die Tastenkombination **`Alt`+`Shift`+`Left Arrow`**. Im Dialog navigiere links auf **JAVA BUILD PATH** und aktiviere den Reiter **PROJECTS**. Wähle **ADD...**, und im Dialog wähle aus der Liste **COM.TUTEGO.GREETER**. **OK** schließt den kleinen Dialog, und unter **REQUIRED PROJECTS IN BUILD PATH** taucht eine Abhängigkeit auf (siehe Abbildung 14.3).

Wir können jetzt zwei einfache Klassen implementieren. Zunächst für das Projekt `com.tutego.greeter`:

Listing 14.9 `com/tutego/insel/greeter/Greeter.java`

```
package com.tutego.insel.greeter;
```

```
public class Greeter {
```

```

private Greeter() { }

public static Greeter instance() {
    return new Greeter();
}

public void greet( String name ) {
    System.out.println( "Hey " + name );
}
}

```

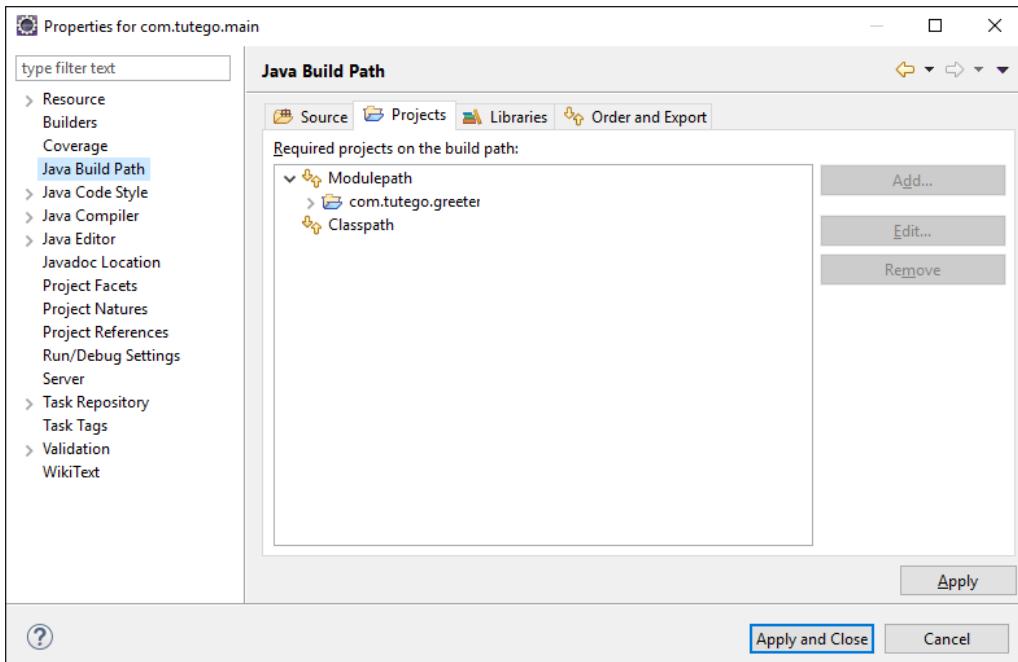


Abbildung 14.3 Abhängigkeit des Eclipse-Projektes com.tutego.main von com.tutego.greeter

Und die Hauptklasse im Projekt *com.tutego.main*:

**Listing 14.10** com/tutego/insel/main/Main

```

package com.tutego.insel.main;

import com.tutego.insel.greeter.Greeter;

public class Main {

```

```

public static void main( String[] args ) {
    Greeter.instance().greet( "Chris" );
}
}

```

Da wir in Eclipse vorher die Abhängigkeit gesetzt haben, gibt es keinen Compilerfehler.

#### 14.3.6 Benannte Module und module-info.java

Die Modulinformationen werden über eine Datei *module-info.java* (kurz Modulinfodatei) deklariert, Annotationen kommen nicht zum Einsatz. Diese zentrale Datei ist der Hauptunterschied zwischen einem Modul und einer einfachen JAR-Datei. In dem Moment, in dem die spezielle Klassendatei *module-info.class* im Modulpfad ist, beginnt die Laufzeitumgebung, das Projekt als Modul zu interpretieren.



Testen wir das, indem wir in unsere Projekte *com.tutego.greeter* und *com.tutego.main* eine Modulinfodatei anlegen. Das kann Eclipse über das Kontextmenü CONFIGURE • CREATE MODULE-INFO.JAVA für uns machen.

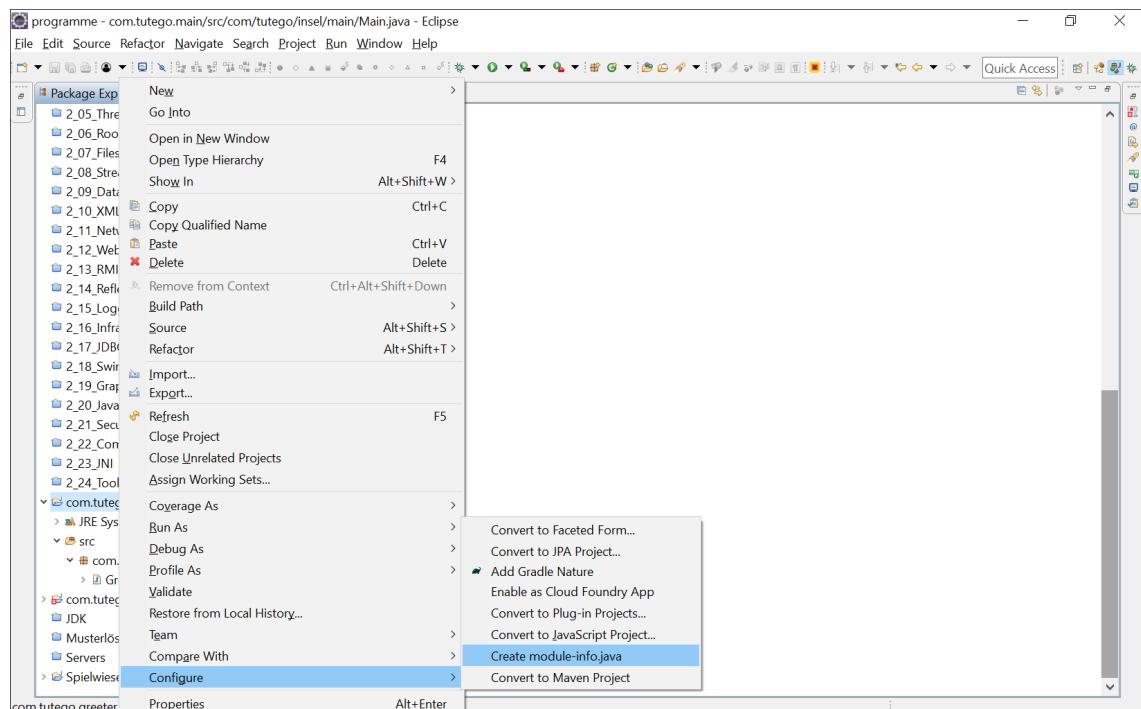


Abbildung 14.4 Datei *module-info* anlegen

Für das erste Modul com.tutego.greeter entsteht:

**Listing 14.11** module-info.java

```
/**  
 *  
 */  
/**  
 * @author Christian  
 *  
 */  
module com.tutego.greeter {  
    exports com.tutego.insel.greeter;  
    requires java.base;  
}
```

Und für die zweite Modulinfodatei – Kommentare ausgeblendet:

**Listing 14.12** module-info.java

```
module com.tutego.main {  
    exports com.tutego.insel.main;  
    requires com.tutego.greeter;  
    requires java.base;  
}
```

Hinter dem Schlüsselwort `module` steht der Name des Moduls, den Eclipse automatisch so wählt, wie das Eclipse-Projekt heißt.<sup>3</sup> Es folgt ein Block in geschweiften Klammern.

Zwei Schlüsselwörter fallen ins Auge, die wir schon vorher bemerkt haben: `exports` und `requires`.

- ▶ Das Projekt/Modul `com.tutego.greeter` exportiert ausschließlich das Paket `com.tutego.insel.greeter`; andere Pakete nicht. Es benötigt (`requires`) `java.base`, wobei das Modul Standard ist und die Zeile gelöscht werden kann.
- ▶ Das Projekt/Modul `com.tutego.main` exportiert das Paket `com.tutego.insel.main`, und es benötigt `com.tutego.greeter` – diese Information nimmt sich Eclipse selbstständig aus den Projektabhängigkeiten.

#### Info

Ein Modul required ein anderes Modul, aber exports ein Paket.

<sup>3</sup> Zur Benennung von Modulen gibt es Empfehlungen in dem englischsprachigen Beitrag <http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2017-May/000687.html>.

Beginnen wir mit den Experimenten in den beiden `module-info.java`-Dateien:

Modul	Aktion	Ergebnis
com.tutego.greeter com.tutego.main	// requires java.base;	Auskommentieren führt zu keiner Änderung, da <code>java.base</code> immer required wird.
com.tutego.greeter	// exports com.tutego.insel.greeter;	Compilerfehler im <code>main</code> -Modul: »The type com.tutego.insel.greeter.Greeter is not accessible«
com.tutego.greeter	exports com.tutego.insel.greeter to god;	Nur das Modul <code>god</code> bekommt Zugriff auf <code>com.tutego.insel.greeter</code> . Das <code>main</code> -Modul meldet »The type com.tutego.insel.greeter.Greeter is not accessible«.
com.tutego.greeter	exports com.tutego.insel.closer;	Hinzufügen führt zum Compilerfehler »The package com.tutego.insel.closer does not exist or is empty«.
com.tutego.main	// requires com.tutego.greeter;	Compilerfehler »The import com.tutego.insel.greeter cannot be resolved«
com.tutego.main	// exports com.tutego.insel.main;	Keins, denn <code>c.t.i.m</code> wird von keinem Modul required.

Tabelle 14.1 Modulsyntax und ihre Effekte

Die Zeile mit `exports com.tutego.insel.greeter to god` zeigt einen qualifizierten Export.

### Übersetzen und Packen von der Kommandozeile

Setzen wir ins Wurzelverzeichnis des Moduls `com.tutego.greeter` ein Batch-Skript `compile.bat`; wir nehmen an, dass das JDK-`bin`-Verzeichnis im PATH ist.

#### Listing 14.13 compile.bat

```
rmdir /s /q lib
mkdir lib
javac -d bin src\module-info.java src\com\tutego\insel\greeter\Greeter.java
jar --create --file=lib/com.tutego.greeter@1.0.jar --module-version=1.0 -C bin .
jar --describe-module --file=lib/com.tutego.greeter@1.0.jar
```

Folgende Schritte führt das Skript aus:

1. Löschen eines vielleicht schon angelegten *lib*-Ordners
2. Anlegen eines neuen *lib*-Ordners für die JAR-Datei
3. Übersetzen der zwei Java-Dateien in den Zielordner *bin*
4. Anlegen einer JAR-Datei. *--create* (abkürzbar zu *-c*) instruiert das Werkzeug, eine neue JAR-Datei anzulegen. *-file* (oder kurz *-f*) bestimmt den Ziellamen, *-module-version* unsere Versionsnummer, und *-C* wechselt das Verzeichnis und beginnt ab dort, die Dateien einzusammeln. Die Kommandozeilsyntax beschreibt Oracle auf der Webseite <https://docs.oracle.com/javase/10/tools/jar.htm>.
5. Die Option *--describe-module* (oder kurz *-d*) zeigt die Modulinformation und führt zu folgender (vereinfachten) Ausgabe: com.tutego.greeter@1.0 jar:file:///C:/.../com.tutego.greeter/lib/com.tutego.greeter@1.0.jar!/module-info.class exports com.tutego.insel.greeter requires java.base.

Für das zweite Projekt ist die *compile.bat* sehr ähnlich, dazu kommt ein Aufruf der JVM, um das Programm zu starten.

#### **Listing 14.14 compile.bat**

```
rmdir /s /q lib
mkdir lib
javac -d bin --module-path ..\com.tutego.greeter\lib src\module-info.java >
src\com\tutego\insel\main>Main.java
jar -c -f=lib\com.tutego.main@1.0.jar --main-class=com.tutego.insel.main.Main >
--module-version=1.0 -C bin .
java -p lib;..\com.tutego.greeter\lib -m com.tutego.main
```

Änderungen gegenüber dem ersten Skript sind:

1. Beim Compilieren müssen wir den Modulpfad mit *--module-path* (oder kürzer mit *-p*) angeben, weil ja das Modul *com.tutego.greeter* required ist.
2. Beim Anlegen der JAR-Datei geben wir über *--main-class* die Klasse mit der *main(...)*-Methode an.
3. Startet die JVM das Programm, lädt sie das Hauptmodul und alle abhängigen Module. Wir geben beide *lib*-Ordner mit den JAR-Dateien an und mit *-m* das so genannte *initiale Modul* für die Hauptklasse.

#### **14.3.7 Automatische Module**

JAR-Dateien spielen seit 20 Jahren eine zentrale Rolle im Java-System; sie vom einen zum anderen Tag abzuschaffen, würde große Probleme bereiten. Ein Blick auf <https://mvnrepository.com/repos> offenbart über 7,7 Millionen Artefakte; es gehen auch Dokumentationen und

andere Dateien in die Statistik ein, doch es gibt eine Größenordnung, wie viele JAR-Dateien im Umlauf sind.

Damit JAR-Dateien ab Java 9 eingebracht werden können, gibt es zwei Lösungen: das JAR in den Klassenpfad oder in den Modulpfad zu setzen. Kommt ein JAR in den Modulpfad und hat es keine Modulinfodatei, entsteht ein *automatisches Modul*. Bis auf eine kleine Einschränkung funktioniert das für die meisten existierenden Java-Bibliotheken.

Ein automatisches Modul hat gewisse Eigenschaften für den Modulnamen und Konsequenzen in den Abhängigkeiten:

- ▶ Ohne Modulinfo haben die automatischen Module keinen selbstgewählten Namen, sondern sie bekommen vom System einen Namen zugewiesen, der sich aus dem Dateinamen ergibt.<sup>4</sup> Vereinfacht gesagt: Angehängte Versionsnummern und die Dateiendung werden entfernt und alle nichtalphanumerischen Zeichen durch Punkte ersetzt, jedoch nicht zwei Punkte hintereinander.<sup>5</sup> Die Version wird erkannt. Die Dokumentation gibt das Beispiel `foo-bar-1.2.3-SNAPSHOT.jar` an, was zum Modulnamen `foo.bar` und der Version `1.2.3-SNAPSHOT` führt.
- ▶ Automatische Module exportieren immer alle ihre Pakete. Wenn es also eine Abhängigkeit zu diesem automatischen Modul gibt, kann der Bezieher alle sichtbare Typen und Eigenschaften verwenden.
- ▶ Automatische Module können alle anderen Module lesen, auch die unbenannten.

Auf den ersten Blick scheint eine Migration für das Modulsystem einfach: Alle JARs kommen in den Modulpfad, egal, ob es Modulinfodateien gibt oder nicht. Allerdings gibt es JAR-Dateien, die von der JVM als automatisches Modul abgelehnt werden, wenn sie nämlich Typen eines Paketes enthalten und dieses Paket sich schon in einem anderen aufgenommenen Modul befindet. Module dürfen keine »split packages« enthalten, also das gleiche Paket noch einmal enthalten. Die Migration erfordert dann a) das Zusammenlegen der Pakete zu einem Modul, b) die Verschiebung in unterschiedliche Pakete oder c) die Nutzung des Klassenpfades.

### 14.3.8 Unbenanntes Modul

Eine Migration auf eine neue Java-Version sieht in der Regel so aus, dass zuerst die JVM gewechselt und geprüft wird, ob die vorhandene Software weiterhin funktioniert. Laufen die Testfälle durch und gibt es keine Auffälligkeiten im Testbetrieb, kann der Produktivbetrieb unter der neuen Version erfolgen. Gibt es keine Probleme, können nach einiger Zeit die neuen Sprachmittel und Bibliotheken verwendet werden.

---

<sup>4</sup> Automatic-Module-Name in die *META-INF*-Datei zu setzen, ist eine Alternative, dazu später mehr.

<sup>5</sup> [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/module/ModuleFinder.html#of\(java.nio.file.Path...\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/module/ModuleFinder.html#of(java.nio.file.Path...)).

Eine vorhandene Java 8-Software muss inklusive aller Einstellungen und Einträge im Klassenpfad weiterhin laufen. Das heißt, eine Laufzeitumgebung ab Java 9 kann den Klassenpfad nicht ignorieren. Da es intern nur einen Modulpfad gibt, müssen auch diese JAR-Dateien zu Modulen werden. Die Lösung ist das *unbenannte Modul* (engl. *unnamed module*): Jedes JAR im Klassenpfad – dabei spielt es keine Rolle, ob es eine *modul-info.class* enthält – kommt in das unbenannte Modul. Davon gibt es nur eines, wir sprechen also im Singular, nicht Plural.

»Unbenannt« sagt schon, dass das Modul keinen Namen hat und folglich auch keine Abhängigkeit zu den JAR-Dateien im unbenannten Modul existieren kann; das ist der Unterschied zu einem automatischen Modul. Ein unbenanntes Modul hat die gleiche Eigenschaft wie ein automatisches Modul, dass es alle Pakete exportiert. Und weil es zur Migration gehört, hat ein unbenanntes Modul auch Zugriff auf alle anderen Module.

### 14.3.9 Lesbarkeit und Zugreifbarkeit

Die Laufzeitumgebung sortiert Module in einen Graphen ein. Die Abhängigkeit der Module führt dabei zur so genannten *Lesbarkeit* (engl. *readability*): Benötigt Modul A Modul B, so *liest* A Modul B, und B wird von A *gelesen*. Für die Funktionsweise des Modulsystems ist dies elementar, denn so werden zur Übersetzungszeit schon Fehler ausgeschlossen, wie Zyklen oder gleiche Pakete in unterschiedlichen Modulen. Die Lesbarkeit ist zentral für eine zuverlässige Konfiguration, engl. *reliable configuration*.

Einen Schritt weiter geht der Begriff der *Erreichbarkeit/Zugänglichkeit* (engl. *accessibility*). Wenn ein Modul ein anderes Modul grundsätzlich lesen kann, bedeutet das noch nicht, dass es an alle Pakete und Typen kommt, denn nur diejenigen Typen sind sichtbar, die exportiert worden sind. Lesbare und erreichbare Typen nennen sich *erreichbar*.

Die nächste Frage ist, welcher Modultyp auf welchen anderen Modultyp Zugriff hat. [Tabelle 14.2](#) fasst die Lesbarkeit am besten zusammen.

Modultyp	Ursprung	Exportiert Pakete	Hat Zugriff auf
Plattformmodul	JDK	Explizit	
Benannte Module	Container mit Modulinfo im Modulpfad	Explizit	Plattformmodule, andere benannte Module, automatische Module
Automatische Module	Container ohne Modulinfo im Modulpfad	Alle	Plattformmodule, andere benannte Module, automatische Module, unbenanntes Modul

Tabelle 14.2 Lesbarkeit der Module

Modultyp	Ursprung	Exportiert Pakete	Hat Zugriff auf
Unbenanntes Modul	Klassendateien und JARs im Klassenpfad	Alle	Plattformmodule, benannte Module, automatische Module

Tabelle 14.2 Lesbarkeit der Module (Forts.)

Der Modulinfodatei kommt dabei die größte Bedeutung zu, denn sie macht aus einem JAR ein Modular JAR; fehlt die Modulinformation, bleibt es ein normales JAR, wie sie Java-Entwickler seit 20 Jahren kennen. Die JAR-Datei kann neu in den Modulpfad kommen oder in den bekannten Klassenpfad. Das ergibt vier Kombinationen:

	Modulpfad	Klassenpfad
JAR mit Modulinformation	Wird benanntes Modul.	Wird unbenanntes Modul.
JAR ohne Modulinformation	Wird automatisches Modul.	Wird unbenanntes Modul.

Tabelle 14.3 JARs im Pfad

JAR-Archive im Klassenpfad sind das bekannte Verhalten, weswegen auch ein Wechsel von Java 8 auf Java 11 möglich sein sollte.

### 14.3.10 Modul-Migration

Nehmen wir an, unsere monolithische Applikation hat keine Abhängigkeiten zu externen Bibliotheken und soll modularisiert werden. Dann besteht der erste Schritt darin, die gesamte Applikation in ein großes benanntes Modul zu setzen. Als Nächstes müssen die einzelnen Bereiche identifiziert werden, damit nach und nach die Bausteine in einzelne Module wandern. Das ist nicht immer einfach, zumal zyklische Abhängigkeiten nicht unwahrscheinlich sind. Bei der Modularisierung des JDK hatten die Oracle-Entwickler viel Mühe.

#### Das Problem mit automatischen Modulen

Traditionell generieren Build-Werkzeuge wie Maven oder Gradle JAR-Dateien, und ein Dateiname hat sich irgendwie ergeben. Werden jedoch diese JAR-Dateien zu automatischen Modulen, spielt der Dateiname plötzlich eine große Rolle. Doch bewusst wurde der Dateiname vermutlich nie gewählt. Referenziert ein benanntes Modul ein automatisches Modul, bringt das zwei Probleme mit sich: Ändert sich der Dateiname – lassen wir die Versionsnummer einmal außen vor –, heißt auch das automatische Modul anders, und die Abhängigkeit kann nicht mehr aufgelöst werden. Das zweite Problem ist größer: Viele Java-Bibliotheken haben noch keine Modulinformationen, und folglich werden Entwickler eine Abhängigkeit zu die-

sem automatischen Modul über den abgeleiteten Namen ausdrücken. Nehmen wir z. B. die beliebte Open-Source-Bibliothek Google Guava. Die JAR-Datei hat den Dateinamen `guava-27.0-jre.jar` – guava heißt folglich das automatische Modul. Ein benanntes Modul (nennen wir es `M1`) kann über `required` Guava eine Abhängigkeit ausdrücken. Konvertiert Google die Bibliothek in ein echtes Java 9-Modul, dann wird sich der Name ändern – geplant ist `com.google.guava`. Und ändert sich der Name, führen alle Referenzierungen in Projekten zu einem Compilerfehler; ein Alias wäre eine tolle Idee, das gibt es jedoch nicht. Und das Problem besteht ja nicht nur im eigenen Code, der Guava referenziert; referenziert das eigene Modul `M1` ein Modul `M2`, das wiederum Guava referenziert, so gibt es das gleiche Problem – wir sprechen von einer *transitiven Abhängigkeit*. Die Änderung des Modulnamens von Guava wird zum Problem, denn wir müssen warten, bis `M2` den Namen korrigiert, damit `M1` wieder gültig ist.

Eine Lösung mildert das Problem ab: In der JAR-Manifest-Datei kann ein Eintrag `Automatic-Module-Name` gesetzt werden – das »überschreibt« den automatischen Modulnamen.

### Beispiel

[zB]

Apache Commons setzt den Namen so:

`Automatic-Module-Name: org.apache.commons.lang3`

Benannte Module, die Abhängigkeiten zu automatischen Modulen besitzen, sind also ein Problem. Es ist zu hoffen, dass die zentralen Java-Bibliotheken, auf die sich so viele Lösungen stützen, schnell Modulinformationen einführen. Das wäre eine Lösung von unten nach oben, englisch *bottom-up*. Das ist das Einzige, was erfolgversprechend ist, aber wohl auch eine lange Zeit benötigen wird. Auch jetzt, einige Zeit nach dem Java 9-Release, haben nur wenige Java-Bibliotheken eine Modulinformation, `Automatic-Module-Name` kommt häufiger vor.

## 14.4 Zum Weiterlesen

Wer sich in die Modulgeschichte einlesen möchte, der bekommt gute Einblicke über diverse Präsentationen der Java-Macher: <https://openjdk.java.net/projects/jigsaw/>. Das Modulsystem bietet noch mehr Möglichkeiten, wie die Öffnung für Reflection oder transitive Abhängigkeiten, also *requires transitive*, damit ein Modul ein anderes Modul für den Nutzer mitbestimmt. Weiterhin existiert eine `provide ... with-` und `uses`-Syntax für Service-Loader. Gibt es eine Schnittstelle `I` und eine Implementierung `C`, so kann die Modulinformationen `provides I with C` enthalten, und der Service-Loader kann mit `ServiceLoader.load(I.class)` den Typ `C` liefern, ohne *META-INF/services*-Dateien, die vor Java 9 nötig waren. Weiterhin können JDK-Module gegen andere ausgetauscht werden und Klassen »gepatcht«, indem sie anderen Modulen zugeordnet werden.

Auch wenn das neue Modulsystem interessant ist, gibt es Kritik, weil es einigen Teams nicht weit genug geht. Ein Kritikpunkt ist die Versionsnummer, die im Moment nur Zierde ist. Scott Stark, der Checkentwickler des bekannten JBoss AS Wildfly, fasst ausführlich Probleme unter <http://tutego.de/go/concerns-regarding-jigsaw> zusammen.

Der Buchmarkt hält ein paar Spezialtitel bereit, die die Thematik ausführlich beschreiben, etwa »Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications« von Sander Mak und Paul Bakker.

# Kapitel 15

## Die Klassenbibliothek

»Was wir brauchen, sind ein paar verrückte Leute;  
seht euch an, wohin uns die normalen gebracht haben.«  
– George Bernard Shaw (1856–1950)

### 15.1 Die Java-Klassenphilosophie

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 Ein-/Ausgabe oder grafischen Oberflächen ist Schluss. Die Java-Bibliothek dagegen versucht, von den plattformspezifischen Eigenschaften zu abstrahieren, und die Entwickler haben sich große Mühe gegeben, alle wichtigen Methoden in wohlgeformten objektorientierten Klassen und Paketen unterzubringen. Diese decken insbesondere die zentralen Bereiche Datenstrukturen, Ein- und Ausgabe, Grafik- und Netzwerkprogrammierung ab.

#### 15.1.1 Modul, Paket, Typ

An oberster Stelle der Java-Bibliothek stehen Module. Sie wiederum bestehen aus Paketen, die wiederum die Typen enthalten.

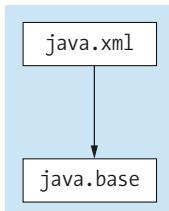
##### Module der Java SE

Die *Java Platform, Standard Edition* (»Java SE«) API besteht aus folgenden Modulen, die alle mit dem Präfix `java` beginnen.

Modul	Beschreibung
java.base	Fundamentale Typen der Java SE-Plattform
java.compiler	Java-Sprachmodell, Annotationsverarbeitung, Java Compiler API
java.datatransfer	API für den Datentransfer zwischen Applikationen, in der Regel die Zwischenablage
java.desktop	Grafische Oberflächen mit AWT und Swing, Accessibility-API, Audio, Drucken und JavaBeans
java.instrument	Instrumentalisierung ist die Veränderung der Java-Programme zur Laufzeit.
java.logging	Logging-API
java.management	Java Management Extensions (JMX)
java.management.rmi	RMI-Connector für den Remote-Zugriff auf die JMX-Beans
java.naming	Java Naming and Directory Interface (JNDI) API
java.prefs	Die Preferences API dient zum Speichern von Benutzereinstellungen.
java.rmi	Entfernte Methodenaufrufe; Remote Method Invocation (RMI) API
java.scripting	Scripting API
java.security.jgss	Java-Binding der IETF Generic Security Services API (GSS-API)
java.security.sasl	Java-Unterstützung für IETF Simple Authentication and Security Layer (SASL)
java.sql	JDBC API für den Zugriff auf relationale Datenbanken
java.sql.rowset	JDBC RowSet API
java.xml	XML-Klassen: Java API for XML Processing (JAXP), Streaming API for XML (StAX), Simple API for XML (SAX), W3C Document Object Model (DOM) API
java.xml.crypto	API für XML-Kryptografie

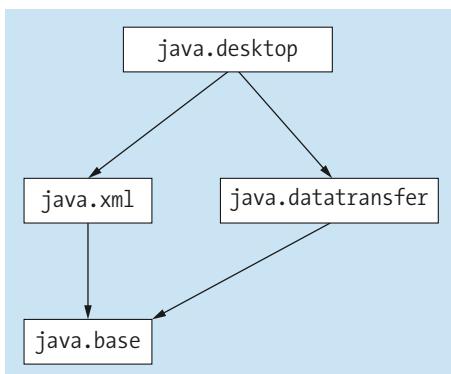
Tabelle 15.1 Module der Java SE

Das `java.base`-Modul ist das wichtigste Modul, und es enthält Kernklassen wie `Object` und `String` usw. Es ist das einzige Modul, das selbst keine Abhängigkeit zu anderen Modulen entält. Jedes andere Modul jedoch bezieht sich mindestens auf `java.base`. Die Javadoc stellt das schön grafisch dar.



**Abbildung 15.1** Das Modul `java.xml` hat eine Abhängigkeit zum `java.base`-Modul.

Zum Teil gibt es mehr Abhängigkeiten, etwa beim Modul `java.desktop`:



**Abbildung 15.2** Abhängigkeiten vom Modul »`java.desktop`«

### Modul `java.se`

Ein besonderes Modul ist `java.se`. Es deklariert selbst keine eigenen Pakete oder Typen, sondern fasst lediglich andere Module zusammen; der Name für so eine Konstruktion ist *Aggregator-Modul*. Das `java.se`-Modul definiert auf diese Weise die API für die Java SE-Plattform (siehe [Abbildung 15.3](#)).

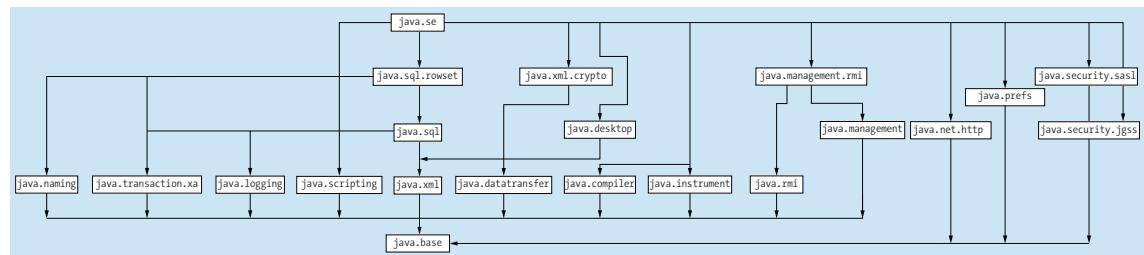


Abbildung 15.3 Abhängigkeiten vom Modul java.se



### Hinweis

Wir werden im Folgenden bei den Java SE-Typen nicht darauf eingehen, aus welchem Modul sie stammen. Die Kenntnis, in welchem Modul sich ein Typ befindet, ist nur dann interessant, wenn kleinere Teilmengen der Java SE gebaut werden.

### Weitere Module

Zwei weitere Module, die ebenfalls mit `java` beginnen, aber nicht zum Java SE-Standard zählen, sind `java.jnlp` (Java Network Launch Protocol) und `java.smartcardio` (Java-API für die Kommunikation mit Smart Cards nach ISO/IEC 7816-4).

Das JDK ist die Standardimplementierung der Java SE. Es liefert den Entwicklern weitere Pakete und Klassen, etwa mit einem HTTP-Server oder den Java-Werkzeugen wie Compiler und Javadoc-Tool. Es gibt mehrere Module, die alle mit dem Präfix `jdk` beginnen.



### Hinweis

Oracle hat aus Java 11 diverse Teile entfernt, etwa JavaFX oder Java EE-Module. Entwickler binden am besten die Referenzimplementierungen ein, <https://stackoverflow.com/questions/48204141/replacements-for-deprecated-jpms-modules-with-java-ee-apis> dokumentiert das. JavaFX war nie ein Teil des Java SE-Standards, sondern eine »Beigabe« vom Oracle JDK. Die Alternative ist, das *OpenJFX* einzubinden.

### 15.1.2 Übersicht über die Pakete der Standardbibliothek

Die Java 11 Core Java SE API besteht aus folgenden Modulen und Paketen – eine Kurzbeschreibung finden die Leser im Anhang:

Module	Enthaltene Pakete
java.base	java.io, java.lang, java.lang.annotation, java.lang.invoke, java.lang.module, java.lang.ref, java.lang.reflect, java.math, java.net, java.net.spi, java.nio, java.nio.channels, java.nio.channels.spi, java.nio.charset, java.nio.charset.spi, java.nio.file, java.nio.file.attribute, java.nio.file.spi, java.security, java.security.acl (deprecated), java.security.cert, java.security.interfaces, java.security.spec, java.text, java.text.spi, java.time, java.time.chrono, java.time.format, java.time.temporal, java.time.zone, java.util, java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks, java.util.function, java.util.jar, java.util.regex, java.util.spi, java.util.stream, java.util.zip, javax.crypto, javax.crypto.interfaces, javax.crypto.spec, javax.net, javax.net.ssl, javax.security.auth, javax.security.auth.callback, javax.security.auth.login, javax.security.auth.spi, javax.security.auth.x500, javax.security.cert
java.compiler	javax.annotation.processing, javax.lang.model, javax.lang.model.element, javax.lang.model.type, javax.lang.model.util, javax.tools
java.datatransfer	java.awt.datatransfer
java.desktop	java.applet, java.awt, java.awt.color, java.awt.desktop, java.awt.dnd, java.awt.event, java.awt.font, java.awt.geom, java.awt.im, java.awt.im.spi, java.awt.image, java.awt.image.renderable, java.awt.print, java.beans, java.beans.beancontext, javax.accessibility, javax.imageio, javax.imageio.event, javax.imageio.metadata, javax.imageio.plugins.bmp, javax.imageio.plugins.jpeg, javax.imageio.plugins.tiff, javax.imageio.spi, javax.imageio.stream, javax.print, javax.print.attribute, javax.print.attribute.standard, javax.print.event, javax.sound.midi, javax.sound.midi.spi, javax.sound.sampled, javax.sound.sampled.spi, javax.swing, javax.swing.border, javax.swing.colorchooser, javax.swing.event, javax.swing.filechooser, javax.swing.plaf, javax.swing.plaf.basic, javax.swing.plaf.metal, javax.swing.plaf.multi, javax.swing.plaf.nimbus, javax.swing.plaf.synth, javax.swing.table, javax.swing.text, javax.swing.text.html, javax.swing.text.html.parser, javax.swing.text.rtf, javax.swing.tree, javax.swing.undo
java.instrument	java.lang.instrument

Tabelle 15.2 Übersicht über die Pakete in den Modulen der Java 11 Core Java SE API

Module	Enthaltene Pakete
java.logging	java.util.logging
java.management	java.lang.management, javax.management, javax.management.loading, javax.management.modelmbean, javax.management.monitor, javax.management.openmbean, javax.management.relation, javax.management.remote, javax.management.timer
java.management.rmi	javax.management.remote.rmi
java.naming	javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap, javax.naming.spi
java.prefs	java.util.prefs
java.rmi	java.rmi, java.rmi.activation, java.rmi.dgc, java.rmi.registry, java.rmi.server, javax.rmi.ssl
java.scripting	javax.script
java.security.jgss	javax.security.auth.kerberos, org.ietf.jgss
java.security.sasl	javax.security.sasl
java.sql	java.sql, javax.sql, javax.transaction.xa
java.sql.rowset	javax.sql.rowset, javax.sql.rowset.serial, javax.sql.rowset.spi
java.xml	javax.xml, javax.xml.catalog, javax.xml.datatype, javax.xml.namespace, javax.xml.parsers, javax.xml.stream, javax.xml.stream.events, javax.xml.stream.util, javax.xml.transform, javax.xml.transform.dom, javax.xml.transform.sax, javax.xml.transform.stax, javax.xml.transform.stream, javax.xml.validation, javax.xml.xpath, org.w3c.dom, org.w3c.dom.bootstrap, org.w3c.dom.events, org.w3c.dom.ls, org.w3c.dom.ranges, org.w3c.dom.views, org.xml.sax, org.xml.sax.ext, org.xml.sax.helpers
java.xml.crypto	javax.xml.crypto, javax.xml.crypto.dom, javax.xml.crypto.dsig, javax.xml.crypto.dsig.dom, javax.xml.crypto.dsig.keyinfo, javax.xml.crypto.dsig.spec

Tabelle 15.2 Übersicht über die Pakete in den Modulen der Java 11 Core Java SE API (Forts.)

Entwickler sollten folgende Pakete von den Möglichkeiten her zuordnen können:

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 java.nio	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. Enthält unverzichtbare Klassen wie String-, Thread- oder Wrapper-Klassen.
java.net	Kommunikation über Netzwerke. Bietet Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP bzw. 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. Unterpakete kümmern sich um reguläre Ausdrücke und Nebenläufigkeit.
javax.swing	Swing-Komponenten für grafische Oberflächen. Das Paket besitzt diverse Unterpakete.

Tabelle 15.3 Wichtige Pakete in der Java SE

Eine vollständige Übersicht aller Pakete gibt der Anhang, »Java SE-Module und Paketübersicht«. Als Entwickler ist es unumgänglich, für die Details die Java-API-Dokumentation unter <https://docs.oracle.com/javase/10/docs/api/overview-summary.html> zu studieren.

### Offizielle Schnittstelle (java- und javax-Pakete)

Das, was die Java-Dokumentation aufführt, bildet den erlaubten Zugang zur Bibliothek. Die Typen sind im Grunde 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 außerdem Klassen, die unterstützt werden, aber nicht Teil der offiziellen API sind. Dazu zählen etwa diverse Swing-Klassen für das Aussehen der Oberfläche.

### Standard Extension API (`javax`-Pakete)

Einige der Java-Pakete beginnen mit `javax`. Dies sind ursprünglich Erweiterungspakete (*Extensions*), die die Kernklassen ergänzen sollten. Im Laufe der Zeit sind jedoch viele der früher zusätzlich einzubindenden Pakete in die Standarddistribution 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 oder ab welcher Java-Version das Paket Teil der Distribution ist. Echte externe Pakete sind unter anderem:

- ▶ *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*
- ▶ Spracheingabe/-ausgabe mit der *Java Speech API*
- ▶ *JavaSpaces* für gemeinsamen Speicher unterschiedlicher Laufzeitumgebungen
- ▶ *JXTA* zum Aufbau von P2P-Netzwerken

Im Endeffekt haben Entwickler es mit folgenden Bibliotheken zu tun:

1. der offiziellen Java-API
2. der API aus JSR-Erweiterungen, wie der Java-Enterprise-API
3. nichtoffiziellen Bibliotheken, wie quelloffenen Lösungen, etwa zum Zugriff auf PDF-Dateien oder Bankautomaten

## 15.2 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, 00:00:00 UTC vergangenen Millisekunden zurück.
- static long nanoTime()  
Liefert die Zeit vom genauesten System-Zeitgeber. Sie hat keinen Bezugspunkt zu irgend-einem Datum.

Die Differenz zweier Zeitwerte kann der groben Abschätzung von Ausführungszeiten von Programmen dienen:

**Listing 15.1** com/tutego/insel/lang/Profiling.java

```
package com.tutego.insel.lang;

import static java.util.concurrent.TimeUnit.NANOSECONDS;
import java.util.Arrays;
import java.util.function.Supplier;
import java.util.function.ToIntFunction;

class Profiling {

    final static String ANGIE =
        "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 static int MAX = 10000;

    enum Algorithm {
        STRING_BUILDER1( () -> { // StringBuffer(size) und append() zur Konkatenation
            StringBuilder sb = new StringBuilder( 2 * MAX * ANGIE.length() );
            for ( int i = MAX; i-- > 0; )
                sb.append( ANGIE ).append( ANGIE );
            return sb.toString().length();
        } ),
        ...
```

```

STRING_BUILDER2( () -> { // StringBuffer und append() zur Konkatenation
    StringBuilder sb = new StringBuilder();
    for ( int i = MAX; i-- > 0; )
        sb.append( ANGIE ).append( ANGIE );
    return sb.toString().length();
} ),
STRING_PLUS( () -> {      // + zur Konkatenation
    String s = "";
    for ( int i = MAX; i-- > 0; )
        s += ANGIE + ANGIE;
    return s.length();
} );

private final Supplier<Integer> supplier;
private Algorithm( Supplier<Integer> supplier ) { this.supplier = supplier; }
int perform() { return supplier.get(); }
}

private static long[] measure() {
    ToLongFunction<Algorithm> duration = algorithm -> {
        long startTime = System.nanoTime();
        int result = algorithm.perform();
        try { return NANOSECONDS.toMillis( System.nanoTime() - startTime ); }
        finally { System.out.println( result ); }
    };
    return Arrays.stream( Algorithm.values() ).mapToLong( duration ).toArray();
}

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(): 6 ms
    System.out.printf( "sb(), append()     : %d ms%n", durations[1] );
    // sb(), append()     : 9 ms
    System.out.printf( "t+=                  : %d ms%n", durations[2] );
    // t+=                  : 15982 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 Strings zusammen.

Vor der Messung gibt es zwei Testläufe und ein `System.gc()`, das die automatische Speicherbereinigung (GC) anweist, Speicher freizugeben. (Das würde in gewöhnlichen Programmen nicht stehen, da der Garbage-Collector 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 (JDK 10) liefert das Programm die Ausgabe:

```
sb(size), append(): 7 ms
sb(), append()      : 9 ms
t+=                 : 15982 ms
```

Das Ergebnis: Bei großen Anhängeoperationen ist es nur unwesentlich besser, einen passend in der Größe initialisierten `StringBuilder` zu benutzen. Über das + entstehen viele temporäre Objekte, was wirklich teuer kommt. Da in Java 9 die Konkatenation von Strings über den Plus-Operator beschleunigt wurde, sind die Zeiten besser als unter Java 8, wo die Ausführungszeit bei 41.262 ms liegt.

### Tipp

Die Werte von `nanoTime()` sind immer aufsteigend, was für `currentTimeMillis()` nicht zwingend gelten muss, da sich Java die Zeit vom Betriebssystem holt, und da kann sich die Systemzeit ändern, etwa wenn der Benutzer die Zeit anpasst. Differenzen von `currentTimeMillis()`-Zeitstempeln sind dann komplett falsch und könnten sogar negativ sein.



### Profiler

Wo im Programm überhaupt Taktzyklen verbraucht werden, zeigt ein *Profiler*. An diesen Stellen kann dann mit der Optimierung begonnen werden. *Java Mission Control* ist ein leistungsfähiges Programm vom JDK und integriert einen freien Profiler. *Java VisualVM* ist ein weiteres freies Programm, das bis Java 8 integriert war, jetzt allerdings unter <https://visualvm.github.io/> bezogen werden muss. NetBeans integriert einen Profiler, Informationen liefern <http://profiler.netbeans.org>. Auf der professionellen und kommerziellen Seite stehen sich *JProfiler* (<https://www.ej-technologies.com/products/jprofiler/overview.html>) und *YourKit* (<https://www.yourkit.com/java/profiler>) gegenüber.

## 15.3 Die Klasse Class

Angenommen, wir wollen einen Klassen-Browser schreiben. Dieser soll alle zum laufenden Programm gehörenden Klassen und darüber hinaus weitere Informationen anzeigen, wie etwa Variablenbelegung, deklarierte Methoden, Konstruktoren und Informationen über die Vererbungshierarchie. Dafür benötigen wir die Bibliotheksklasse `Class`. Exemplare der Klasse `Class` sind Objekte, die entweder eine Java-Klasse oder Java-Schnittstelle repräsentieren (dass auch Schnittstellen durch `Class`-Objekte repräsentiert werden, wird im Folgenden nicht mehr ausführlich unterschieden).

In diesem Punkt unterscheidet sich Java von vielen herkömmlichen Programmiersprachen, da sich Eigenschaften von Klassen vom gerade laufenden Programm mittels der `Class`-Objekte abfragen lassen. Bei den Exemplaren von `Class` handelt es sich um eine eingeschränkte Form von Meta-Objekten<sup>1</sup> – die Beschreibung eines Java-Typs, die aber nur ausgewählte Informationen preisgibt. Neben normalen Klassen werden auch Schnittstellen durch ein `Class`-Objekt repräsentiert, und sogar Arrays und primitive Datentypen – statt `Class` wäre wohl der Klassename `Type` passender gewesen ...

### 15.3.1 An ein Class-Objekt kommen

Zunächst müssen wir für eine bestimmte Klasse das zugehörige `Class`-Objekt in Erfahrung bringen. `Class`-Objekte selbst kann nur die JVM erzeugen. Wir können das nicht (die Objekte sind immutable, und der Konstruktor ist privat).<sup>2</sup> Um einen Verweis auf ein `Class`-Objekt zu bekommen, bieten sich folgende Lösungen an:

- ▶ Ist ein Exemplar der Klasse verfügbar, rufen wir die `getClass()`-Methode des Objekts auf und erhalten das `Class`-Exemplar der zugehörigen Klasse.
- ▶ Jeder Typ enthält eine statische Variable mit dem Namen `.class` vom Typ `Class`, die auf das zugehörige `Class`-Exemplar verweist.
- ▶ Auch auf primitiven Datentypen ist das Ende `.class` erlaubt. Das gleiche `Class`-Objekt liefert die statische Variable `TYPE` der Wrapper-Klassen. Damit ist `int.class == Integer.TYPE`.
- ▶ Die Klassenmethode `Class.forName(String)` kann eine Klasse erfragen, und wir erhalten das zugehörige `Class`-Exemplar als Ergebnis. Ist der Typ noch nicht geladen, sucht und bindet `forName(String)` die Klasse ein. Weil das Suchen schiefgehen kann, ist eine `ClassNotFoundException` möglich.
- ▶ Haben wir bereits ein `Class`-Objekt, sind aber nicht an ihm, sondern an seinen Vorfahren interessiert, so können wir einfach mit `getSuperclass()` ein `Class`-Objekt für die Oberklasse erhalten.

---

<sup>1</sup> Echte Metaklassen wären Klassen, deren jeweils einziges Exemplar die normale Java-Klasse ist. Dann wären etwa die normalen Klassenvariablen in Wahrheit Objektvariablen in der Metaklasse.

<sup>2</sup> Und in der Javadoc heißt es: »Constructor. Only the Java Virtual Machine creates Class objects.«

Das folgende Beispiel zeigt drei Möglichkeiten auf, an ein Class-Objekt für `java.util.Date` heranzukommen:

```
Listing 15.2 src/main/java/com/tutego/insel/meta/GetClassObject.java, main()
Class<Date> c1 = java.util.Date.class;
System.out.println( c1 );           // class java.util.Date
Class<?> c2 = new java.util.Date().getClass();
// oder Class<? extends Date> c2 = ...

System.out.println( c2 );           // class java.util.Date
try {
    Class<?> c3 = Class.forName( "java.util.Date" );
    System.out.println( c3 );           // class java.util.Date
}
catch ( ClassNotFoundException e ) { e.printStackTrace(); }
```

Die Variante mit `forName(String)` ist sinnvoll, wenn der Name der gewünschten Klasse bei der Übersetzung des Programms noch nicht feststand. Sonst ist die vorhergehende Technik eingängiger, und der Compiler kann prüfen, ob es den Typ gibt. Eine volle Qualifizierung ist nötig, `Class.forName("Date")` würde nur nach `Date` in dem Default-Paket suchen – die Rückgabe ist ja keine Sammlung.

### Beispiel

[zB]

Klassenobjekte für primitive Elemente liefert `forName(String)` nicht! Die beiden Anweisungen `Class.forName("boolean")` und `Class.forName(boolean.class.getName())` führen zu einer `ClassNotFoundException`.

`class java.lang.Object`

- `final Class<? extends Object> getClass()`  
Liefert zur Laufzeit das Class-Exemplar, das die Klasse des Objekts repräsentiert.

`final class java.lang.Class<T>`  
`implements Serializable, GenericDeclaration, Type, AnnotatedElement`

- `static Class<?> forName(String className) throws ClassNotFoundException`  
Liefert das Class-Exemplar für die Klasse oder Schnittstelle mit dem angegebenen voll qualifizierten Namen. Falls sie bisher noch nicht vom Programm benötigt wurde, sucht und lädt der Klassenlader die Klasse. Die Methode liefert niemals `null` zurück. Falls die Klasse nicht geladen und eingebunden werden konnte, gibt es eine `ClassNotFoundException`. Eine alternative Methode `forName(String name, boolean initialize, ClassLoader`

loader) ermöglicht auch das Laden mit einem gewünschten Klassenlader. Der Klassenname muss immer voll qualifiziert sein.

### **ClassNotFoundException und NoClassDefFoundError \***

Eine ClassNotFoundException lösen die Methoden

- ▶ `forName(...)` aus `Class` und
- ▶ `loadClass(String name [, boolean resolve])` aus `ClassLoader` bzw.
- ▶ `findSystemClass(String name)` aus `ClassLoader`

immer dann aus, wenn der Klassenlader die Klasse nach ihrem Klassennamen nicht finden kann. Auslöser ist also die Anwendung, die dynamisch Typen laden will, die dann aber nicht vorhanden sind.

Neben der Exception-Klasse gibt es ein NoClassDefFoundError – ein harter `LinkageError`, den die JVM immer dann auslöst, wenn sie eine im Bytecode referenzierte Klasse nicht laden kann. Nehmen wir zum Beispiel eine Anweisung wie `new MeineKlasse()`. Führt die JVM diese Anweisung aus, versucht sie, den Bytecode von `MeineKlasse` zu laden. Ist der Bytecode für `MeineKlasse` nach dem Compilieren entfernt worden, löst die JVM durch den nicht geglückten Ladeversuch den NoClassDefFoundError aus. Auch tritt der Fehler auf, wenn beim Laden des Bytecodes die Klasse `MeineKlasse` zwar gefunden wurde, aber `MeineKlasse` einen statischen Initialisierungsblock besitzt, der wiederum eine Klasse referenziert, für die keine Klasse datei vorhanden ist.

Während `ClassNotFoundException` häufiger vorkommt als `NoClassDefFoundError`, ist die Ausnahme im Allgemeinen ein Indiz dafür, dass ein Java-Archiv im Modulpfad fehlt.

### **Probleme nach Anwendung eines Obfuscators \***

Dass der Compiler automatisch Bytecode gemäß diesem veränderten Quellcode erzeugt, führt nur dann zu unerwarteten Problemen, wenn wir einen Obfuscator über den Programmtext laufen lassen, der nachträglich den Bytecode modifiziert und damit die Bedeutung des Programms bzw. des Bytecodes verschleiert und dabei Typen umbenennt. Offensichtlich darf ein Obfuscator Typen, deren `Class`-Exemplare abgefragt werden, nicht umbenennen; oder der Obfuscator müsste die entsprechenden Zeichenketten ebenfalls korrekt ersetzen (aber natürlich nicht alle Zeichenketten, die zufällig mit Namen von Klassen übereinstimmen).

#### **15.3.2 Eine Class ist ein Type**

In Java gibt es unterschiedliche Typen, wobei Klassen, Schnittstellen und Aufzählungstypen von der JVM als `Class`-Objekte repräsentiert werden. In der Reflection-API repräsentiert die Schnittstelle `Type` alle Typen. Das ist bisher die implementierende Klasse `Class`, und dazu kommen einige Unterschnittstellen:

- ▶ ParameterizedType (repräsentiert generische Typen wie `List<T>`)
- ▶ TypeVariable<D> (repräsentiert beispielsweise `T extends Comparable<? super T>`)
- ▶ WildcardType (repräsentiert etwa `? super T`)
- ▶ GenericArrayType (repräsentiert so etwas wie `T[ ]`)

Die einzige Methode von `Type` ist `get TypeName()`, und das ist sogar »nur« eine Default-Methode, die `toString()` aufruft.

`Type` ist die Rückgabe diverser Methoden in der Reflection-API, etwa von `getGenericSuperclass()` und `getGenericInterfaces()` der Klasse `Class` und von vielen weiteren Methoden, die die Javadoc unter »USE« aufzählt.

## 15.4 Klassenlader

In Java ist eine Kaskade von unterschiedlichen Klassenladern für das Laden von Klassen verantwortlich. Bei Java arbeiten mehrere Klassenlader in einer Kette zusammen.

- ▶ An erster Stelle steht der Klassenlader für alle »Kern«-Klassen, wie `Object`, `String`, der *Bootstrap-Klassenlader*. Findet der eine gewünschte Klasse nicht, geht die Anfrage weiter.
- ▶ *Applikations-Klassenlader* (auch *System-Klassenlader*): Wenn eine Klasse auch dort nicht zu finden ist, folgt die Suche über den benutzerdefinierten Klassenpfad und eigene Klassenlader.

Aus Sicherheitsgründen beginnt der Klassenlader bei einer neuen Klasse immer mit dem Bootstrap-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.

### Abfragen des Klassenpfades

Ob der eigene Klassenpfad überhaupt gesetzt ist, ermittelt ein einfaches `echo %CLASSPATH%` (Windows) bzw. `echo $CLASSPATH` (Unix).

Zur Laufzeit steht der normale Klassenpfad in der System-Property `java.class.path`.

Eigenschaft	Beispielbelegung
<code>java.class.path</code>	<code>C:\Users\Christian\Insel\programme\2_17_Reflection_Annotationen</code>

**Tabelle 15.4** Mögliche Ausgaben von `System.out.println(System.getProperty("java.class.path"))`



### Hinweis

Wird die JVM über `java -jar` aufgerufen, beachtet sie nur Klassen in dem genannten JAR und ignoriert den Klassenpfad.

#### 15.4.1 Die Klasse `java.lang.ClassLoader`

Jeder Klassenlader in Java ist vom Typ `java.lang.ClassLoader`. Die Methode `loadClass(...)` erwartet einen so genannten *binären Namen*, der an den voll qualifizierten 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.
- `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.

## 15.5 Die Utility-Klassen System und Properties

In der Klasse `java.lang.System` finden sich Methoden zum Erfragen und Ändern von Systemvariablen, zum Umlenken der Standarddatenströme, 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` finden sich zusätzlich Hilfsmethoden, wie etwa für 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.

java.lang.System	java.lang.Runtime
<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 15.4 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.

Dass die statische Methode System.arraycopy(..) zum Kopieren von Arrays nicht in java.util.Arrays stationiert ist, lässt sich nur historisch erklären. Und System.exit(int) leitet an Runtime.getRuntime().exit(int) weiter. Einige Methoden sind veraltet und 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 die Brut lebt mit uns für immer.

### 15.5.1 Speicher der JVM

Das Runtime -Objekt hat drei Methoden, die Auskunft über den Speicher der JVM geben:

- ▶ `maxMemory()` liefert die Anzahl Bytes, die maximal für die JVM verfügbar sind. Der Wert kann mit `-Xmx` auf der Kommandozeile beim Aufruf der JVM gesetzt werden.
- ▶ `totalMemory()` ist das, was aktuell genutzt wird, und bis auf `maxMemory()` wachsen kann. Es kann prinzipiell auch wieder schrumpfen. Es gilt: `maxMemory() > totalMemory()`.
- ▶ `freeMemory()` ist das, was frei für neue Objekte ist und die automatische Speicherbereinigung auch wieder anhebt. Es gilt: `totalMemory() > freeMemory()`. Es ist `freeMemory()` nicht der gesamte freie verfügbare Speicherbereich, denn es fehlt noch der »Anteil« von `maxMemory()`.

Zwei Informationen fehlen also, die berechnet werden müssen:

Benutzter Speicher:

```
long usedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
```

Freier Gesamtspeicher:

```
long totalFreeMemory = Runtime.getRuntime().maxMemory() - usedMemory;
```

[zB]

#### Beispiel

Gib Informationen über den Speicher auf einem Rechner aus:

```
long totalMemory      = Runtime.getRuntime().totalMemory();
long freeMemory       = Runtime.getRuntime().freeMemory();
long maxMemory        = Runtime.getRuntime().maxMemory();
long usedMemory       = totalMemory - freeMemory;
long totalFreeMemory = maxMemory - usedMemory;

System.out.printf(
    "total=%d MiB, free=%d MiB, max=%d MiB, used=%d MiB, total free=%d MiB%n",
    totalMemory >> 20, freeMemory >> 20, maxMemory >> 20,
    usedMemory >> 20, totalFreeMemory >> 20 );
```

Die Ausgabe kann sein:

```
total=126 MiB, free=124 MiB, max=2016 MiB, used=1 MiB, total free=2014 MiB
```

### 15.5.2 Anzahl CPUs/Kerne

Die Runtime-Methode `availableProcessors()` liefert die Anzahl logischer Prozessoren bzw. Kerne.

#### Beispiel

```
System.out.println( Runtime.getRuntime().availableProcessors() ); // 4
```

[zB]

### 15.5.3 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.

#### Beispiel

Gib den Namen des Betriebssystems aus:

```
System.out.println( System.getProperty( "os.name" ) ); // z. B. Windows 10
```

Gib alle Systemeigenschaften auf dem Bildschirm aus:

```
System.getProperties().list( System.out );
```

Die Ausgabe beginnt mit:

```
-- listing properties --
sun.desktop=windows
awt.toolkit=sun.awt.windows.WToolkit
java.specification.version=9
file.encoding.pkg=sun.io
sun.cpu.isalist=amd64
...
```

[zB]

Eine Liste der wichtigen Standardsystemeigenschaften:

Schlüssel	Bedeutung
<code>java.version</code>	Version der Java-Laufzeitumgebung
<code>java.class.path</code>	Eigener Klassenpfad

Tabelle 15.5 Standardsystemeigenschaften

Schlüssel	Bedeutung
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 15.5 Standardsystemeigenschaften (Forts.)

### 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` bzw. 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 für `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.

### 15.5.4 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 Weißraum) 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(String[])-Methode auf, da sie vor dem Namen der Klasse stehen und bereits von der Java-Laufzeitumgebung verarbeitet werden.

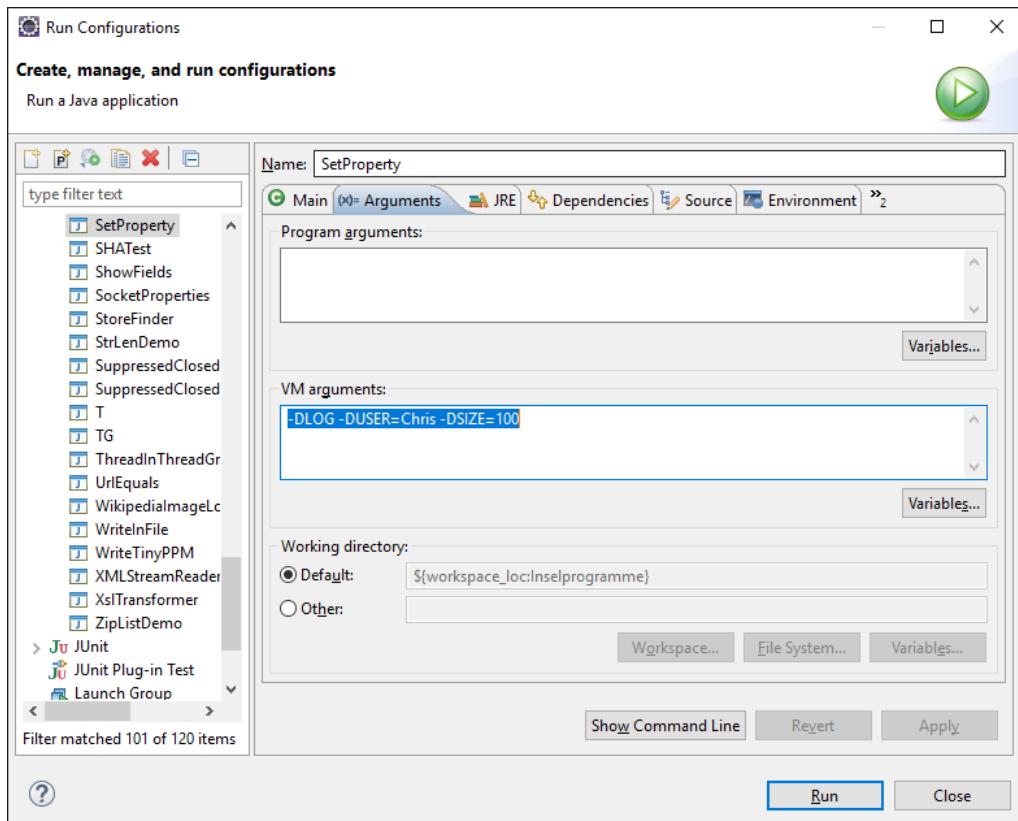
Um die Eigenschaften auszulesen, nutzen wir das bekannte System.getProperty(...):

**Listing 15.3** com/tutego/insel/lang/SetProperty.java, main()

```
Optional<String> logProperty = ofNullable( System.getProperty( "LOG" ) );
Optional<String> usernameProperty = ofNullable( System.getProperty( "USER" ) );
Optional<String> sizeProperty = ofNullable( System.getProperty( "SIZE" ) );

System.out.println( logProperty.isPresent() ); // true
usernameProperty.ifPresent( System.out::println ); // Chris
sizeProperty.map( Integer::parseInt ).ifPresent( System.out::println ); // 100
System.out.println( System.getProperty( "DEBUG", "false" ) ); // false
```

Wir bekommen über getProperty(String) 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 null-Test sagt also, ob logProperty vorhanden ist oder nicht. Statt -DLOG führt auch -DLOG= zum gleichen Ergebnis, denn der assoziierte Wert ist der Leer-String. 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(String, String), der zwei Argumente übergeben werden, denn das zweite steht für einen Default-Wert. So kann immer ein Standardwert angenommen werden.



**Abbildung 15.5** Entwicklungsumgebungen erlauben es, die Kommandozeilenargumente in einem Fenster zu setzen. Unter Eclipse gehen wir dazu unter »Run • Run Configurations«, dann zu »Arguments«.

### Boolean.getBoolean(String)

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(String)` 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` anzutreffen, 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(String)` 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>3</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.

### 15.5.5 Zeilenumbruchzeichen, `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)
- ▶ Macintosh: Carriage Return (Wagenrücklauf)
- ▶ Windows: beide Zeichen (Carriage Return und Line Feed)

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).

In Java gibt es drei Möglichkeiten, an das Zeilenumbruchzeichen bzw. die Zeilenumbruchzeichenfolge des Systems heranzukommen:

1. mit dem Aufruf von `System.getProperty("line.separator")`
2. mit dem Aufruf von `System.lineSeparator()`
3. Nicht immer ist es nötig, das Zeichen (bzw. genau genommen eine mögliche Zeichenfolge) einzeln zu erfragen. Ist das Zeichen Teil einer formatierten Ausgabe beim `Formatter`, `String.format(...)` bzw. `printf(...)`, so steht der Formatspezifizierer `%n` für genau die im System hinterlegte Zeilenumbruchzeichenfolge.

---

<sup>3</sup> Das liegt an der Implementierung: `Boolean.valueOf("false")` liefert genauso `false` wie `Boolean.valueOf("")` oder `Boolean.valueOf(null)`.

### 15.5.6 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.



#### Beispiel

Was ist der Suchpfad? Den liefert `System.getenv("path")`<sup>4</sup>

Name der Variablen	Beschreibung	Beispiel
COMPUTERNAME	Name des Computers	MOE
HOMEDRIVE	Laufwerk des Benutzerverzeichnisses	C:
HOMEPATH	Pfad des Benutzerverzeichnisses	\Users\Christian
OS	Name des Betriebssystems <sup>4</sup>	Windows_NT
PATH	Suchpfad	C:\windows\SYSTEM32; C:\windows ...
PATHEXT	Dateiendungen, die für ausführbare Programme stehen	.COM;.EXE;.BAT;.CMD;.VBS; .VBE;.JS;.JSE;.WSF;.WSH;.MSC
SYSTEMDRIVE	Laufwerk des Betriebssystems	C:
TEMP und auch TMP	temporäres Verzeichnis	C:\Users\CHRIST~1\AppData\Local\Temp
USERDOMAIN	Domäne des Benutzers	MOE

Tabelle 15.6 Auswahl einiger unter Windows verfügbarer Umgebungsvariablen

<sup>4</sup> Das Ergebnis weicht von `System.getProperty("os.name")` ab, was bei Windows 10 schon »Windows 10« liefert.

Name der Variablen	Beschreibung	Beispiel
USERNAME	Name des Nutzers	<i>Christian</i>
USERPROFILE	Profilverzeichnis	<i>C:\Users\Christian</i>
WINDIR	Verzeichnis des Betriebssystems	<i>C:\windows</i>

Tabelle 15.6 Auswahl einiger unter Windows verfügbarer Umgebungsvariablen (Forts.)

Einige der Variablen sind auch über die System-Properties (`System.getProperties()`, `System.getProperty(...)`) erreichbar.

### Beispiel

Gib die Umgebungsvariablen des Systems aus.

```
Map<String, String> map = System.getenv();
map.forEach( (k, v) -> System.out.printf( "%s=%s%n", k, v ) );
```

[zB]

## 15.6 Sprachen der Länder

Beginnen Entwickler mit Ausgaben auf der Konsole oder grafischen Oberfläche, so verdrachten sie oft die Ausgabe fest mit einer Landessprache. Ändert sich die Sprache, kann die Software nicht mit anderen landesüblichen Regeln etwa bei der Formatierung von Fließkomma-zahlen umgehen. Dabei ist es gar nicht schwer, »mehrsprachige« Programme zu entwickeln, die unter verschiedenen Sprachen lokalisierte Ausgaben liefern. Im Grunde müssen wir alle sprachabhängigen Zeichenketten und Formatierungen von Daten durch Code ersetzen, der die landesüblichen Ausgaben und Regeln berücksichtigt. Java bietet hier eine Lösung an: zum einen durch die Definition einer Sprache, die dann Regeln vorgibt, nach denen die Java-API Daten automatisch formatieren kann, und zum anderen durch die Möglichkeit, sprach-abhängige Teile in Ressourcen-Dateien auszulagern.

### 15.6.1 Sprachen in Regionen über Locale-Objekte

In Java repräsentieren Locale-Objekte Sprachen in geografischen, politischen oder kulturellen 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. Locale-

Objekte werden dann zum Beispiel einem Formatter – der hinter `String.format(...)` und `printf(...)` steht – oder einem Scanner übergeben, diese Ausgaben nennen sich englisch *locale-sensitive*.

### Locale-Objekte aufbauen

Locale-Objekte werden immer mit dem Namen der Sprache und optional mit dem Namen des Landes bzw. einer Region und Variante erzeugt. Die Locale-Klasse bietet drei Möglichkeiten zum Aufbau der Objekte:

- ▶ Locale-Konstruktor
- ▶ Die geschachtelte Klasse `Builder` von Locale nutzt das Builder-Pattern zum Aufbau neuer Locale-Objekte.
- ▶ über die Locale-Methode `forLanguageTag(...)` und eine String-Kennung



#### Beispiel

Im Konstruktor der Klasse Locale werden Länderabkürzungen angegeben, etwa für ein Sprachobjekt für Großbritannien oder Frankreich:

```
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>5</sup> (ISO Language Code) identifiziert, und die Ländernamen sind Zwei-Buchstaben-Kürzel, die in ISO 3166<sup>6</sup> (ISO Country Code) beschrieben sind.



#### Beispiel

Drei Varianten zum Aufbau der Locale.JAPANESE:

```
Locale loc1 = new Locale( "ja" );
Locale loc2 = new Locale.Builder().setLanguage( "ja" ).build();
Locale loc3 = Locale.forLanguageTag( "ja" );
```

```
final class java.util.Locale
implements Cloneable, Serializable
```

5 [http://www.loc.gov/standards/iso639-2/php/code\\_list.php](http://www.loc.gov/standards/iso639-2/php/code_list.php)

6 <https://de.wikipedia.org/wiki/ISO-3166-1-Kodierliste>

- `Locale(String language)`  
Erzeugt ein neues Locale-Objekt für die Sprache (`language`), die nach dem ISO-693-Standard gegeben ist. Ungültige Kennungen werden nicht erkannt.
- `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.
- `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.setDefault(Locale)` diese ändern.

Die `Locale`-Klasse hat weitere Methoden; Entwickler sollten für den `Builder`, für `forLanguage-Tag(...)` und die neuen Erweiterungen und Filtermethoden die Javadoc studieren.<sup>7</sup>

### Konstanten für einige Sprachen

Die `Locale`-Klasse besitzt Konstanten für häufig auftretende Sprachen optional mit Ländern. 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. Hinter einer Abkürzung wie `Locale.UK` steht nichts anderes als die Initialisierung mit `new Locale("en", "GB")`.

### Methoden, die Locale-Exemplare annehmen

`Locale`-Objekte sind als Objekte eigentlich uninteressant – sie haben Methoden, doch spannender ist der Typ als Identifikation für eine Sprache. In der Java-Bibliothek gibt es Dutzende von Methoden, die `Locale`-Objekte annehmen und anhand deren ihr Verhalten anpassen. So zum Beispiel `printf(Locale, ...)`, `format(Locale, ...)`, `toLowerCase(Locale)`.

#### Tipp

Gibt es keine Variante einer Formatierungs-/Parse-Methode mit einem `Locale`-Objekt, so unterstützt die Methode in der Regel kein sprachabhängiges Verhalten; das Gleiche gilt für Objekte, die kein `Locale` über einen Konstruktor/Setter annehmen. `Double.toString(..)` ist so ein Beispiel, auch `Double.parseDouble(..)`. In internationalisierten Anwendungen werden diese Methoden selten zu finden sein. Auch eine String-Konkatenation mit z. B. einer Fließkommazahl ist nicht erlaubt – sie ruft intern eine `Double`-Methode auf –, und ein `String.format(..)` ist allemal besser.



<sup>7</sup> Auf Englisch beschreibt das Java-Tutorial von Oracle die Erweiterungen unter <http://docs.oracle.com/javase/tutorial/i18n/locale/index.html>.

## Methoden von Locale \*

Locale-Objekte bieten eine Reihe von Methoden an, die etwa den ISO-639-Code des Landes preisgeben.



### Beispiel

Gib für Sprachen in ausgewählten Ländern zugängliche Locale-Informationen aus. Das Objekt System.out und Locale.\* sind statisch importiert:

**Listing 15.4** src/main/java/com/tutego/insel/locale/GermanyLocal.java, main()

```
out.println( GERMANY.getCountry() );           // DE
out.println( GERMANY.getLanguage() );          // de
out.println( GERMANY.getVariant() );           //
out.println( GERMANY.getISO3Country() );        // DEU
out.println( GERMANY.getISO3Language() );        // deu
out.println( CANADA.getDisplayCountry() );       // Kanada
out.println( GERMANY.getDisplayLanguage() );     // Deutsch
out.println( GERMANY.getDisplayName() );         // Deutsch (Deutschland)
out.println( CANADA.getDisplayName() );          // Englisch (Kanada)
out.println( GERMANY.getDisplayName(CANADA) );   // Deutsch (Deutschland)
out.println( CANADA.getDisplayName(FRENCH) );     // anglais (Canada)
out.println( GERMANY.getDisplayVariant() );      //
```

Es gibt auch statische Methoden zum Erfragen von Locale-Objekten:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- static Locale getDefault()  
Liefert die von der JVM voreingestellte Sprache, die standardmäßig vom Betriebssystem stammt.
- static Locale[] getAvailableLocales()  
Liefert eine Aufzählung aller installierten Locale-Objekte. Das Feld enthält mindestens Locale.US und ca. 160 Einträge.
- static String[] getISOCountries()  
Liefert ein Array mit allen aus zwei Buchstaben bestehenden ISO-3166-Country-Codes.
- static Set<String> getISOCountries(Locale.IsoCountryCode type)  
Liefert eine Menge mit allen ISO-3166-Country-Codes, wobei die Aufzählung IsoCountry-Code bestimmt: PART1\_ALPHA2 liefert den Code aus zwei Buchstaben, PART1\_ALPHA3 aus drei Buchstaben, PART3 aus vier Buchstaben.

Auf der anderen Seite haben wir Methoden, die die Kürzel nach den ISO-Normen liefern:

```
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 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.
- `String getVariant()`  
Liefert das Kürzel der Variante oder einen leeren String.
- Die genannten Methoden liefern zwar Kürzel, aber das ist nicht gedacht als menschenlesbare Ausgabe. Für diverse `getXXX()`-Methoden gibt es entsprechende `getDisplayXXX()`-Methoden:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- `String getDisplayCountry(Locale inLocale)`  
`final String getDisplayCountry()`  
Liefert den Namen des Landes für Bildschirmausgaben für eine Sprache oder `Locale.getDefault()`.
- `String getDisplayLanguage(Locale inLocale)`  
`String getDisplayLanguage()`  
Liefert den Namen der Sprache für Bildschirmausgaben für eine Sprache oder `Locale.getDefault()`.
- `String getDisplayName(Locale inLocale)`  
`final String getDisplayName()`  
Liefert den Namen der Einstellungen für eine Sprache oder `Locale.getDefault()`.
- `String getDisplayVariant(Locale inLocale)`  
`final String getDisplayVariant()`  
Liefert den Namen der Variante für eine Sprache oder `Locale.getDefault()`.

## 15.7 Wichtige Datum-Klassen im Überblick

Weil Datumsberechnungen verschlungene Gebilde sind, können wir den Entwicklern von Java dankbar sein, dass sie uns viele Klassen zur Datumsberechnung und -formatierung beigelegt haben. Die Entwickler hielten die Klassen so abstrakt, dass lokale Besonderheiten wie Ausgabeformatierung, Parsen, Zeitzonen, Sommer- und Winterzeit unter verschiedenen Kalendern möglich sind.

Bis zur Java-Version 1.1 stand zur Darstellung und Manipulation von Datumswerten ausschließlich die Klasse `java.util.Date` zur Verfügung. Diese hatte mehrere Aufgaben:

- ▶ Erzeugung eines Datum/Zeit-Objekts aus Jahr, Monat, Tag, Minute und Sekunde
- ▶ Abfrage von Tag, Monat, Jahr ... mit der Genauigkeit von Millisekunden
- ▶ Ausgabe und Verarbeitung von Datum-Zeichenketten

Da die `Date`-Klasse nicht ganz fehlerfrei und internationalisiert war, wurden im JDK 1.1 neue Klassen eingeführt:

- ▶ `Calendar` nimmt sich der Aufgabe von `Date` an, zwischen verschiedenen Datumsrepräsentationen und Zeitskalen zu konvertieren. Die Unterklasse `GregorianCalendar` wird direkt erzeugt.
- ▶ `DateFormat` zerlegt Datum-Zeichenketten und formatiert die Ausgabe. Auch Datumsformate sind vom Land abhängig, das Java durch `Locale`-Objekte darstellt, und von einer Zeitzone, die durch die Exemplare der Klasse `TimeZone` repräsentiert ist.

In Java 8 zog eine weitere Datumsbibliothek mit ganz neuen Typen ein. Endlich können auch Datum und Zeit getrennt repräsentiert werden:

- ▶ `LocalDate`, `LocalTime`, `LocalDateTime` sind die temporalen Klassen für ein Datum, für eine Zeit und für eine Kombination aus Datum und Zeit.
- ▶ `Period` und `Duration` stehen für Abstände.

### 15.7.1 Der 1.1.1970

Der 1.1.1970 war ein Donnerstag mit wegweisenden Änderungen. In der katholischen Kirche wurde der Allgemeine Römische Kalender eingeführt,<sup>8</sup> und die Briten freuten sich, dass die Volljährigkeit von 24 Jahren auf 18 Jahre fiel. Zu etwas Technischem: Der 1.1.1970, 0:00:00 UTC heißt auch *Unix Epoche*, und eine *Unixzeit* wird relativ zu diesem Zeitpunkt in Sekunden beschrieben. So kommen wir 100.000.000 Sekunden nach dem 1.1.1970 beim 3. März 1973 um 09:46:40 aus. Das *Unix Millennium* wurde bei 1.000.000.000 Sekunden nach dem 1.1.1970 gefeiert und repräsentiert den 9. September 2001, 01:46:40.

---

<sup>8</sup> Cäsar und der julianische Kalender liegen noch weiter zurück.

### 15.7.2 System.currentTimeMillis()

Auch für uns Java-Entwickler ist die Unixzeit von Bedeutung, denn viele Zeiten in Java sind relativ zu diesem Datum. Der Zeitstempel 0 bezieht sich auf den 1.1.1970 0:00:00 Uhr Greenwich-Zeit – das entspricht 1 Uhr nachts deutscher Zeit. Die Methode `System.currentTimeMillis()` liefert die vergangenen Millisekunden – nicht Sekunden! – relativ zum 1.1.1970, 00:00 Uhr UTC, wobei allerdings die Uhr des Betriebssystems nicht so genau gehen muss. Die Anzahl der Millisekunden wird in einem `long` repräsentiert, also in 64 Bit. Das reicht für etwa 300 Millionen Jahre.

#### Warnung

Die Werte von `currentTimeMillis()` sind nicht zwingend aufsteigend, da sich Java die Zeit vom Betriebssystem holt, und da kann sich die Systemzeit ändern. Der Benutzer kann die Zeit anpassen, oder ein Dienst wie das Network Time Protocol (NTP). Differenzen von `currentTimeMillis()`-Zeitstempeln sind dann komplett falsch und könnten sogar negativ sein. Eine Alternative ist `nanoTime()`, das keinen Bezugspunkt hat, genauer und immer aufsteigend ist.<sup>9</sup>



### 15.7.3 Einfache Zeitumrechnungen durch TimeUnit

Eine Zeitdauer wird in Java oft durch Millisekunden ausgedrückt. 1.000 Millisekunden entsprechen 1 Sekunde,  $1.000 \times 60$  Millisekunden 1 Minute usw. Diese ganzen großen Zahlen sind jedoch nicht besonders anschaulich, sodass zur Umrechnung `TimeUnit`-Objekte mit ihren `toXXX(...)`-Methoden genutzt werden. Java deklariert folgende Konstanten in `TimeUnit`: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `DAYS`, `HOURS`, `SECONDS`, `MINUTES`.

Jedes der Aufzählungselemente definiert die Umrechnungsmethoden `toDays(...)`, `toHours(...)`, `toMicros(...)`, `toMillis(...)`, `toMinutes(...)`, `toNanos(...)`, `toSeconds(...)`; sie bekommen ein `long` und liefern ein `long` in der entsprechenden Einheit. Zudem gibt es zwei `convert(...)`-Methoden, die von einer Einheit in eine andere umrechnet.



#### Beispiel

Konvertiere 23.746.387 Millisekunden in Stunden:

```
int v = 23_746_387;
System.out.println( TimeUnit.MILLISECONDS.toHours( v ) ); // 6
System.out.println( TimeUnit.HOURS.convert( v, TimeUnit.MILLISECONDS ) ); // 6
```

<sup>9</sup> Die Seite <https://stackoverflow.com/questions/351565/system-currenttimemillis-vs-system-nanotime> geht auf Details ein und verlinkt auf interne Implementierungen.

```
enum java.util.concurrent.TimeUnit
    extends Enum<TimeUnit>
    implements Serializable, Comparable<TimeUnit>
```

- NANoseconds, MICROseconds, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS  
Aufzählungselemente von TimeUnit
- long toDays(long duration)
- long toHours(long duration)
- long toMicros(long duration)
- long toMillis(long duration)
- long toMinutes(long duration)
- long toNanos(long duration)
- long toSeconds(long duration)
- long convert(long sourceDuration, TimeUnit sourceUnit)  
Liefert sourceUnit.toXXX(sourceDuration), wobei XXX für die jeweilige Einheit steht. Beispielsweise liefert es HOURS.convert(sourceDuration, sourceUnit), dann sourceUnit.toHours(1). Die Lesbarkeit der Methode ist nicht optimal, daher sollten die anderen Methoden bevorzugt werden. Ergebnisse werden unter Umständen abgeschnitten, nicht gerundet. Gibt es einen Überlauf folgt eine keine ArithmeticException.
- long convert(Duration duration)  
Konvertiert die übergebene duration in die Zeiteinheit, die das aktuelle TimeUnit repräsentiert. So liefert TimeUnit.MINUTES.convert(Duration.ofHours(12)) zum Beispiel 720. Damit sind etwa aunit.convert(Duration.ofNanos(n)) und aunit.convert(n, NANoseconds) gleich. Neu in Java 11.

## 15.8 Date-Time-API

Seit Java 8 gibt es das Paket `java.time`, das alle bisherigen Java-Typen rund um Datum- und Zeitverarbeitung überflüssig macht. Mit anderen Worten: Mit den neuen Typen lassen sich `Date`, `Calendar`, `GregorianCalendar`, `TimeZone` usw. streichen und ersetzen. Natürlich gibt es Adapter zwischen den APIs, doch gibt es nur noch sehr wenige zwingende Gründe, heute bei neuen Programmen auf die älteren Typen zurückzugreifen – ein Grund ist natürlich die heilige Kompatibilität.

Die neue API basiert auf dem standardisierten Kalendersystem von ISO-8601, und das deckt ab, wie ein Datum, wie Zeit, Datum und Zeit, UTC, Zeitintervalle (Dauer/Zeitspanne) und Zeitzonen repräsentiert werden. Die Implementierung basiert auf dem gregorianischen Kalender, wobei auch andere Kalendertypen denkbar sind. Javas Kalendersystem greift auf andere Standards bzw. Implementierungen zurück, unter anderem auf das Unicode Common Loca-

le Data Repository (CLDR) zur Lokalisierung von Wochentagen oder die Time-Zone Database (TZDB), die alle Zeitzonenwechsel seit 1970 dokumentiert. In Java nutzen die XML-APIs schon länger ISO-8601-Kalender, denn Schema-Dateien nutzen einen XMLGregorianCalendar, und selbst für Dauern gibt es einen eigenen Typ, Duration.

### Geschichte

Über die alten Datumsklassen meckert die Java-Community seit über zehn Jahren; nicht ganz zu Unrecht, da ein Date zum Beispiel einen Datums- sowie einen Zeitanteil hat, Kalender fehlen, die Sommerzeitumstellung verschiedener Länder nicht korrekt behandelt wird und wegen weiterer Schwächen.<sup>10</sup> Daher geht die Entwicklung der Date-Time-API lange zurück und basiert auf Ideen von Joda-Time (<http://joda-time.sourceforge.net/>), einer populären quellöffentlichen Bibliothek. Spezifiziert im JSR 310 (eingereicht am 30. Jan 2007)<sup>11</sup> und angedacht für Java 7 (das vier Jahre später, im Juli 2011, kam), wurde die API erst in Java 8 Teil der Java SE. Für Java 1.7 gibt es einen Back-Port (<https://github.com/ThreeTen/threetenbp>), um später leicht die Codebasis auf Java 8 zu migrieren, der durchaus interessant ist.

### Erster Überblick

Die zentralen temporalen Typen aus der Date-Time-API sind schnell dokumentiert:

Typ	Beschreibung	Feld(er)
LocalDate	Repräsentiert ein übliches Datum.	Jahr, Monat, Tag
LocalTime	Repräsentiert eine übliche Zeit.	Stunden, Minuten, Sekunden, Nanosekunden
LocalDateTime	Kombination aus Datum und Zeit	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden
Period	Dauer zwischen zwei LocalDates	Jahr, Monat, Tag
Year	Nur Jahr	Jahr
Month	Nur Monat	Monat
MonthDay	Nur Monat und Tag	Monat, Tag
OffsetTime	Zeit mit Zeitzone	Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Offset

Tabelle 15.7 Alle temporalen Klassen aus java.time

<sup>10</sup> Der Quellcode stammt von IBM. Trifft Oracle jetzt die Schuld, weil Sun die Implementierung damals übernahm? Siehe zu den Kritiken auch <http://tutego.de/go/dategotchas>.

<sup>11</sup> <https://jcp.org/en/jsr/detail?id=310>

Typ	Beschreibung	Feld(er)
OffsetDateTime	Datum und Zeit mit Zeitzone als UTC-Offset	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Offset
ZonedDateTime	Datum und Zeit mit Zeitzone als ID und Offset	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Info
Instant	Zeitpunkt (fortlaufende Maschinenzeit)	Nanosekunden
Duration	Zeitintervall zwischen zwei Instants	Sekunden/Nanosekunden

Tabelle 15.7 Alle temporalen Klassen aus java.time (Forts.)

### 15.8.1 Menschenzeit und Maschinenzeit

Datum und Zeit, die wir als Menschen in Einheiten wie Tagen und Minuten verstehen, nennen wir *Menschenzeit* (engl. *human time*), die fortlaufende Zeit des Computers, die eine Auflösung im Nanosekundenbereich hat, *Maschinenzeit*. Die Maschinenzeit startet dabei von einer Zeit, die wir *Epoche* nennen, z. B. die Unix-Epoche.

Aus Sehen Kinder alles? Die Sichtbarkeit protected lässt sich gut ablesen, dass die meisten Klassen für uns Menschen gemacht sind und sich nur Instant/Duration auf die Maschinenzeit bezieht. LocalDate, LocalTime und LocalDateTime repräsentieren Menschenzeit ohne Bezug zu einer Zeitzone, ZonedDateTime mit Zeitzone. Bei der Auswahl der richtigen Zeitklassen für eine Aufgabenstellung ist das natürlich die erste Überlegung, ob die Menschenzeit oder die Maschinenzeit repräsentiert werden soll, dann folgt, was genau für Felder nötig sind und ob eine Zeitzone relevant ist oder nicht. Soll zum Beispiel die Ausführungszeit gemessen werden, ist es unnötig, zu wissen, an welchem Datum die Messung begann und endet; hier ist Duration korrekt, nicht Period.



#### Beispiel

```
LocalDate now = LocalDate.now();
System.out.println( now ); // 2018-03-09
System.out.printf( "%d. %s %d%n", now.getDayOfMonth(), now.getMonth(), now.getYear() );
// 9. MARCH 2018
LocalDate bdayMLKing = LocalDate.of( 1929, Month.JANUARY, 15 );
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate( TextStyle.MEDIUM );
System.out.println( bdayMLKing.format( formatter ) ); // 15. Januar 1929
```

Die Methode `getMonth()` auf einem `LocalDate` liefert als Ergebnis ein `java.time.Month`-Objekt, und das sind Aufzählungen; die `toString()`-Repräsentation liefert die Konstante in Großbuchstaben.

Alle Klassen basieren standardmäßig auf dem ISO-System, andere Kalendersysteme, wie der japanische Kalender, werden über Typen aus `java.time.chrono` erzeugt, natürlich sind auch ganz neue Systeme möglich.

### Beispiel

```
ChronoLocalDate now = JapaneseChronology.INSTANCE.dateNow();
System.out.println( now );                                // Japanese Heisei 19-10-23
```

[zB]

## Paketübersicht

Die Typen der Date-Time-API verteilen sich auf verschiedene Pakete:

- ▶ `java.time`: Enthält die Standardklassen wie `LocalTime` und `Instant`. Alle Typen basieren auf dem Kalendersystem ISO-8601, das landläufig unter »gregorianischer Kalender« bekannt ist. Der wird erweitert zum so genannten Proleptic Gregorian Kalender, das ist ein gregorianischer Kalender, der auch für die Zeit vor 1582 (der Einführung des Kalenders) gültig ist, damit eine konsistente Zeitlinie entsteht.
- ▶ `java.time.chrono`: Hier befinden sich vorgefertigte alternative (also Nicht-ISO-)Kalendersysteme, wie japanischer Kalender, Thai-Buddhist-Kalender, islamischer Kalender (genannt Hijrah) und ein paar weitere.
- ▶ `java.time.format`: Klassen zum Formatieren und Parsen von Datum- und Zeit, wie der genannte `DateTimeFormatter`
- ▶ `java.time.zone`: unterstützende Klassen für Zeitzonen, etwa `ZonedDateTime`
- ▶ `java.time.temporal`: tiefer liegende API, die Zugriff und Modifikation einzelner Felder eines Datums/Zeitwerts erlaubt

## Designprinzipien

Bevor wir uns mit den einzelnen Klassen auseinandersetzen, wollen wir uns mit den Designprinzipien beschäftigen, denn alle Typen der Date-Time-API folgen wiederkehrenden Mustern. Die erste und wichtigste Eigenschaft ist, dass alle Objekte *immutable* sind, also nicht veränderbar. Das ist bei der »alten« API anders: `Date` und die `Calendar`-Klassen sind veränderbar, mit teils verheerenden Folgen; denn werden diese Objekte rumgereicht und verändert, kann es zu unkalkulierbaren Seiteneffekten kommen. Die Klassen der neuen Date-Time-API sind *immutable*, und so stehen die Datums-/Zeit-Klassen wie `LocalTime` oder `Instant` den

veränderbaren Typen wie `Date` oder `Calendar` gegenüber. Alle Methoden, die nach Änderung aussehen, erzeugen neue Objekte mit den gewünschten Änderungen. Seiteneffekte bleiben also aus, und alle Typen sind threadsicher.

Unveränderbarkeit ist eine Designeigenschaft wie auch die Tatsache, dass `null` nicht als Argument erlaubt wird. In der Java-API wird oftmals `null` akzeptiert, weil es etwas Optionales ausdrückt, doch die Date-Time-API strafft dies in der Regel mit einer `NullPointerException`. Dass `null` nicht als Argument und nicht als Rückgabe im Einsatz ist, kommt einer weiteren Eigenschaft zugute: Die API gestattet »flüssige« Ausdrücke, also kaskadierte Aufrufe, da viele Methoden die `this`-Referenz zurückgeben, so wie das auch von `StringBuilder` bekannt ist.

Zu diesen eher technischen Eigenschaften kommt die konsistente Namensgebung hinzu, die sich von der Namensgebung der bekannten JavaBeans absetzt. So gibt es keine Konstruktoren und keine Setter (das brauchen die immutablen Klassen nicht), sondern Muster, die viele der Typen aus der Date-Time-API einhalten:

Methode	Klassen-/Exemplarmethode	Grundsätzliche Bedeutung
<code>now()</code>	Statisch	Liefert ein Objekt mit aktueller Zeit/aktuuellem Datum.
<code>ofXXX()</code>	Statisch	Erzeugt neue Objekte.
<code>fromXXX()</code>	Statisch	Erzeugt neue Objekte aus anderen Repräsentationen.
<code>parseXXX()</code>	Statisch	Erzeugt neues Objekt aus einer String- Repräsentation.
<code>format()</code>	Exemplar	Formatiert und liefert einen String.
<code>getXXX()</code>	Exemplar	Liefert Felder eines Objekts.
<code>isXXX()</code>	Exemplar	Fragt Status eines Objekts ab.
<code>withXXX()</code>	Exemplar	Liefert eine Kopie des Objekts mit einer geänderten Eigenschaft.
<code>plusXXX()</code>	Exemplar	Liefert eine Kopie des Objekts mit einer aufsummierten Eigenschaft.
<code>minusXXX()</code>	Exemplar	Liefert eine Kopie des Objekts mit einer reduzierten Eigenschaft.
<code>toXXX()</code>	Exemplar	Konvertiert ein Objekt in einen neuen Typ.

Tabelle 15.8 Namensmuster in der Date-Time-API

Methode	Klassen-/Exemplarmethode	Grundsätzliche Bedeutung
atXXX()	Exemplar	Kombiniert dieses Objekt mit einem anderen Objekt.
XXXInto()	Exemplar	Kombiniert eigenes Objekt mit einem anderen Zielobjekt.

Tabelle 15.8 Namensmuster in der Date-Time-API (Forts.)

Die Methode `now()` haben wir schon in den ersten Beispielen verwendet, sie liefert zum Beispiel das aktuelle Datum. Weitere Erzeugermethoden sind die mit dem Präfix `of`, `from` oder `with`; Konstruktoren gibt es nicht. `withXXX()`-Methoden nehmen die Rolle der Setter ein.

### 15.8.2 Datumsklasse LocalDate

Ein Datum (ohne Zeitzone) repräsentiert die Klasse `LocalDate`. Damit lässt sich zum Beispiel ein Geburtsdatum repräsentieren.

Ein temporales Objekt kann über die statischen `of(...)`-Fabrikmethoden aufgebaut, über `ofInstant(Instant instant, ZoneId zone)` oder von einem anderen temporalen Objekt abgeleitet werden. Interessant sind die Methoden, die mit einem `TemporalAdjuster` arbeiten.

#### Beispiel

```
LocalDate today = LocalDate.now();
LocalDate nextMonday = today.with( TemporalAdjusters.next( DayOfWeek.SATURDAY ) );
System.out.printf( "Heute ist der %s, und frei ist am Samstag, den %s",
                  today, nextMonday );
```

[zB]

Mit den Objekten in der Hand können wir diverse Getter nutzen und einzelne Felder erfragen, etwa `getDayOfMonth()`, `getDayOfYear()` (liefern `int`) oder `getDayOfWeek()`, das eine Aufzählung vom Typ `DayOfWeek` liefert, und `getMonth()`, das eine Aufzählung vom Typ `Month` liefert. Weiterhin gibt es `long toEpochDay()` und `long toEpochSecond(LocalTime time, ZoneOffset offset)`.

Dazu kommen Methoden, die mit `minusXXX(..)` oder `plusXXX(..)` neue `LocalDate`-Objekte liefern, wenn zum Beispiel mit `minusYear(long yearsToSubtract)` eine Anzahl Jahre zurückgefahren werden soll. Durch die Negation des Vorzeichens kann auch die jeweils entgegengesetzte Methode genutzt werden, sprich, `LocalDate.now().minusMonths(1)` kommt zum gleichen Ergebnis wie `LocalDate.now().plusMonths(-1)`. Die `withXXX(..)`-Methoden belegen ein Feld neu und liefern ein modifiziertes neues `LocalDate`-Objekt.

Von einem `LocaleDate` lassen sich andere temporale Objekte bilden, `atTime(...)` etwa liefert `LocalDateTime`-Objekte, bei denen gewisse Zeitfelder belegt sind. `atTime(int hour, int minute)` ist so ein Beispiel. Mit `until(...)` lässt sich eine Zeitdauer vom Typ `Period` liefern. Interessant sind zwei Methoden, die einen Strom von `LocalDate`-Objekten bis zu einem Endpunkt liefern:

- ▶ `Stream<LocalDate> datesUntil( LocalDate endExclusive )`
- ▶ `Stream<LocalDate> datesUntil( LocalDate endExclusive, Period step )`

## 15.9 Logging mit Java

Das Loggen (Protokollieren) von Informationen über Programmzustände ist ein wichtiger Teil, um später den Ablauf und die Zustände von Programmen rekonstruieren und verstehen zu können. Mit einer Logging-API lassen sich Meldungen auf die Konsole oder in externe Speicher wie Text-/XML-Dateien, Datenbanken schreiben oder über einen Chat verbreiten.

### 15.9.1 Logging-APIs

Bei den Logging-Bibliotheken und APIs ist die Java-Welt leider gespalten. Da die Java-Standardsbibliothek in den ersten Versionen keine Logging-API anbot, füllte die Open-Source-Bibliothek *log4j* schnell diese Lücke. Sie wird heute in nahezu jedem größeren Java-Projekt eingesetzt. Als in Java 1.4 die Logging-API (JSR 47) einzog, war die Java-Gemeinde erstaunt, dass `java.util.logging` (JUL) weder API-kompatibel mit dem beliebten `log4j` noch so leistungsfähig wie `log4j` ist.<sup>12</sup>

Im Laufe der Jahre veränderte sich das Bild. Während in der Anfangszeit Entwickler ausschließlich auf `log4j` bauten, werden es langsam mehr Projekte mit der JUL. Ein erster Grund ist der, dass einige Entwickler externe Abhängigkeiten vermeiden wollen (wobei das nicht wirklich funktioniert, da nahezu jede eingebundene Java-Bibliothek selbst auf `log4j` baut), und der zweite, dass für viele Projekte JUL einfach reicht. In der Praxis bedeutet dies für größere Projekte, dass mehrere Logging-Konfigurationen das eigene Programm verschmutzen, da jede Logging-Implementierung unterschiedlich konfiguriert wird.

---

<sup>12</sup> Oracles Logging-API ist dagegen nur ein laues Lüftchen, das nur Grundlegendes wie hierarchische Logger bietet. An die Leistungsfähigkeit von `log4j` mit einer großen Anzahl von Schreibern in Dateien (File-Appender, RollingFileAppender, DailyRollingFileAppender), Syslog und NT-Logger (SyslogAppender, NTEventLogAppender), Datenbanken, Versand über das Netzwerk (JMSAppender, SMTPAppender, SocketAppender, TelnetAppender) kommt das Standard-Logging nicht heran.

### 15.9.2 Logging mit java.util.logging

Mit der Java-Logging-API lässt sich eine Meldung schreiben, die sich dann zur Wartung oder zur Sicherheitskontrolle einsetzen lässt. Die API ist einfach:

**Listing 15.5** src/main/java/com/tutego/insel/logging/CULDemo.java, JULDemo

```
package com.tutego.insel.logging;

import static java.time.temporal.ChronoUnit.MILLIS;
import static java.time.Instant.now;
import java.time.Instant;
import java.util.logging.Level;
import java.util.logging.Logger;

public class JULDemo {

    private static final Logger log = Logger.getLogger( JULDemo.class.getName() );

    public static void main( String[] args ) {
        Instant start = now();
        log.info( "Wir starten mit JUL" );

        try {
            log.log( Level.INFO, "In {0} Sekunden geht es los", 0 );
            throw null; // Fehler erzeugen
        }
        catch ( Exception e ) {
            log.log( Level.SEVERE, "Oh Oh", e );
        }

        log.info( () -> String.format( "Laufzeit %s ms", start.until( now(), MILLIS ) ) );
    }
}
```

Lassen wir das Beispiel laufen, folgt auf der Konsole die Warnung:

```
Mai 21, 2017 10:24:57 NACHM. com.tutego.insel.logging.JULDemo main
INFORMATION: Wir starten mit JUL
Mai 21, 2017 10:24:57 NACHM. com.tutego.insel.logging.JULDemo main
INFORMATION: In 0 Sekunden geht es los
Mai 21, 2017 10:24:57 NACHM. com.tutego.insel.logging.JULDemo main
SCHWERWIEGEND: Oh Oh
```

```
java.lang.NullPointerException
at com.tutego.insel.logging.JULDemo.main(JULDemo.java:19)
```

Mai 21, 2017 10:24:57 NACHM. com.tutego.insel.logging.JULDemo main  
INFORMATION: Laufzeit 131 ms

### Das Logger-Objekt

Zentral ist das Logger-Objekt, das über `Logger.getAnonymousLogger()` oder über `Logger.getLogger(String name)` geholt werden kann, wobei `name` in der Regel mit dem voll qualifizierten Klassennamen belegt ist. Oft ist der Logger als private statische finale Variable in der Klasse deklariert.

### Loggen mit Log-Level

Nicht jede Meldung ist gleich wichtig. Einige sind für das Debuggen oder wegen der Zeitmessungen hilfreich, doch Ausnahmen in den catch-Zweigen sind enorm wichtig. Damit verschiedene Detailgrade unterstützt werden, lässt sich ein *Log-Level* festlegen. Er bestimmt, wie »ernst« der Fehler bzw. eine Meldung ist. Das ist später wichtig, wenn die Fehler nach ihrer Dringlichkeit aussortiert werden. Die Log-Level sind in der Klasse `Level` als Konstanten deklariert.<sup>13</sup>

- ▶ FINEST (kleinste Stufe)
- ▶ FINER
- ▶ FINE
- ▶ CONFIG
- ▶ INFO
- ▶ WARNING
- ▶ SEVERE (höchste Stufe)

Zum Loggen selbst bietet die `Logger`-Klasse die allgemeine Methode `log(Level level, String msg)` bzw. für jeden Level eine eigene Methode:

Level	Aufruf über log(...)	Spezielle Log-Methode
SEVERE	<code>log(Level.SEVERE, msg)</code>	<code>severe(String msg)</code>
WARNING	<code>log(Level.WARNING, msg)</code>	<code>warning(String msg)</code>
INFO	<code>log(Level.INFO, msg)</code>	<code>info(String msg)</code>

Tabelle 15.9 Log-Level und Methoden

<sup>13</sup> Da das Logging-Framework in Version 1.4 zu Java stieß, nutzt es noch keine typisierten Aufzählungen, denn die gibt es erst seit Java 5.

Level	Aufruf über log(...)	Spezielle Log-Methode
CONFIG	log(Level.CONFIG, msg)	config(String msg)
FINE	log(Level.FINE, msg)	fine(String msg)
FINER	log(Level.FINER, msg)	finer(String msg)
FINEST	log(Level.FINEST, msg)	finest(String msg)

Tabelle 15.9 Log-Level und Methoden (Forts.)

Alle diese Methoden setzen eine Mitteilung vom Typ String ab. Sollen eine Ausnahme und der dazugehörige Stack-Trace geloggt werden, müssen Entwickler zu folgender Logger-Methode greifen, die auch schon das Beispiel nutzt:

- void log(Level level, String msg, Throwable thrown)

Die Varianten von severe(...), warning(...) usw. sind nicht überladen mit einem Parameter vom Typ Throwable.

## 15.10 Maven: Build-Management und Abhängigkeiten auflösen

Software zu bauen und die Abhängigkeiten der Module zu überwachen ist eine Tätigkeit, die nicht manuell, sondern automatisch passieren sollte. Die freie und quelloffene Software *Apache Maven* hat sich hier als Quasistandard durchgesetzt. Maven lässt sich zum einen von der Website <https://maven.apache.org/> beziehen und als Werkzeug von der Kommandozeile nutzen oder schön integriert über die Entwicklungsumgebung; alle wichtigen IDEs unterstützen Maven von Haus aus.

Um Projekte zu beschreiben, nutzt Maven *POM*-Dateien (das steht für *Project Object Model*). Sie enthalten die gesamte Konfiguration und Projektbeschreibung. Dazu zählen unter anderem:

- ▶ Name und Kennung des Projekts
- ▶ Abhängigkeiten
- ▶ Compilereinstellungen
- ▶ Lizenz

Eine POM-Datei heißt in der Regel *pom.xml* und liegt im Hauptverzeichnis einer Anwendung – die Dateiendung signalisiert, dass es sich um eine XML-Datei handelt. Ein Java-Compiler und die Werkzeuge müssen mit Maven nicht mehr von Hand aufgerufen werden, die ganze Steuerung liegt abschließend bei Maven.

### 15.10.1 Beispielprojekt in Eclipse mit Maven

Statt in Eclipse mit FILE • NEW • JAVA PROJECT ein Projekt aufzubauen, ist der Schritt mit Maven ein anderer. Das liegt unter anderem daran, dass Maven eine eigene Verzeichnisstruktur definiert, die *Standard Directory Layout* genannt wird. Sie unterscheidet sich von der Standardstruktur von Eclipse, die nur ein einfaches *src*- und *bin*-Verzeichnis vorsieht. Mit Maven ist das feiner untergliedert für Hauptklasse, Testfälle, Ressourcen usw.

In Eclipse öffnet FILE • NEW • OTHER einen Dialog, in dem unter MAVEN der Punkt MAVEN PROJECT erscheint. Aktivieren wir CREATE A SIMPLE PROJECT und navigieren wir zum nächsten Dialog, lassen sich die zentralen Projekteigenschaften einstellen, die sich *Koordinaten* nennen. Wir wählen für ein Beispiel:

- ▶ GROUP ID: com.tutego.webapp. Gruppierungsbezeichnung vergleichbar mit dem Paketnamen. Repräsentiert das Unternehmen.
- ▶ ARTIFACT ID: tutego-webapp. Name des Artefakts, also das Produkt, das gebaut wird.
- ▶ NAME: new-tutego-webapp

Mit FINISH schließen wir ab. Im Package Explorer sieht die Ordnerstruktur anders aus als üblich, mit vier Code-Ordnern *src/main/java*, *src/main/resources*, *src/main/test* und *src/test/resources*. Neu ist auch ein Ordner *target*; einen versteckten *bin*-Ordner gibt es nicht mehr.

Beim Öffnen der *pom.xml* nutzt Eclipse einen eigenen Editor und unterschiedliche Reiter (OVERVIEW, DEPENDENCIES ...) für verschiedene Aspekte der Konfigurationsdatei. Unter POM.XML lässt sich die XML-Datei direkt bearbeiten, wenn die bereitgestellten Editoren besondere Einstellungen nicht zulassen.

### 15.10.2 Properties hinzunehmen

Als Erstes wollen wir den Java-Compiler auf Version 11 hochsetzen. Dazu falten wir im Reiter OVERVIEW den Bereich PROPERTIES aus und drücken CREATE..., um zwei Eigenschaften hinzuzufügen, die je aus NAME und VALUE bestehen:

- ▶ maven.compiler.target auf 11
- ▶ maven.compiler.source auf 11

Wechseln wir in die XML-Ansicht, hat Eclipse es so umgesetzt:

```
<properties>
  <maven.compiler.target>11</maven.compiler.target>
  <maven.compiler.source>11</maven.compiler.source>
</properties>
```

Wir wollen von Hand eine weitere Eigenschaft in das Element properties hinzunehmen für die Encodierung, die immer UTF-8 sein soll:

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

Zwar nutzt Maven nun den Java 11-Compiler und sieht alle Dateien encodiert als UTF-8 an, doch Eclipse bekommt davon nichts mit. Wir müssen Eclipse daher befehlen, aus der POM Informationen auszulesen und in die Projekteigenschaften zu übertragen. Daher gehen wir links auf den PACKAGE EXPLORER und aktivieren im Kontextmenü MAVEN • UPDATE PROJECT ... Danach steht bei den Projekten JRE SYSTEM LIBRARY [JAVASE-11].

### 15.10.3 Dependency hinzunehmen

Wir wollen als Beispiel eine Abhängigkeit zu einem kleinen Web-Framework Spark (<http://sparkjava.com>) herstellen. Wir wechseln im POM-Editor auf den Reiter DEPENDENCIES und drücken die Schaltfläche ADD. In das Textfeld GROUP ID kommt »com.sparkjava«, in ARTIFACT ID »spark-core«, und bei VERSION setzen wir »2.8.0« hinein.

Die POM-Datei sieht nach dem Hinzufügen der Abhängigkeit so aus:

**Listing 15.6** pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.tutego.webapp</groupId>
    <artifactId>tutego-webapp</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>new-tutego-webapp</name>
    <properties>
        <maven.compiler.target>11</maven.compiler.target>
        <maven.compiler.source>11</maven.compiler.source>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.sparkjava</groupId>
            <artifactId>spark-core</artifactId>
            <version>2.8.0</version>
        </dependency>
    </dependencies>
</project>
```

**Tipp**

Die besondere Stärke von Maven liegt im Auflösen transitiver Abhängigkeiten. Welche JAR-Dateien und Abhängigkeiten Spark nach sich zieht, zeigt der Reiter **DEPENDENCY HIERARCHY**.

Alles ist vorbereitet, Zeit für das Hauptprogramm:

**Listing 15.7** src/main/java/SparkServer.java

```
public class SparkServer {
    public static void main( String[] args ) {
        spark.Spark.get( "/hello", ( req, res ) -> "Hallo Browser " + req.userAgent() );
    }
}
```

Starten wir das Programm wie üblich mit RUN AS • JAVA APPLICATION, startet ein Webserver, und unter der URL <http://localhost:4567/hello> können wir die Ausgabe ablesen. (Die Warnungen können wir ignorieren.) Über RUN AS liegen auch weitere Menüpunkte, die direkt in gewisse Stellen des Lebenszyklus eines Maven-Builds hineinspringen lassen.



Über das Terminal-Icon lässt sich die spezielle Maven-Console öffnen, die Mavens Konsolenausgabe zeigt.

#### 15.10.4 Lokales und Remote-Repository

Das Auflösen der abhängigen Java-Archive dauert beim ersten Mal länger, da Maven ein Remote Repository kontaktiert und von dort immer die neusten JAR-Dateien bezieht und lokal ablegt. Das umfangreiche Remote Repository speichert zu vielen bekannten quelloffenen Projekten fast alle Versionen von JAR-Dateien. Das *Central Repository* hat die URL <https://repo.maven.apache.org/maven2/>.



Unter WINDOW • SHOW VIEW • OTHER ... • MAVEN • MAVEN REPOSITORIES zeigt Eclipse alle eingebundenen Repositories an.

In Eclipses PACKAGE EXPLORER lassen sich unter dem Projekt bei MAVEN DEPENDENCIES über 15 JAR-Dateien ablesen. Gespeichert werden sie selbst nicht im Projekt, sondern in einem lokalen Repository, das im Heimverzeichnis des Anwenders liegt und *.m2* heißt. Auf diese Weise teilen sich alle Maven-Projekte die gleichen JAR-Dateien, und diese müssen nicht projektweise immer neu bezogen und aktualisiert werden.

#### 15.10.5 Lebenszyklus, Phasen und Maven-Plugins

Ein Maven-Build besteht aus einem dreistufigen Lebenszyklus *clean*, *default* und *site*. Innerhalb dessen gibt es *Phasen*, zum Beispiel in *default* die Phase *compile* zum Übersetzen der Quellen. Alles, was Maven ausführt, sind *Plugins*, etwa *compiler* und viele andere, die <http://>

[maven.apache.org/plugins/](http://maven.apache.org/plugins/) auflistet. Ein Plugin kann unterschiedliche *Goals* ausführen. So kennt zum Beispiel das Javadoc-Plugin – beschrieben unter <http://maven.apache.org/components/plugins/maven-javadoc-plugin/> – 14 Goals. Ein Goal wird später über die Kommandozeile angesprochen oder über das RUN As-Kontextmenü auf der *pom.xml* in Eclipse.

### 15.10.6 Archetypes

Ein Maven-*Archetype* ist eine Vorlage für neue Projekte, sodass gleich diverse Klassen und Konfigurationen für einen schnellen Start generiert werden. Die Archetypen werden in einem Katalog gesammelt, und <http://repo.maven.apache.org/maven2/archetype-catalog.xml> ist so ein Remote-Katalog.

In Eclipse lässt sich ein Katalog über WINDOWS • PROPERTIES • MAVEN • ARCHETYPES und ADD REMOTE CATALOG ... hinzufügen. Anschließend können wir in Eclipse wie am Anfang ein Maven-Projekt aufbauen, nur ist jetzt CREATE A SIMPLE PROJECT nicht zu aktivieren, sondern mit NEXT auf die nächste Dialogseite zu wechseln. Jetzt lässt sich aus einer riesigen Liste ein Archetyp auswählen.

## 15.11 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, objektrelationalen Mappern (<http://www.eclipse.org/eclipse-link>, <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 *Apache Commons* (<http://commons.apache.org>) eine hübsche Sammlung an Klassen und Methoden zusammengetragen, und das Studium der Quellen sollte für Softwareentwickler mehr zum Alltag gehören. Die Website <https://www.openhub.net/> 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 16

## Einführung in die nebenläufige Programmierung

»just be.«

– Calvin Klein (\* 1942)

### 16.1 Nebenläufigkeit und Parallelität

Computersysteme lösen Probleme in der echten Welt, sodass wir zum Einstieg auch in der Realwelt bleiben, um uns dem Umfeld der nebenläufigen Programmierung zu nähern. Gehen wir durch die Welt, bemerken wir viele Dinge, die gleichzeitig passieren: Die Sonne scheint, auf der Straße fahren Mofas, Autos werden gelenkt, das Radio spielt, Menschen sprechen, einige essen, Hunde tollen auf der Wiese. Nicht nur passieren diese Dinge gleichzeitig, sondern es gibt mannigfaltige Abhängigkeiten, wie Wartesituationen: An der roten Ampel warten einige Autos, während bei der grünen Ampel Menschen über die Straße gehen – beim Signalwechsel dreht sich das Spiel um.

Wenn viele Dinge gleichzeitig passieren, nennen wir ein interagierendes System *nebenläufig*. Dabei gibt es Vorgänge, die echt *parallel* ausgeführt werden können, und bei machen Dingen sieht es so aus, als ob sie parallel passierten, aber in Wirklichkeit passieren sie nur schnell hintereinander. Was wir dann wahrnehmen, ist eine *Quasiparallelität*. Wenn zwei Menschen etwa gleichzeitig essen, ist das parallel, aber wenn jemand isst und atmet, so sieht das zwar von außen gleichzeitig aus, ist es aber nicht, sondern Schlucken und Atmen sind sequenziell.<sup>1</sup> Auf Software übertragen: Die gleichzeitige Abarbeitung von Programmen und Nutzung von Ressourcen ist nebenläufig; es ist eine technische Realisierung der Maschine (also Hardware), ob diese Nebenläufigkeit durch parallele Abarbeitung – etwa durch mehrere Prozessoren oder Kerne – auch wirklich umgesetzt wird.

Nebenläufige Programme werden in Java durch Threads realisiert, wobei jeder Thread einer Aufgabe entspricht. Im Idealfall findet die Abarbeitung der Threads auch parallel statt, wenn die Maschine mehrere Prozessoren oder Kerne hat. Ein Programm, das nebenläufig realisiert ist, kann durch zwei Prozessoren bzw. Kerne in der parallelen Abarbeitung in der Zeit halbiert werden, muss es aber nicht, es ist immer noch Sache des Betriebssystems, wie es die Threads ausführt.

---

<sup>1</sup> Lassen wir Kleinkinder einmal außen vor.

### 16.1.1 Multitasking, Prozesse, Threads

Ein modernes Betriebssystem gibt dem Benutzer die Illusion, dass verschiedene Programme gleichzeitig ausgeführt werden – die Betriebssysteme unterstützen das *Multitasking* und nennen sich *multitaskingfähig*. Bei der Ausführung eines Programms erzeugt das Betriebssystem einen so genannten Prozess – alle laufenden 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 korrumpiert; 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.

Diese Nebenläufigkeit der Prozesse wird durch das Betriebssystem gewährleistet, das auf Einprozessormaschinen die Prozesse alle paar Millisekunden umschaltet. Der Teil des Betriebssystems, der die Umschaltung übernimmt, heißt *Dispatcher*. Daher ist das Programm bei einem Prozessor (mit nur einem Kern) zwar nebenläufig, aber nicht wirklich parallel, sondern das Betriebssystem gaukelt uns dies durch eine verzahnte Bearbeitung der Prozesse vor. Wenn mehrere Prozessoren oder mehrere Prozessorkerne am Werke sind, werden die Programmteile tatsächlich parallel abgearbeitet. Die Informationen, welche Prozesse welche Rechenzeit bekommen, stammen vom *Scheduler*.

### 16.1.2 Threads und Prozesse

Bei modernen Betriebssystemen gehört zu jedem Prozess mindestens ein *Thread* (zu Deutsch *Faden* oder *Ausführungsstrang*), der den Programmcode ausführt. Damit werden genau genommen die Prozesse nicht mehr nebenläufig 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.

#### Java-Threads sind native Betriebssystem-Threads

Die Programmierung von Threads ist in Java einfach möglich, und die nebenläufigen Aktivitäten vermitteln dem Benutzer den Eindruck von Gleichzeitigkeit. Alle modernen Betriebssysteme unterstützen Threads direkt, und so 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 Mehrkern-/Mehrprozessorsystemen, da sich das Betriebssystem um die ganze Thread-Verwaltung kümmert.

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.

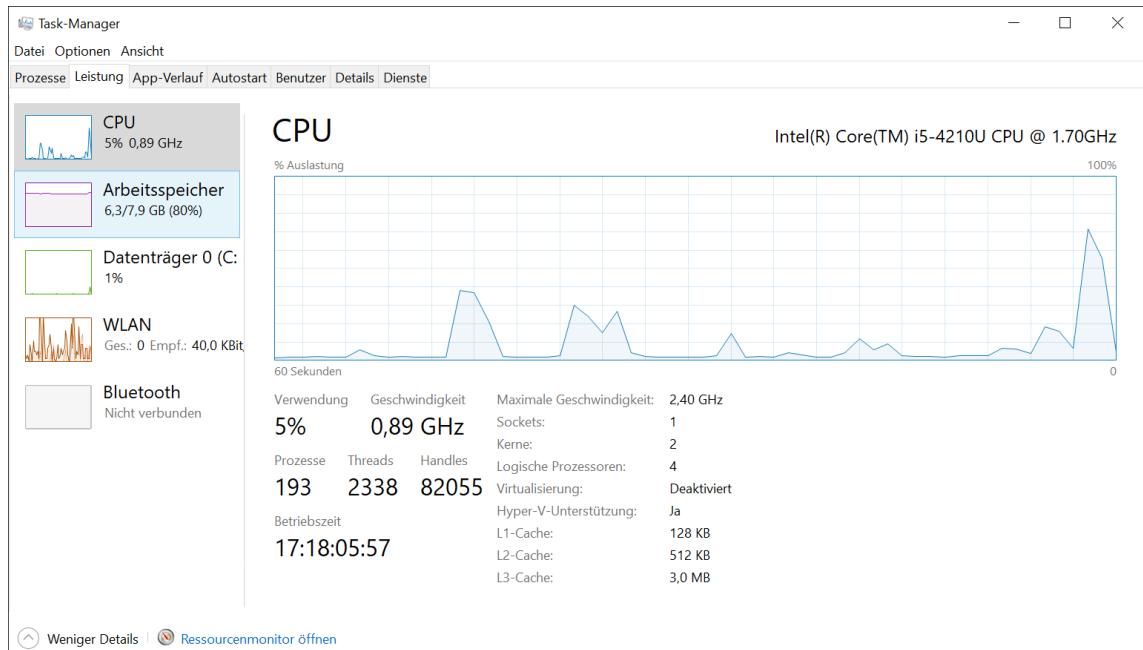


Abbildung 16.1 Windows zeigt im Task-Manager die Anzahl laufender Threads an.

### 16.1.3 Wie nebenläufige 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 wird kontinuierlich aktualisiert. Schließlich werden die Ergebnisse in eine Datei geschrieben. Die Schritte sind:

1. Baue ein Fenster auf.
2. Öffne die Datenbank vom Netzserver, 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 Nebenläufigkeit und parallele Abarbeitung 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 nebenläufig ausgeführt werden können:

- ▶ Das Öffnen des Fensters und das Öffnen der Datenbank können 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 Datenbankserver 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 Datenbankserver 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 nebenläufige Modellierung 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 Netzwerzkzugriff auf eine Datenbank anfällt, kann für andere Aktivitäten genutzt werden. Tabelle 16.1 gibt die Elemente zum Kombinieren noch einmal an.

Ressource	Belastung
Hauptspeicherzugriffe	Prozessor
Dateioperationen	Festplatte
Datenbankzugriff	Server, Netzwerkverbindung

**Tabelle 16.1** 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 nebenläufig implementierter Sortieralgorithmus mit einem Prozessor(kern) 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.

#### 16.1.4 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 nebenläufig ausführen soll.
- ▶ Lock: Dient dem Markieren von kritischen Abschnitten, in denen sich nur ein Thread befinden darf.
- ▶ Condition: Threads können auf die Benachrichtigung anderer Threads warten.

## 16.2 Laufende Threads, neue Threads erzeugen

Die folgenden Abschnitte verdeutlichen, wie der nebenläufige Programmcode in einen Runnable verpackt und dem Thread zur Ausführung vorgelegt wird.

### 16.2.1 Main-Thread

Jeder Programmcode in Java läuft immer in einem Thread. Wenn die JVM startet, erzeugt sie automatisch einen Thread, genannt *Main-Thread*. Er arbeitet die `main(...)`-Methode ab. Das heißt, alles, was wir in die `main(...)`-Methode setzen, wird von dem Main-Thread abgearbeitet. In einem Debugger lässt sich das gut nachvollziehen.

In der Regel gibt es in der Java-Umgebung noch weitere Threads, die von Java-Bibliotheken gestartet werden. Öffnet Java ein Fenster, gibt es einen GUI-Thread, in dem Ereignisse abgearbeitet werden.

### 16.2.2 Wer bin ich?

In Java wird ein Thread über die Klasse `java.lang.Thread` repräsentiert. Er verbindet die Java-Seite mit der Betriebssystemseite.

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 nichtstatische Thread-Methoden wie `toString()` verwenden.



#### Beispiel

Ruf `toString()` auf dem aktuell laufenden Thread auf:

```
public static void main( String[] args ) {
    System.out.println( Thread.currentThread() ); // Thread[main,5,main]
}
```

Die String-Präsentation besteht aus dem Thread-Namen, der Priorität und einer zugehörigen Thread-Gruppe.

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.

### 16.2.3 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 überge-

ben. Wird der Thread gestartet, arbeitet er die Programmzeilen aus dem Befehlsobjekt nebenläufig zum restlichen Programmcode ab. Die funktionale Schnittstelle `Runnable` schreibt nur eine `run()`-Methode vor.

```
interface java.lang.Runnable
```

- `void run()`

Implementierende Klassen realisieren die Operation und setzen dort den nebenläufig auszuführenden Programmcode ein.



Abbildung 16.2 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 16.1** src/main/java/com/tutego/insel/thread/DateCommand.java

```
package com.tutego.insel.thread;

public class DateCommand implements Runnable {
    @Override public void run() {
        Stream.generate( LocalDateTime::now ).limit( 20 ).forEach( System.out::println );
    }
}
```

**Listing 16.2** src/main/java/com/tutego/insel/thread/CounterCommand.java

```
package com.tutego.insel.thread;

class CounterCommand implements Runnable {
    @Override public void run() {
        IntStream.range( 0, 20 ).forEach( System.out::println );
    }
}
```

Unser nebenläufig auszuführender Programmcode in `run()` besteht aus einem limitierten Stream. Im ersten Fall gibt das Programm 20 aktuelle Datumswerte aus und im anderen Fall einen Schleifenzähler.

#### 16.2.4 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 neben der eigentlichen 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 16.3** src/main/java/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 so aussehen kann:

```
0
1
2
3
4
5
6
7
8
2017-02-17T13:40:35.315250400
2017-02-17T13:40:35.327260600
2017-02-17T13:40:35.327260600
2017-02-17T13:40:35.327260600
9
10
11
12
13
14
15
16
17
18
```

19

2017-02-17T13:40:35.327260600

...

In meiner Ausgabe ist die Verzahnung der beiden Threads zu erkennen. Zwar ist innerhalb eines Threads durch die Schleife die Reihenfolge klar definiert, doch wann welcher Thread an der Reihe ist, ist frei. So sollte auch nicht verwundern, dass die erste Zeile in meiner Ausgabe vom Zähl-Thread kommt – dem eigentlich zweiten Thread. Das zeigt deutlich den Nicht-determinismus<sup>2</sup> bei Threads. Interpretiert werden kann die Ausgabe durch die unterschiedlichen Laufzeiten, die für die Datums- und Zeitausgabe nötig sind; bei der Datumsverarbeitung sind viel mehr Objekte nötig, sie aufzubauen dauert. Aber das ist ein Internum, das bei der Reihenfolge nicht beachtet werden darf. Wer Thread-Ergebnisse in bestimmten Reihenfolgen erwartet, muss Threads synchronisieren.

```
class java.lang.Thread
implements Runnable
```

- `Thread(Runnable target)`  
Erzeugt einen neuen Thread mit einem Runnable, das den nebenläufig 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.

### Tipp



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, und der Konstruktor der Oberklasse startet einen Thread. Bildet die Applikation ein Exemplar der Unterkorrekte, 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.

## 16.2.5 Runnable parametrisieren

Die `run()`-Methode von Runnable hat keine Parameterliste und keine Rückgabe. Es stellt sich daher die Frage, wie Informationen in das Runnable reinkommen und wieder rauskommen.

---

<sup>2</sup> Nicht vorhersehbar; bedeutet hier: Wann der Scheduler den Kontextwechsel vornimmt, ist unbekannt.

Dass `run()` keine Parameterliste hat, ist schnell dadurch erklärt, dass ja völlig unbekannt ist, welche Parameter es überhaupt gibt.

Es gibt zwei einfache Möglichkeiten, innerhalb von `run()` auf Parameter zuzugreifen:

- ▶ Implementiert eine Klasse die Schnittstelle `Runnable`, kann sich ein Konstruktor die Werte in internen Zuständen merken. Ruft der Thread `run()` auf, kann die Methode auf die Werte zurückgreifen.
- ▶ Realisiert ein Lambda-Ausdruck die funktionale Schnittstelle `Runnable`, so kann der Rumpf auf lokale Variablen aus dem Kontext zugreifen.

Das Ablegen von Ergebnissen ist schwieriger als auf den ersten Blick vermutet. Das liegt an dem Problem, dass es nebenläufige Schreibzugriffe geben könnte, und daher sind Synchronisierungsmechanismen nötig. Grundsätzlich sind auch hier zwei Möglichkeiten denkbar:

- ▶ Eine `Runnable`-Implementierung kommt zu einem Ergebnis und legt es in einem eigenen Zustand ab, der später von außen erfragt wird.
- ▶ `run()` schreibt in nicht selbstverwalteten Speicher.

### 16.2.6 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 nebenläufige Aktivitäten programmieren wollen:

**Listing 16.4** src/main/java/com/tutego/insel/thread/DateThread.java, DateThread

```
public class DateThread extends Thread {
    @Override public void run() {
        Stream.generate( LocalDateTime::now ).limit( 20 ).forEach( System.out::println );
    }
}
```

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 16.5** src/main/java/com/tutego/insel/thread/DateThreadUser.java, 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(), so dass sie den nebenläufig auszuführenden Programmcode enthält.

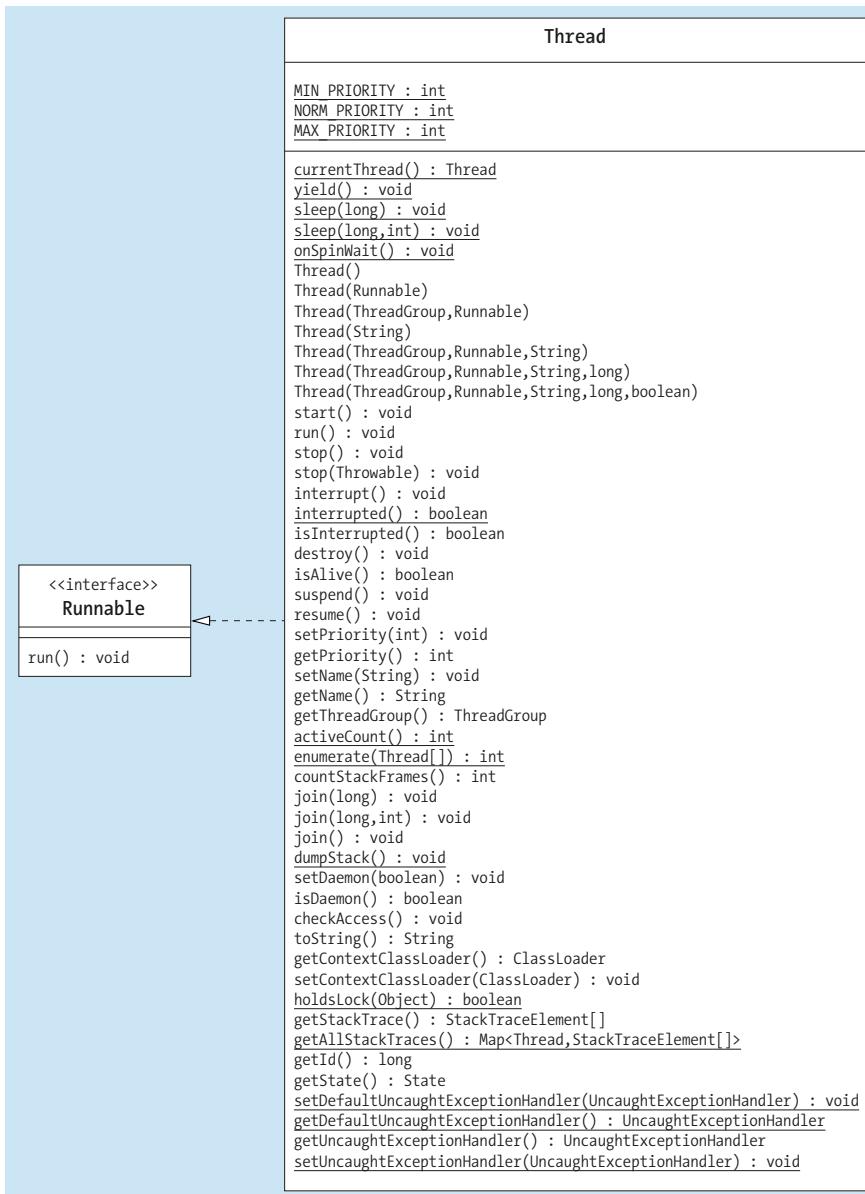


Abbildung 16.3 UML-Diagramm der Klasse Thread, die Runnable implementiert

### Überschreiben von start() und Selbststarter

Die Methode `start()` kann von uns auch überschrieben werden, was aber nur selten sinnvoll bzw. 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 sich ein Thread auch selbst starten. Der Konstruktor ruft dazu einfach die eigene `start()`-Methode auf:

```
class DateThread extends Thread {
    DateThread() {
        start();
    }
    // ... der Rest bleibt ...
}
```

### Statt start() wurde run() aufgerufen – ja, wo laufen sie denn?

`run()` ist wie `start()` eine öffentliche Methode der Klasse `Thread` – `Thread` implementiert `Runnable`, daher muss die `run()`-Methode `public` sein. 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()`, mit dem Unterschied, dass die Objektmethode `run()` nicht nebenläufig 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.

## 16.3 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.

### 16.3.1 Der Name eines Threads

Ein Thread hat eine ganze Menge Eigenschaften – wie einen Zustand, eine Priorität und auch einen Namen. Der Name 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.

### 16.3.2 Die Zustände eines Threads \*

Bei einem Thread-Exemplar können wir einige Zustände feststellen:

1. *Neu*: Der Lebenslauf eines Thread-Objekts beginnt mit `new`, doch läuft der Thread noch nicht.
2. *Laufend* und *bereit*: Durch `start()` gelangt der Thread in den Zustand »bereit« bzw. »laufend«. Der Thread wechselt immer zwischen den zwei Zuständen hin und her. Läuft der Thread, hat ihn der Scheduler vom Betriebssystem zur Ausführung ausgewählt: ist er *bereit*, so läuft er aktuell nicht, wird aber vom Scheduler berücksichtigt für die nächste Runde, bis der Scheduler ihm wieder Rechenzeit zuordnet.
3. *Wartend*: Dieser Zustand wird mittels spezieller Synchronisationstechniken oder Ein-/Ausgabefunktionen erreicht – der Thread verweilt in einem Wartezustand.
4. *Terminiert*: Nachdem die Aktivität des Thread-Objekts beendet wurde, kann es nicht mehr aktiviert werden und ist tot, also beendet.

#### Mit Lebenszeichen – Zustand über `Thread.State`

In welchem Zustand ein Thread gerade ist, zeigt die Methode `getState()`. Sie liefert ein Objekt vom Typ der Aufzählung `Thread.State` (die einzige Aufzählung im Paket `java.lang`), das Folgendes deklariert:

Zustand	Erläuterung
NEW	neuer Thread, noch nicht gestartet
RUNNABLE	Läuft in der JVM.
BLOCKED	Wartet auf einen MonitorLock, wenn er etwa einen synchronized-Block betreten möchte.
WAITING	Wartet etwa auf ein notify().
TIMED_WAITING	Wartet etwa in einem sleep().
TERMINATED	Ausführung ist beendet.

Tabelle 16.2 Zustände eines Threads

Zudem lässt sich die Methode `isAlive()` verwenden, die erfragt, ob der Thread gestartet wurde, aber noch nicht tot ist.

### 16.3.3 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.



#### Beispiel

Der ausführende Thread soll 2 Sekunden lang schlafen. Einmal mit `Thread.sleep(...)`:

```
try {
    Thread.sleep( 2000 );
} catch ( InterruptedException e ) { }
```

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.

```
class java.lang.Thread
    implements Runnable
```

- `static void sleep(long millis) throws InterruptedException`

Der aktuell ausgeführte Thread wird `millis` Millisekunden schlafen gelegt; eine kleine Ungenauigkeit ist natürlich möglich. 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 `millis` Millisekunden und zusätzlich `nanos` Nanosekunden schlafen gelegt. Im Gegensatz zu `sleep(long)` wird bei einer negativen Millisekundenanzahl eine `IllegalArgumentException` ausgelöst; diese Exception wird auch ausgelöst, wenn die Nanosekundenanzahl nicht zwischen 0 und 999.999 liegt.

#### Tipp

Um Entwickler von der Nutzung einer veralteten API abzubringen, kann gemeinerweise eine Verzögerung eingebaut werden – schnell wird sich dann die neue API durchsetzen.



```
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.

#### Tipp

Um nach einer gewissen Zeit etwas zu tun, muss kein eigener Thread gestartet und dann in den Ruhemodus gelegt werden, sondern es können aus der Java-API Timer verwendet werden.



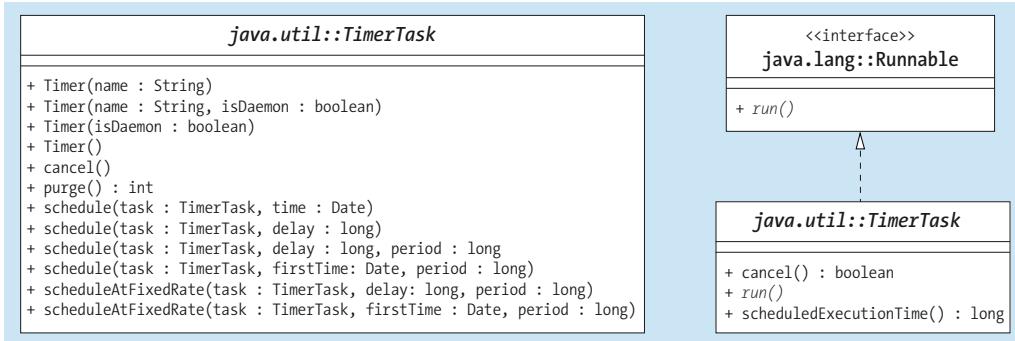


Abbildung 16.4 UML-Diagramme für Timer und TimerTask

#### 16.3.4 Mit yield() und onSpinWait() auf Rechenzeit verzichten

Im besten Fall signalisiert ein Ereignis, dass Daten bereitstehen und abgeholt werden können. Doch nicht immer gibt es eine solche API. Dann findet sich oft ein wiederholter Test auf eine Bedingung, und wird diese wahr, stehen zum Beispiel Daten zum Abholen bereit. In Code sieht das so aus:

```

while ( ! datenDa() )
;
datenVearbeiten();

```

Dieses *aktive Warten*, engl. *busy waiting* oder auch *spinning* genannt, auf Daten – was auf Englisch *polling* genannt wird – mithilfe einer *spin-loop* verschwendet so programmiert viel Rechenzeit. Eine Lösung ist, mit `sleep(...)` einige Millisekunden zu schlafen, wobei es schwierig ist, die richtige Anzahl an Schlafsekunden zu ermitteln. Zu kurz geschlafen heißt: noch einmal weiterpollen; zu lang geschlafen heißt: Es gibt eine unnötige Verzögerung.

Die Thread-Klasse bietet zwei weitere Methoden zum Programmieren kooperativer Threads: die Methoden `yield()` und `onSpinWait()`. Anders als bei `sleep(...)` gibt es hier keine Millisekunden anzugeben.

```

while ( ! datenDa() )
Thread.onSpinWait();
datenVearbeiten();

```

Insgesamt ist Warten aber nie klug. Besser ist es, sich benachrichtigen zu lassen, wenn Daten anliegen.

```

class java.lang.Thread
implements Runnable

```

- static void onSpinWait()
 

Signalisiert der Laufzeitumgebung, dass der Thread in einer *spin-loop* auf Daten wartet. Diesen Hinweis kann die JVM an den Prozessor weitergeben, und die Laufzeit ist bei typischen Systemen besser als mit `yield()`.<sup>3</sup>
- static void yield()
 

Der laufende Thread gibt freiwillig seine Rechenzeit ab, sodass er bezüglich seiner Priorität wieder in die Thread-Warteschlange des Systems einordnet wird. Einfach ausgedrückt, signalisiert `yield()` der Thread-Verwaltung: »Ich setze diese Runde aus und mache weiter, wenn ich das nächste Mal dran bin.« Die Methode ist für Implementierungen der JVM nicht verbindlich. Die API-Dokumentation warnt eher vor der Methode: »It is rarely appropriate to use this method.«

### 16.3.5 Wann Threads fertig sind

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.

### 16.3.6 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. Die naheliegende Möglichkeit, mit der Thread-Methode `stop()` einen Thread abzuwürgen, wollen wir in [Abschnitt 16.3.8](#), »Der `stop()` von außen und die Rettung mit `ThreadDeath`\*«, diskutieren.

Wenn wir den Thread schon nicht von außen beenden wollen, können wir ihn immerhin bitten, seine Arbeit aufzugeben und den Freitod zu wählen. Periodisch müsste er dann nur überprüfen, ob jemand von außen den Abbruchswunsch geäußert hat.

---

<sup>3</sup> Siehe »JEP 285: Spin-Wait Hints«, <http://openjdk.java.net/jeps/285>

### 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 2 Sekunden wird der Unterbrechungswunsch mit `interrupt()` gemeldet. Auf dieses Signal achtet die sonst unendlich laufende Schleife und bricht ab:

**Listing 16.6** src/main/java/com/tutego/insel/thread/ThreadausInterruptus.java, main(...)

```
Runnable killingMeSoftly = () -> {
    System.out.println( "Es gibt ein Leben vor dem Tod." );

    while ( ! Thread.currentThread().isInterrupted() ) {
        System.out.println( "Und er läuft und er läuft und er läuft" );

        try {
            Thread.sleep( 500 );
        }
        catch ( InterruptedException e ) {
            Thread.currentThread().interrupt();
            System.out.println( "Unterbrechung in sleep()" );
        }
    } // end while

    System.out.println( "Das Ende" );
};

Thread t = new Thread( killingMeSoftly );
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
```



### Hinweis

Wird von Thread geerbt, sind die Methoden `isInterrupted()` und `interrupt()` sofort in der Unterklasse zugänglich.

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 erregt 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 Schlafen stören. Das mag die Methode überhaupt nicht und reagiert – gut dokumentiert in der Javadoc – wie folgt:

1. `sleep(...)` löst bei Unterbrechung mit `interrupt()` eine `InterruptedException` aus. Das ist genau die Ausnahme, die unser `try-catch`-Block auffängt.
2. Die Methode setzt das Interrupt-Flag zurück. Eine Abfrage über `isInterrupted()` meldet folglich keine Unterbrechung.

Es ist gut, dass das Interrupt-Flag zurückgesetzt wird, denn es hat ja schon seinen Dienst erwiesen. In unserem Fall muss aber erneut `interrupt()` aufgerufen werden, da das Abbruch-Flag neu gesetzt werden muss, damit `isInterrupted()` in der `while`-Schleife das Ende bestimmen kann.

Wenn wir mit der Objektmethode `isInterrupted()` arbeiten, müssen wir beachten, dass neben `sleep(...)` auch die Object-Methoden `join(...)` und `wait(...)` durch die `InterruptedException` das Flag löschen.



### Tipp

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: `interrupted()`, `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, was `isInterrupted()` nicht tut. Zwei aufeinanderfolgende Aufrufe von `interrupted()` führen daher zu einem `false`, es sei denn, in der Zwischenzeit erfolgt eine weitere Unterbrechung.

### 16.3.7 Unbehandelte Ausnahmen, Thread-Ende und `UncaughtExceptionHandler`

Kommt es bei der Thread-Ausführung zu einer unbehandelten nichtgeprüften Ausnahme, so beendet die JVM den Thread. Das wird nicht gesondert auf der Konsole geloggt, lediglich die Exception-Meldung kommt.



#### Beispiel

Die JVM beendet den Thread aufgrund einer nicht abgefangenen Exception, der main-Thread läuft weiter:

```
Thread t = new Thread( () -> {
    System.out.println( Thread.currentThread() );
    System.out.println( 1 / 0 );
}, "Knallerthread" );
t.start();
System.out.println( t.isAlive() );
Thread.sleep( 1000 );
System.out.println( Thread.currentThread() );
System.out.println( t.isAlive() );
```

Die Ausgabe ist:

```
true
Thread[Knallerthread,5,main]
Exception in thread "Knallerthread" java.lang.ArithmetricException: / by zero
  at T.lambda$0(T.java:8)
  at java.base/java.lang.Thread.run(Thread.java:844)
Thread[main,5,main]
false
```

#### `UncaughtExceptionHandler` setzen

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 geschachtelte Schnittstelle, die eine Operation `void uncaughtException(Thread t, Throwable e)` vorschreibt. Eine Implementierung der Schnittstelle lässt sich entweder einem individuellen Thread oder allen Threads hängen, sodass im Fall des Abbruchs durch unbehandelte Ausnahmen die JVM die Methode `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.

### 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 andere Lösung gibt `Callable` – siehe [Abschnitt 16.4.3, »Threads mit Rückgabe über Callable«](#) –, mit dem ein spezieller Fehlercode zurückgegeben oder eine Exception angezeigt werden kann.

### 16.3.8 Der `stop()` von außen und die Rettung mit `ThreadDeath` \*

Wenn ein Thread nicht auf `interrupt()` hört, aber aus irgendwelchen Gründen dringend beendet werden muss, müssen wir wohl oder übel die veraltete Methode `stop()` einsetzen.



Abbildung 16.5 Dass die Methode `stop()` veraltet ist, zeigen in Eclipse eine unter-schlängelte Linie und ein Symbol am linken Rand an. Steht der Cursor auf der problematischen Zeile, weist eine Fehlermeldung ebenfalls auf das Problem hin.

DEPRECATED gibt uns schon einen guten Hinweis darauf, `stop()` besser nicht zu benutzen (leider gibt es hier, im Gegensatz zu den meisten anderen veralteten Methoden, keinen einfachen, empfohlenen Ersatz). Überschreiben können wir `stop()` auch nicht, da es final ist. Wenn wir einen Thread von außen beenden, geben wir ihm keine Chance mehr, seinen Zustand konsistent zu verlassen. Zudem kann die Unterbrechung an beliebiger Stelle erfolgen, sodass angeforderte Ressourcen frei in der Luft hängen können.

```
class java.lang.Thread
implements Runnable
```

- `final void stop()`

Wurde der Thread gar nicht gestartet oder ist er bereits abgearbeitet bzw. beendet, kehrt die Methode sofort zurück. Andernfalls wird über `checkAccess(...)` geprüft, ob wir überhaupt das Recht haben, den Thread abzuwürgen. Dann wird der Thread beendet, egal, was er zuvor unternommen hat; jetzt kann er nur noch sein Testament in Form eines `ThreadDeath`-Objekts als Exception anzeigen.

### 16.3.9 Ein Rendezvous mit `join(...)` \*

Wollen wir Aufgaben auf mehrere Threads verteilen, kommt der Zeitpunkt, an dem die Ergebnisse eingesammelt werden. Die Resultate können allerdings erst dann zusammengebracht werden, wenn alle Threads mit ihrer Ausführung fertig sind. Da sie sich zu einem bestimmten Zeitpunkt treffen, heißt das auch *Rendezvous*.

Zum Warten gibt es mehrere Strategien. Zunächst lässt sich mit `Callable` arbeiten, um dann mit `get()` synchron auf das Ende zu warten. Arbeiten wir mit `Runnable`, so kann ein Thread keine direkten Ergebnisse wie eine Methode nach außen geben, weil die `run()`-Methode den Ergebnistyp `void` hat. Da ein nebenläufiger Thread zudem asynchron arbeitet, wissen wir nicht einmal, wann wir das Ergebnis erwarten können.

Die Übertragung von Werten ist kein Problem. Hier können Klassenvariablen und auch Objektvariablen helfen, denn über sie können wir kommunizieren. Jetzt fehlt nur noch, dass wir auf das Ende der Aktivität eines Threads warten können. Das geht mit `join(...)`-Methoden.

In unserem folgenden Beispiel legt ein Thread `thread` in der Variablen `result` ein Ergebnis ab. Wir können die Auswirkungen von `join()` sehen, wenn wir die auskommentierte Zeile hinzunehmen:

**Listing 16.7** src/main/java/com/tutego/insel/thread/JoinTheThread.java, main(...)

```
class JoinerRunnable implements Runnable {
    public int result;

    @Override public void run() {
        result = LocalDate.now().getDayOfYear();
    }
}

JoinerRunnable runnable = new JoinerRunnable();
Thread thread = new Thread( runnable );
thread.start();
// thread.join();
System.out.println( runnable.result );
```

Ohne den Aufruf von `join()` wird als Ergebnis 0 ausgegeben, denn das Starten des Threads kostet etwas Zeit. In dieser Zeit geben wir die automatisch auf 0 initialisierte Objektvariable aus. Nehmen wir `join()` hinein, wird die `run()`-Methode bis zum Ende ausgeführt, und der Thread setzt die Variable `result` auf den Tag des Jahres (Wertebereich 1 bis 366). Das sehen wir dann auf dem Bildschirm.

```
class java.lang.Thread
implements Runnable
```

- `final void join() throws InterruptedException`

Der aktuell ausgeführte Thread wartet auf den Thread, für den die Methode aufgerufen wird, bis dieser beendet ist.

- `final void join(long millis) throws InterruptedException`

Wie `join()`, doch wartet diese Variante höchstens `millis` Millisekunden. Wurde der Thread bis dahin nicht vollständig beendet, fährt das Programm fort. Auf diese Weise kann versucht werden, innerhalb einer bestimmten Zeitspanne auf den Thread zu warten, sonst aber weiterzumachen. Ist `millis` gleich 0, so hat dies die gleiche Wirkung wie `join()`.

- `final void join (long millis, int nanos) throws InterruptedException`

Wie `join(long)`, jedoch mit potenziell genauerer Angabe der maximalen Wartezeit

Nach einem `thread.join(long)` ist mitunter die `thread.isAlive()`-Methode nützlich, denn sie sagt aus, ob `thread` noch aktiv arbeitet oder beendet ist.

In TimeUnit gibt es mit `timedJoin(...)` eine Hilfsmethode, die es uns erlaubt, mit der Dauer schöner zu arbeiten.

```
class java.lang.TimeUnit
implements Runnable
```

- `void timedJoin(Thread thread, long timeout) throws InterruptedException`  
Berechnet aus der TimeUnit und dem timeout Millisekunden (ms) und Nanosekunden (ns) und führt ein `join(ms, ns)` auf dem thread aus.

### Warten auf den Langsamsten

Große Probleme lassen sich in mehrere Teile zerlegen, und jedes Teilproblem kann dann von einem Thread gelöst werden. Dies ist insbesondere bei Mehrprozessorsystemen eine lohnenswerte Investition. Zum Schluss müssen wir nur noch darauf warten, dass die Threads zum Ende gekommen sind, und das Ergebnis einsammeln. Dazu eignet sich `join(...)` gut.



### Beispiel

Zwei Threads arbeiten an einem Problem. Anschließend wird gewartet, bis beide ihre Aufgabe erledigt haben. Dann könnte etwa ein anderer Thread die von a und b verwendeten Ressourcen wieder nutzen:

```
Thread a = new Thread(runnableA);
Thread b = new Thread(runnableB);
a.start();
b.start();
a.join();
b.join();
```

Es ist unerheblich, wessen `join()` wir zuerst aufrufen, da wir ohnehin auf den langsamsten Thread warten müssen. Wenn ein Thread schon beendet ist, kehrt `join()` sofort zurück.

Eine andere Lösung für zusammenlaufende Threads besteht darin, diese in einer Thread-Gruppe zusammenzufassen. Dann können sie zusammen behandelt werden, sodass nur das Ende der Thread-Gruppe beobachtet wird.

### 16.3.10 Arbeit niederlegen und wieder aufnehmen \*

Wollen wir erreichen, dass ein Thread für eine bestimmte Zeit die Arbeit niederlegt und ein anderer den schlafenden Thread wieder aufwecken kann, müssten wir das selbst implementieren. Zwar gibt es mit `suspend()` und `resume()` zwei Methoden, doch diese Start-Stopp-Technik ist nicht erwünscht, da sie ähnlich problematisch ist wie `stop()`.

```
class java.lang.Thread
    implements Runnable
```

- `final void suspend()`  
Lebt der Thread, wird er so lange eingefroren (schlafen gelegt), bis `resume()` aufgerufen wird.
- `final void resume()`  
Weckt einen durch `suspend()` lahmgelegten Thread wieder auf, der dann wieder arbeiten kann.

### 16.3.11 Priorität \*

Jeder Thread verfügt über eine Priorität, die aussagt, wie viel Rechenzeit ein Thread relativ zu anderen Threads erhält. Die Priorität ist eine Zahl zwischen `Thread.MIN_PRIORITY` (1) und `Thread.MAX_PRIORITY` (10). Durch den Wert kann der Scheduler erkennen, welchem Thread er den Vorzug geben soll, wenn mehrere Threads auf Rechenzeit warten. Bei seiner Initialisierung bekommt jeder Thread die Priorität des erzeugenden Threads. Normalerweise ist es die Priorität `Thread.NORM_PRIORITY` (5).

Das Betriebssystem (oder die JVM) nimmt die Threads immer entsprechend der Priorität aus der Warteschlange heraus (daher *Prioritätswarteschlange*). Ein Thread mit der Priorität  $N$  wird vor alle Threads mit der Wichtigkeit kleiner  $N$ , aber hinter diejenigen der Priorität größer gleich  $N$  gesetzt. Ruft nun ein kooperativer Thread mit der Priorität  $N$  die Methode `yield()` auf, bekommt ein Thread mit der Priorität  $\leq N$  auch eine Chance zur Ausführung.

Die Priorität kann durch Aufruf von `setPriority(...)` geändert und mit `getPriority()` abgefragt werden. Allerdings macht Java nur sehr schwache Aussagen über die Bedeutung und Auswirkung von Thread-Prioritäten.

#### Beispiel

Wir weisen dem Thread `t` die höchste Priorität zu:

```
t.setPriority( Thread.MAX_PRIORITY );
```

[zB]

```
class java.lang.Thread
    implements Runnable
```

- `final int getPriority()`  
Liefert die Priorität des Threads.

- `final void setPriority(int newPriority)`  
Setzt die Priorität des Threads. Die Methode liefert eine `IllegalArgumentException`, wenn die Priorität nicht zwischen `MIN_PRIORITY` (1) und `MAX_PRIORITY` (10) liegt.

### Granularität und Vorrang

Die zehn Prioritätsstufen garantieren nicht zwingend unterschiedliche Ausführungen. Obwohl anzunehmen ist, dass ein Thread mit der Priorität `NORM_PRIORITY+1` häufiger Programmcode ausführt als ein Thread mit der Priorität `NORM_PRIORITY`, kann ein Betriebssystem dies anders implementieren. Nehmen wir an, die Plattform implementiert lediglich fünf Prioritätsstufen. Ist 1 die niedrigste Stufe und 5 die höchste – die mittlere Stufe ist 3 –, werden wahrscheinlich `NORM_PRIORITY` und `NORM_PRIORITY + 1` auf die Stufe 3 transformiert und haben demnach dieselbe Priorität. Was wir daraus lernen: Auch bei unterschiedlichen Prioritäten können wir nicht erwarten, dass ein bestimmtes Programmstück zwingend schneller läuft. Zudem gibt es Betriebssysteme mit Schedulern, die keine Prioritäten unterstützen oder diese unerwartet interpretieren.

## 16.4 Der Ausführer (Executor) kommt

Zur nebenläufigen Ausführung eines `Runnable` ist immer ein Thread notwendig. Obwohl die nebenläufige Abarbeitung von Programmcode ohne Threads nicht möglich ist, sind doch beide in der bisherigen Programmierung 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 später über einen Setter 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. Mit anderen Worten: Ein existierender Thread kann nicht einfach ein neues `Runnable` abarbeiten.
- ▶ 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 Tagesschau) 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.

### 16.4.1 Die Schnittstelle Executor

Anstatt das Runnable direkt an einen Thread und somit an seinen Ausführer zu binden, gibt es eine Abstraktion für alle »Abarbeiter«. 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.

### 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. Mehr dazu im nächsten Abschnitt.
- static ScheduledExecutorService newSingleThreadScheduledExecutor()
- static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)  
Gibt spezielle Executor-Objekte zurück, um Wiederholungen festzulegen. Mehr dazu in [Abschnitt 16.4.7](#), »ScheduledExecutorService für wiederholende Aufgaben und Zeitsteuerungen nutzen«.

Es gibt über 20 Methoden in Executors; diese Aufzählung hier zeigt nur die, die für uns in den nächsten Abschnitten relevant sind.

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.

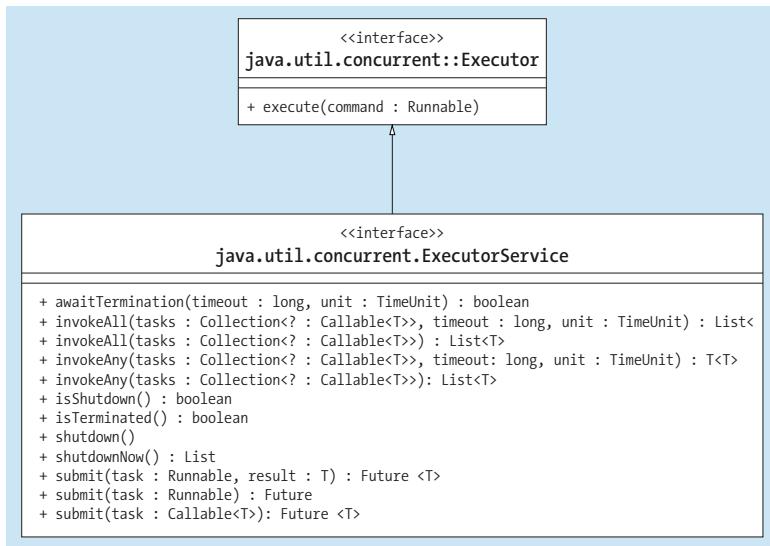


Abbildung 16.6 Die Schnittstelle ExecutorService, die Executor erweitert



### Tipp

Natürlich lassen sich auch eigene »Ausführer« schreiben, zum Beispiel einen, der ein Runnable von dem Swing-GUI-Thread ausführen lässt:

```
Executor executor = runnable -> SwingUtilities.invokeLater( runnable );
```

Beziehungsweise kürzer:

```
Executor executor = SwingUtilities::invokeLater;
```

### 16.4.2 Glücklich in der Gruppe – die Thread-Pools

Threads aufzubauen, vom Betriebssystem verwalten zu lassen und dann wieder abzubauen und aus den internen Tabellen herauszunehmen kostet Zeit. Wenn es bei einem Server zum Beispiel darum geht, eine Anfrage direkt ohne Verzögerung zu beantworten, kann die Zeit zum Aufbau eines Threads störend sein. Auch sind mit einem Thread Ressourcen wie Stack-Speicher verbunden, die vom Betriebssystem reserviert werden müssen.

Eine Optimierung besteht darin, Threads aufzubauen und lebendig im Pool zu halten. Es gibt dann drei Schritte:

1. Gibt es eine Anfrage zur Abarbeitung eines Runnable, wird ein freier Thread aus dem Pool genommen.
2. Der Thread arbeitet die `run()`-Methode vom Runnable ab.
3. Nach der Abarbeitung legt sich der Thread wieder in den Pool zurück.

Thread-Pools haben den weiteren Vorteil, dass sie die Auslastung und parallele Verarbeitung beschränken können. Verwaltet ein Thread-Pool nur eine feste Anzahl von Threads und sind alle Threads in Arbeit, kann das System neue Anfragen ablehnen oder zum Warten zwingen. Das beugt effektiv Denial-of-Service-Angriffen vor. Wenn zum Beispiel bei einem Webserver jede eingehende Verbindung einen Thread aufmacht, kann leicht ein Server mit Anfragen bombardiert werden.

### **Executors.newCachedThreadPool(...)**

Eine wichtige statische Methode der Klasse Executors ist `newCachedThreadPool(...)`. Dahinter verbirgt sich ein `ThreadPoolExecutor`-Konstruktor-Aufruf. Das Ergebnis ist ein `ExecutorService`-Objekt, eine Implementierung von `Executor` mit der Methode `execute(Runnable)`:

**Listing 16.8** src/main/java/com/tutego/insel/thread/concurrent/ThreadPoolDemo.java, main(), Teil 1

```
Runnable r1 = () -> {
    System.out.println( "1.1 " + Thread.currentThread().getName() );
    System.out.println( "1.2 " + Thread.currentThread().getName() );
};

Runnable r2 = () -> {
    System.out.println( "2.1 " + Thread.currentThread().getName() );
    System.out.println( "2.2 " + Thread.currentThread().getName() );
};
```

Jetzt lässt sich der Thread-Pool als `ExecutorService` beziehen und lassen sich die beiden Befehlsobjekte als `Runnable` über `execute(...)` ausführen:

**Listing 16.9** src/main/java/com/tutego/insel/thread/concurrent/ThreadPoolDemo.java, main(), Teil 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:

```
1.1 pool-1-thread-1
2.1 pool-1-thread-2
2.2 pool-1-thread-2
1.2 pool-1-thread-1
2.1 pool-1-thread-2
2.2 pool-1-thread-2
1.1 pool-1-thread-1
1.2 pool-1-thread-1
```

Am Namen des Threads ist abzulesen, dass hier zwei Threads von einem Thread-Pool »pool-1« verwendet werden: »thread-1« und »thread-2«. Nach dem Ausführen der beiden ersten Aufträge `r1` und `r2` und der kleinen Warterei sind die Threads »pool-1-thread-1« und »pool-1-thread-2« wieder frei, sodass `r1` und `r2` wieder von diesen beiden Threads abgearbeitet wird. 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 Runnables werden zum Stoppen angeregt. Die Rückgabe ist eine Liste der zu beenden Runnables.

### 16.4.3 Threads mit Rückgabe über Callable

Der nebenläufige Thread kann nur über Umwege Ergebnisse zurückgeben. In einer eigenen Klasse, die `Runnable` erweitert, lässt sich im Konstruktor zum Beispiel eine Datenstruktur übergeben, in die der Thread ein berechnetes Ergebnis hineinlegt. Die Datenstruktur kann dann vom Aufrufer auf Änderungen hin untersucht werden.

Die Java-Bibliothek bietet noch einen anderen Weg, denn während `run()` in `Runnable` als Rückgabe `void` hat, übermittelt `call()` einer anderen Schnittstelle `Callable` eine Rückgabe und kann auch eine Ausnahme auslösen. Zum Vergleich:

<pre>interface java.lang.Runnable</pre> <ul style="list-style-type: none"> <li>► void run()</li> </ul> <p>Diese Methode enthält den nebenläufig auszuführenden Programmcode.</p>	<pre>interface java.util.concurrent.Callable&lt;V&gt;</pre> <ul style="list-style-type: none"> <li>► V call() throws Exception</li> </ul> <p>Diese Methode enthält den nebenläufig auszuführenden Programmcode und liefert eine Rückgabe vom Typ V.</p>
--	---

**Tabelle 16.3** Methoden in Runnable und Callable**Abbildung 16.7** Die einfache Schnittstelle Callable mit einer Operation

### Beispiel: Felder sortieren über Callable

Wir wollen ein Beispiel implementieren, das ein Feld sortiert. Das Sortieren soll ein Callable im Hintergrund übernehmen. Ist die Operation beendet, soll der Verweis auf das sortierte Feld zurückgegeben werden. Das Sortieren erledigt wie üblich `Arrays.sort(...)`:

**Listing 16.10** src/main/java/com/tutego/insel/thread/concurrent/SorterCallable.java, SorterCallable

```

class SorterCallable implements Callable<byte[]> {

    private final byte[] b;

    SorterCallable( byte[] b ) {
        this.b = b;
    }

    @Override public byte[] call() {
        Arrays.sort( b );
        return b;
    }
}
  
```

Natürlich bringt es wenig, das Callable-Objekt aufzubauen und selbst `call()` aufzurufen, denn ein Thread soll die Aufgabe im Hintergrund erledigen. Dazu ist jedoch nicht die Klasse `Thread` selbst zu verwenden, sondern ein `ExecutorService`, den wir etwa über `Executors.newCachedThreadPool()` bekommen:

**Listing 16.11** src/main/java/com/tutego/insel/thread/concurrent/CallableGetDemo.java, main(), Ausschnitt

```
byte[] b = new byte[ 4000000 ];
new Random().nextBytes( b );
Callable<byte[]> c = new SorterCallable( b );
ExecutorService executor = Executors.newCachedThreadPool();
Future<byte[]> result = executor.submit( c );
```

Der ExecutorService bietet eine submit(Callable)-Methode, die unser Callable annimmt und einen Thread für die Abarbeitung aussucht. Die Rückgabe ist ein mysteriöses Future ...

### ExecutorService führt aus: Callable und ein Runnable mit Zukunft

Aus Gründen der Symmetrie gibt es neben submit(Callable) noch zwei submit(...)-Methoden, die ebenfalls ein Runnable annehmen. Zusammen ergeben sich:

```
interface java.util.concurrent.ExecutorService
extends Executor
```

- <T> Future<T> submit(Callable<T> task)  
Der ExecutorService soll die Aufgabe abarbeiten und Zugriff auf das Ergebnis über die Rückgabe geben.
- Future<?> submit(Runnable task)  
Der ExecutorService arbeitet das Runnable ab und ermöglicht es, über das Future-Objekt zu erfragen, ob die Ausgabe schon abgearbeitet wurde oder nicht. get() liefert am Ende null.
- <T> Future<T> submit(Runnable task, T result)  
Wie submit(task), nur: Die get(...)-Anfrage über Future liefert result.

Um ein Runnable in ein Callable umzuwandeln, gibt es noch einige Hilfsmethoden in der Klasse Executors. Dazu zählen die statische Methode callable(Runnable task), die ein Callable<Object> liefert, und die Methode callable(Runnable task, T result), die ein Callable<T> zurückgibt.

#### 16.4.4 Erinnerungen an die Zukunft – die Future-Rückgabe

Weil das Ergebnis asynchron ankommt, liefert submit(Callable) ein Future-Objekt zurück, über das wir herausfinden können, ob das Ergebnis schon da ist oder ob wir noch warten müssen. Eigentlich ist nach einem submit(...) die beste Zeit, noch andere nebenläufige Aufgaben anzustoßen, um dann später mit get(...) das Ergebnis einzusammeln. Das Programmiermuster ist immer gleich: Erst Arbeit an den ExecutorService übergeben, dann etwas anderes

machen und später zurückkommen. Da wir in unserem Beispiel jedoch in der Zwischenzeit nichts anderes zu tun haben, als ein Bytefeld zu sortieren, setzen wir das Callable ab und warten mit get() sofort auf das sortierte Feld:

```
Listing 16.12 src/main/java/com/tutego/insel/thread/concurrent/CallableGetDemo.java, main()
byte[] b = new byte[ 4000000 ];
new Random().nextBytes( b );
Callable<byte[]> c = new SorterCallable( b );
ExecutorService executor = Executors.newCachedThreadPool();
Future<byte[]> result = executor.submit( c );
// Jetzt werden erst einmal andere Dinge gemacht, und später ...
try {
    byte[] bs = result.get();
    System.out.printf( "%d, %d, %d%n",
                       bs[0], bs[1], bs[bs.length - 1] ); // -128, -128, 127
}
catch ( InterruptedException | ExecutionException e ) {
    e.printStackTrace();
}
```

Da das Feld sortiert ist und der Wertebereich eines Bytes mit –128 bis +127 sehr klein ist, ist vermutlich bei 4.000.000 Werten das kleinste Element der Zufallszahlen –128 und das größte 127.

Die Operationen der Schnittstelle Future im Einzelnen:

```
interface java.util.concurrent.Future<V>
```

- **V get()** throws InterruptedException, ExecutionException  
Wartet auf das Ergebnis und gibt es dann zurück. Die Methode blockiert so lange, bis das Ergebnis da ist. Es kann zu Ausnahmen kommen: CancellationException, wenn die Berechnung abgebrochen wurde, ExecutionException, wenn die Berechnung eine Ausnahme auslöste, InterruptedException, wenn der aktuelle Thread beim Warten unterbrochen wurde.
- **V get(long timeout, TimeUnit unit)**  
throws InterruptedException, ExecutionException, TimeoutException  
Wartet eine gegebene Zeit auf das Ergebnis und gibt es dann zurück. Kommt es in der vorgegebenen Dauer nicht, gibt es eine TimeoutException.
- **boolean isDone()**  
Wurde die Arbeit beendet oder sogar abgebrochen?

- `boolean cancel(boolean mayInterruptIfRunning)`  
Bricht die Arbeit ab.
- `boolean isCancelled()`  
Wurde die Arbeit vor dem Ende abgebrochen?

### Warten mit Zeitbeschränkung

Nicht immer ist das potenziell unendliche Blockieren erwünscht. Für diesen Fall ermöglicht die überladene Methode von `get(...)` eine Parametrisierung mit einer Wartezeit und Zeiteinheit:

**Listing 16.13** src/main/java/com/tutego/insel/thread/concurrent/  
CallableGetTimeUnitDemo.java, Ausschnitt

```
byte[] bs = result.get( 2, TimeUnit.SECONDS );
```

Ist das Ergebnis nicht innerhalb von 2 Sekunden verfügbar, löst die Methode eine `TimeoutException` aus, die so aussehen wird:

```
java.util.concurrent.TimeoutException
  at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:228)
  at java.util.concurrent.FutureTask.get(FutureTask.java:91)
  at com.tutego.insel.thread.concurrent.CallableDemo.main(CallableDemo.java:27)
```



#### Tipp

Wenn zwischen dem Absenden der Aufgabe und dem Abholen des Ergebnisses genügend Zeit vergeht, dass es beim Abholen zu keiner blockierenden Wartesituation kommt, ist das perfekt. Ungünstig ist es, wenn kurz nach dem `submit(...)` schon das `get(...)` kommt, aber das Ergebnis auf sich warten lässt – dann wäre nicht viel über das Future gewonnen. Eine interessante Lösung bietet eine Implementierung von Future, das CompletableFuture, das Aufgaben in eine Abfolge setzt. Die Idee ist einfach: Damit es zu keiner Wartezeit kommt, wird das Ergebnis, wenn es von einem Schritt berechnet wurde, zum nächsten Verarbeitungsschritt direkt weitergeleitet.

### Callable oder Runnable mit FutureTask ummanteln

Achten wir genau auf den Stack-Aufruf von eben, fällt der Typ `java.util.concurrent.FutureTask` ins Auge. Die Klasse implementiert Future, Runnable und RunnableFuture und wird intern von der Java-Bibliothek verwendet, wenn wir mit `submit(...)` etwas beim ExecutorService absetzen. Auch wir können den Typ direkt als Wrapper um ein Callable oder Runnable nutzen, denn es gibt praktische Callback-Methoden, die wir überschreiben können, etwa `done()`, wenn eine Berechnung fertig ist.

Dazu ein Beispiel: Ein Callable liefert den Namen des Benutzers. Ein FutureTask legt sich um dieses Callable und bekommt mit, wann das Callable fertig ist, und modifiziert dann den Benutzernamen und gibt weiterhin eine Meldung aus.

**Listing 16.14** src/main/java/com/tutego/insel/thread/concurrent/WrappedUsername.java,  
Ausschnitt

```
Callable<String> username = () -> System.getProperty( "user.name" );
FutureTask<String> wrappedUsername = new FutureTask<>( username ) {
    @Override protected void done() {
        try {
            System.out.printf( "done: isDone=%s, isCancelled=%s%n", isDone(), isCancelled() );
            System.out.println( "done: get=" + get() );
        }
        catch ( InterruptedException | ExecutionException e ) { /* Ignore */ }
    }
    @Override protected void set( String v ) {
        System.out.println( "set: " + v );
        super.set( v.toUpperCase() );
    }
};
ExecutorService scheduler = Executors.newCachedThreadPool();
scheduler.submit( wrappedUsername );
System.out.println( "main: " + wrappedUsername.get() );
scheduler.shutdown();
```

Wichtig in der Nutzung ist, nicht die Rückgabe vom submit(...) auszuwerten, was wir normalerweise machen, sondern das übergebene FutureTask zu erfragen.

Die Ausgaben vom Programm sind oft ein wenig durcheinander:

```
set: Christian
done: isDone=true, isCancelled=false
done: get=CHRISTIAN
main: CHRISTIAN
```

Die Reihenfolge in den Aufrufen ist immer so: Der FutureTask stellt die Fertigstellung vom Callable fest und ruft set(...) auf, anschließend done().

#### 16.4.5 Mehrere Callable-Objekte abarbeiten

Die Methode submit(Callable) vom ExecutorService nimmt genau ein Callable an und führt es aus:

- <T> Future<T> submit(Callable<T> task)

Muss eine Anwendung mehrere Callable-Objekte abarbeiten, kann es natürlich mehrere Aufrufe von `submit(Callable)` geben. Doch ein `ExecutorService` kann von sich aus mehrere Callable-Objekte abarbeiten. Dabei gibt es zwei alternative Varianten:

- ▶ Alle Callable-Objekte einer Liste werden ausgeführt, und das Ergebnis ist eine Liste von Future-Objekten.
- ▶ Alle Callable-Objekte einer Liste werden ausgeführt, doch das erste, das mit der Arbeit fertig wird, ergibt das Resultat.

Das ergibt zwei Methoden, und da sie zusätzlich mit einer Zeitbeschränkung ausgestattet werden, sind es vier:

```
interface java.util.concurrent.ExecutorService
extends Executor
```

- `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`  
throws `InterruptedException`  
Führt alle Aufgaben aus. Liefert eine Liste von Future-Objekten, die die Ergebnisse repräsentieren.
- `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` throws `InterruptedException`  
Führt alle Aufgaben aus und wird die Ergebnisse als Liste von Future-Objekten liefern, so lange die Zeit `timeout` in der gegebenen Zeiteinheit nicht überschritten wird.
- `<T> T invokeAny(Collection<? extends Callable<T>> tasks)`  
throws `InterruptedException, ExecutionException`  
Führt alle Aufgaben aus, aber liefert das Ergebnis eines Ausführers, der als Erster fertig ist. Ein `get(...)` wird also nie warten müssen.
- `<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`  
throws `InterruptedException, ExecutionException, TimeoutException`  
Führt alle Aufgaben aus, gilt aber nur für eine beschränkte Zeit. Das erste Ergebnis eines Callable-Objekts, das in der Zeit fertig wird, gibt `invokeAny(...)` zurück.



#### Tipp

Die `get(long timeout, TimeUnit unit)`-Methode von `Future` sendet dem Thread kein Interrupt, wenn er es nicht schafft, in der Zeit ein Ergebnis zu produzieren. Das ist der Vorteil der genannten `xxxAny(..., TimeUnit)`-Methoden, die ein Interrupt auslösen.

### 16.4.6 CompletionService und ExecutorCompletionService

Die invokeAll(...)-Methoden aus dem ExecutorService sind praktisch, wenn es darum geht, mehrere Aufgaben nebenläufig abzusenden und später die Ergebnisse einzusammeln. Allerdings ist die Rückgabe vom Typ `List<Future<T>>`, und wir werden nicht informiert, wenn ein Ergebnis vorliegt. Wir können zwar die Liste immer wieder ablaufen und jedes Future-Objekt mit `isDone()` fragen, ob es fertig ist, aber das ist keine ideale Lösung.

Mit `java.util.concurrent.CompletionService` gibt es eine weitere Java-Schnittstelle – die keinen Basistyp erweitert –, mit der wir ein Callable oder Runnable arbeiten lassen können und später nacheinander die Ergebnisse einsammeln können, die fertig sind. Die Java-Bibliothek bringt mit `ExecutorCompletionService` eine Implementierung der Schnittstelle mit, die intern die fertigen Ergebnisse in einer Queue sammelt, und wir können die Queue abfragen. Schauen wir uns das in einem Beispiel an.

**Listing 16.15** src/main/java/com/tutego/insel/thread/concurrent/ExecutorCompletionServiceDemo.java, Ausschnitt

```
ExecutorService executor = Executors.newCachedThreadPool();
CompletionService<Integer> completionService =
    new ExecutorCompletionService<>( executor );
List.of( 4, 3, 2, 1 ).forEach( duration -> completionService.submit( () -> {
    TimeUnit.SECONDS.sleep( duration );
    return duration;
} ) );

for ( int i = 0; i < 4; i++ ) {
    try {
        System.out.println( completionService.take().get() );
    }
    catch ( InterruptedException | ExecutionException e ) {
        e.printStackTrace();
    }
}

executor.shutdown();
```

Der Typ `ExecutorCompletionService` erwartet im Konstruktor einen Executor, der den Code ausführen soll; wir setzen einen Thread-Pool ein. `CompletionService` hat zwei `submit(...)`-Methoden:

- `Future<V> submit(Runnable task, V result)`
- `Future<V> submit(Callable<V> task)`

Abgesendet werden vier Callable-Exemplare, die 4, 3, 2, 1 Sekunden warten und ihre Wartezeit am Ende zurückgeben. Natürlich wird als Erstes das Callable mit der Rückgabe »1« fertig, dann 2 usw.

Für die Rückgaben interessiert sich unser Programm nicht, denn es nutzt die `take()`-Methode. Insgesamt hat `CompletionService` drei Entnahme-Methoden:

- `Future<V> take()`  
Liefert das Ergebnis von der ersten abgeschlossenen Aufgabe und entfernt es von der internen Queue. Liegt kein Ergebnis an, wartet die Methode.
- `Future<V> poll()`  
Liefert das Ergebnis von der ersten abgeschlossenen Aufgabe und entfernt es von der internen Queue. Liegt kein Ergebnis an, wartet `poll()` nicht, sondern liefert `null`.
- `Future<V> poll(long timeout, TimeUnit unit)`  
Wartet wie `take()` auf ein Ergebnis, doch kommt dieses nach dem Ablauf von `timeout` nicht, liefert die Methode wie `poll()` als Rückgabe `null`.

Was der Schnittstelle fehlt, ist eine Methode, die die verblendende Anzahl liefert. Wir müssen in unserem Code daher einen Zähler als extra Variable einführen.

#### 16.4.7 ScheduledExecutorService für wiederholende Aufgaben und Zeitsteuerungen nutzen

Die Klasse `ScheduledThreadPoolExecutor` ist eine weitere Klasse neben `ThreadPoolExecutor`, die die Schnittstellen `Executor` und `ExecutorService` implementiert. Die wichtige Schnittstelle, die diese Klasse außerdem implementiert, ist aber `ScheduledExecutorService`, ein direkter Untertyp von `ExecutorService`. Hier sind mehrere `scheduleXXX(...)`-Operationen deklariert, die ein `Runnable` oder `Callable` zu bestimmten Zeiten und Wiederholungen ausführen. (Zwar gibt es mit dem `java.util.Timer` etwas Ähnliches, doch der `ScheduledThreadPoolExecutor` nutzt Threads aus dem Pool.)

`Executors` bietet mehrere statische Methoden, die uns fertig konfigurierte `ScheduledExecutorService`-Objekte liefern, zum Beispiel `newScheduledThreadPool(int corePoolSize)`.

Das folgende Beispiel führt nach einer Startzeit von einer Sekunde alle zwei Sekunden eine Ausgabe aus:

```
Listing 16.16 src/main/java/com/tutego/insel/thread/concurrent/
ScheduledExecutorServiceDemo.java. main()
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( 1 );
scheduler.scheduleAtFixedRate(
    () -> System.out.println( "Tata" ),
    1 /* initial delay */,
```

```
2 /* period */,  
TimeUnit.SECONDS );
```

Nach einer Sekunde Startverzögerung bekommen wir jede zweite Sekunde ein »Tata«.

## 16.5 Zum Weiterlesen

Der Band »Java SE 9 Standard-Bibliothek« geht detaillierter auf die API ein.



# Kapitel 17

## 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 eine Reihe von schwierigen Problemen in der Informatik ist wohl deswegen noch nicht gelöst, weil eine passende Datenorganisation bis jetzt nicht gefunden wurde.

Die wichtigsten Datenstrukturen, wie Listen, Mengen, Kellerspeicher und Assoziativspeicher, sollen in diesem Kapitel vorgestellt werden.

### 17.1 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 Arrays
- ▶ `java.util.LinkedList`: Liste durch verkettete Elemente
- ▶ `java.util.concurrent.CopyOnWriteArrayList`: schnelle Liste, optimal für häufige nebelaufige Lesezugriffe

---

<sup>1</sup> Das Wort *Algorithmus* geht auf den Namen des persisch-arabischen Mathematikers Ibn Mûsâ Al-Chwârimî zurück, der im 9. Jahrhundert lebte.

- ▶ `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.

### 17.1.1 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 17.1** src/main/java/com/tutego/insel/util/list/MemorizeYourRoadTripRoute.java

```
package com.tutego.insel.util.list;

import java.text.*;
import java.util.*;

public class MemorizeYourRoadTripRoute {
    @SuppressWarnings( "resource" )
    public static void main( String[] args ) {
        List<String> cities = new ArrayList<>();

        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!" );  
}  
}
```

### 17.1.2 Auswahlkriterium ArrayList oder LinkedList

Eine `ArrayList` (das Gleiche gilt für `Vector`) speichert Elemente in einem internen Array. `LinkedList` dagegen speichert die Elemente in einer verketteten Liste und realisiert die Verkettung mit einem eigenen Hilfsobjekt für jedes Listenelement. Es ergeben sich Einsatzgebiete, die einmal für `LinkedList` und einmal für `ArrayList` sprechen:

- ▶ Da `ArrayList` intern ein Array benutzt, ist der Zugriff auf ein spezielles Element über die Position in der Liste sehr schnell. Eine `LinkedList` muss aufwändiger durchsucht werden, und dies kostet Zeit.
  - ▶ Die verkettete Liste ist aber deutlich im Vorteil, wenn Elemente mitten in der Liste gelöscht oder eingefügt werden; hier muss einfach nur die Verkettung der Hilfsobjekte an einer Stelle verändert werden. Bei einer `ArrayList` bedeutet dies viel Arbeit, es sei denn, das Element kann am Ende gelöscht oder – bei ausreichender Puffergröße – eingefügt werden. Soll ein Element nicht am Ende eingefügt oder gelöscht werden, müssen alle nachfolgenden Listenelemente verschoben werden.
  - ▶ Bei einer `ArrayList` kann die Größe des internen Arrays zu klein werden. Dann bleibt der Laufzeitumgebung nichts anderes übrig, als ein neues, größeres Array-Objekt anzulegen und alle Elemente zu kopieren.

### 17.1.3 Die Schnittstelle List

Die Schnittstelle `List` schreibt das allgemeine Verhalten für Listen vor. Die allermeisten Methoden kennen wir schon vom `Collection`-Interface, und zwar deshalb, weil `List` die Schnittstelle `Collection` erweitert. Hinzugekommen sind Methoden, die einen Bezug zur Position eines Elements haben – Mengen, die auch `Collection` implementieren, kennen keinen Index.

## Hinzufügen und Setzen von Elementen

Die `add(...)`-Methode fügt neue Elemente an die Liste an, wobei eine Position die Einfügestellen bestimmen kann. Die Methode `addAll(...)` fügt fremde Elemente einer anderen Sammlung in die Liste ein. `set(...)` setzt ein Element an eine bestimmte Stelle, überschreibt das ursprüngliche Element und verschiebt es auch nicht wie `add(...)`. Die Methode `size()` nennt die Anzahl der Elemente in der Datenstruktur:

**Listing 17.2** src/main/java/com/tutego/insel/util/list/ListDemo.java, Ausschnitt

```
List<String> list1 = new ArrayList<>();
list1.add( "Eva" );
list1.add( 0, "Charisma" );
list1.add( "Pallas" );

List<String> list2 = Arrays.asList( "Tina", "Wilhelmine" );
list1.addAll( 3, list2 );
list1.add( "XXX" );
list1.set( 5, "Eva" );

System.out.println( list1 );           // [Charisma, Eva, Pallas, Tina, Wilhelmine, Eva]
System.out.println( list1.size() ); // 6
```

## Positionsanfragen und Suchen

Ob die Sammlung leer ist, bestimmt `isEmpty()`. Ein Element an einer speziellen Stelle erfragen kann `get(int)`. Ob Elemente Teil der Sammlung sind, beantworten `contains(...)` und `containsAll(...)`. Wie bei Strings liefern `indexOf(...)` und `lastIndexOf(...)` die Fundpositionen:

**Listing 17.3** src/main/java/com/tutego/insel/util/list/ListDemo.java, Ausschnitt

```
boolean b = list1.contains( "Tina" );
System.out.println( b );           // true

b = list1.containsAll( Arrays.asList( "Tina", "Eva" ) );
System.out.println( b );           // true

Object o = list1.get( 1 );
System.out.println( o );           // Eva

int i = list1.indexOf( "Eva" );
System.out.println( i );           // 1
```

```
i = list1.lastIndexOf( "Eva" );
System.out.println( i );           // 5

System.out.println( list1.isEmpty() ); // false
```

### Listen zu Arrays und neue Listen bilden

Von den Listen können Arrays abgeleitet werden und sich Schnittmengen bilden lassen:

**Listing 17.4** src/main/java/com/tutego/insel/util/list/ListDemo.java, Ausschnitt

```
String[] array = list1.toArray( new String[list1.size()] );
// alternativ: String[] array = list1.toArray( String[]::new );

System.out.println( array[3] );      // "Tina"

List<String> list3 = new LinkedList<>( list1 );
System.out.println( list3 );        // [Charisma, Eva, Pallas, Tina,
                                  // Wilhelmine, Eva]
list3.retainAll( Arrays.asList( "Tina", "Eva" ) );
System.out.println( list3 );        // [Eva, Tina, Eva]
```

### Löschen von Elementen

Außerdem gibt es Methoden zum Löschen von Elementen. Hier bietet die Liste eine überlappende `remove(...)`-Methode, `removeIf(...)` und `removeAll(...)`. Den kürzesten Weg, alles aus der Liste zu löschen, bietet `clear()`:

**Listing 17.5** src/main/java/com/tutego/insel/util/list/ListDemo.java, Ausschnitt

```
System.out.println( list1 );      // [Charisma, Eva, Pallas, Tina, Wilhelmine, Eva]
list1.remove( 1 );
System.out.println( list1 );      // [Charisma, Pallas, Tina, Wilhelmine, Eva]

list1.remove( "Wilhelmine" );
System.out.println( list1 );      // [Charisma, Pallas, Tina, Eva]

list1.removeAll( Arrays.asList( "Pallas", "Eva" ) );
System.out.println( list1 );      // [Charisma, Tina]

list1.clear();
System.out.println( list1 );      // []
```



### Hinweis

Die Methode `remove(int)` löscht ein Element an der gegebenen Stelle, `remove(Object)` sucht einmal nach einem `equals`-gleichen Objekt und löscht es dann, würde aber nicht weitersuchen nach anderen Vorkommen. `removeIf(...)` und `removeAll(...)` laufen immer komplett über die ganze Datenstruktur und schauen, ob ein Element dem Kriterium genügt, um es zu löschen, was mehrmals vorkommen kann.

#### Beispiel:

Lösche alle null-Referenzen und Weißraum-Strings aus der Liste:

```
List<String> list = new ArrayList<>();
Collections.addAll( list, "1", "", " ", "zwei", null, "Polizei" );
list.removeIf( e -> Objects.isNull( e ) || e.trim().isEmpty() );
System.out.println( list ); // [1, zwei, Polizei]
```

### Zusammenfassung

Die Methoden der Schnittstelle `List` (inklusive der aus der erweiterten Schnittstelle `Collection`) 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.
- `static <E> List<E> copyOf(Collection<? extends E> coll)`  
Erzeugt eine unmodifizierbare Kopie.

- `E get(int index)`  
Liefert das Element an der angegebenen Stelle der Liste.
- `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, die ab einer bestimmten Stelle weitersucht, 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.
- `default boolean removeIf(Predicate<? super E> filter)`  
Entfernt alle Elemente aus der Liste, bei denen das Prädikat erfüllt ist.
- `boolean retainAll(Collection<?> c)`  
Optional. Entfernt alle Objekte aus der Liste, die nicht in der Collection `c` vorkommen.
- `default void replaceAll(UnaryOperator<E> operator)`  
Ruft auf jedem Element den Operator auf und schreibt das Ergebnis zurück.
- `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).
- `default void sort(Comparator<? super E> c)`  
Sortiert die Liste, entspricht `Collections.sort(this, c)`.
- `boolean equals(Object o)`  
Vergleicht die Liste mit einer anderen Liste. Zwei Listenobjekte sind gleich, wenn ihre Elemente paarweise gleich sind.
- `int hashCode()`  
Liefert den Hashwert der Liste.

Was `List` der `Collection` hinzufügt, sind also die indexbasierten 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)`, zudem die beiden Default-Methoden `replaceAll(...)` und `sort(...)`.



### Tipp

Die `remove(...)`-Methode ist überschrieben und löscht a) einmal ein Element, das mit `equals(...)` gesucht wird, und b) mit `remove(int)` ein Element an der gegebenen Position. Irrigerend ist Folgendes:

```
list = new ArrayList<>( Arrays.asList( 9, 8, 1, 7 ) );
Integer index = 1;
list.remove( index );
System.out.println( list ); // [9, 8, 7]
```

Es wird `remove(Object)` aufgerufen, weil `Object` dem Argumenttyp `Integer` am ähnlichsten ist. Somit verschwindet das Element `Integer.valueOf(1)` aus der Liste. Unboxing findet nicht statt. Das dürfte zu erwarten sein, und daher sind die Namensgebung `index` und der Objekttyp im Beispiel bewusst irreführend. Gemeint war natürlich:

```
list = new ArrayList<>( List.of( 9, 8, 1, 7 ) );
int realIndex = 1;
list.remove( realIndex );
System.out.println( list ); // [9, 1, 7]
```

## Kopieren und Ausschneiden

Die Listen-Klassen implementieren `clone()` und erzeugen eine flache Kopie.

Um einen Bereich zu löschen, nutzen wir `subList(from, to).clear()`. Die `subList`-Technik deckt gleich noch einige andere Operationen ab, für die es keine speziellen Range-Varianten gibt, zum Beispiel `indexOf(...)`, also die Suche in einem Teil der Liste.

### Beispiel

Baue eine Liste auf, kürze sie, und gib die Elemente rückwärts aus:

```
List<String> list = new ArrayList<>()
    Arrays.asList( "0 1 2 3 4 5 6 7 8 9".split( " " ) );
list.subList( 2, list.size() - 2 ).clear();
System.out.println( list );                                // [0, 1, 8, 9]
for ( ListIterator<String> it = list.listIterator( list.size() );
      it.hasPrevious(); )
    System.out.print( it.previous() + " " );     // 9 8 1 0
```



`subList(...)` erzeugt wie viele andere Methoden der Collection-Datenstrukturen eine Ansicht auf die Liste, was bedeutet, dass Änderungen an dieser Teilliste zu Änderungen des Originals führen. Das gilt auch für `clear()`, das dazu genutzt werden kann, einen Teilbereich der Originalliste zu löschen.

### Tipp



Die zum Löschen naheliegende Methode `removeRange(int, int)` kann nicht (direkt) eingesetzt werden, da sie `protected`<sup>2</sup> ist. Das lässt sich zum Beispiel wie folgt beheben:

```
List<gewünschterTyp> list = new ArrayList<gewünschterTyp>() {
    @Override public void removeRange( int fromIndex, int toIndex ) {
        super.removeRange( fromIndex, toIndex );
    }
};
```

<sup>2</sup> In `AbstractList` ist `removeRange(int, int)` gültig mit einem `ListIterator` implementiert, also nicht abstrakt. Die API-Dokumentation begründet das damit, dass `removeRange(...)` nicht zur offiziellen Schnittstelle von Listen gehört, sondern für die Autoren neuer Listenimplementierungen gedacht ist.

### 17.1.4 ArrayList

Jedes Exemplar der Klasse `ArrayList` vertritt ein Array mit variabler Länge. Der Zugriff auf die Elemente erfolgt effizient über Indizes, was `ArrayList` über die Implementierung der Markierungsschnittstelle `RandomAccess` andeutet.

#### Eine `ArrayList` erzeugen

Um ein `ArrayList`-Objekt zu erzeugen, existieren drei Konstruktoren:

```
class java.util.ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

- `ArrayList()`  
Eine leere Liste mit einer Anfangskapazität von zehn Elementen wird angelegt. Werden mehr als zehn Elemente eingefügt, muss sich die Liste vergrößern.
- `ArrayList(int initialCapacity)`  
Eine Liste mit der internen Größe von `initialCapacity`-vielen Elementen wird angelegt.
- `ArrayList(Collection<? extends E> c)`  
Kopiert alle Elemente der Collection `c` in das neue `ArrayList`-Objekt.

#### Die interne Arbeitsweise von `ArrayList` und `Vector` \*

Die Klassen `ArrayList` und `Vector` verwalten zwei Größen: zum einen die Anzahl der gespeicherten Elemente nach außen, zum anderen die interne Größe des Arrays. Ist die Kapazität des Arrays größer als die Anzahl der Elemente, so können noch Elemente aufgenommen werden, ohne dass die Liste etwas unternehmen muss. Die Anzahl der Elemente in der Liste, die Größe, liefert die Methode `size()`; die Kapazität des darunterliegenden Arrays liefert `capacity()`.

Die Liste vergrößert sich automatisch, falls mehr Elemente aufgenommen werden, als ursprünglich am Platz vorgesehen waren. Diese Operation heißt *Resizing*. Dabei spielt die Größe `initialCapacity` für effizientes Arbeiten eine wichtige Rolle. Sie sollte passend gewählt sein. Betrachten wir daher zunächst die Funktionsweise der Liste, falls das interne Array zu klein ist.

Wenn das Array zehn Elemente fasst, nun aber ein elftes eingefügt werden soll, so muss das Laufzeitsystem einen neuen Speicherbereich reservieren und jedes Element des alten Arrays in das neue kopieren. Das kostet Zeit. Schon aus diesem Grund sollte der Konstruktor `ArrayList(int initialCapacity)` oder `Vector(int initialCapacity)` gewählt werden, weil dieser eine Initialgröße festsetzt. Das Wissen über unsere Daten hilft dann der Datenstruktur. Falls

kein Wert voreingestellt wurde, so werden zehn Elemente angenommen. In vielen Fällen ist dieser Wert zu klein.

Nun haben wir zwar darüber gesprochen, dass ein neues Array angelegt wird und die Elemente kopiert werden, wir haben aber nichts über die Größe des neuen Arrays gesagt. Hier gibt es Strategien wie die »Verdopplungsmethode« beim `Vector`. Wird er vergrößert, so ist das neue Array doppelt so groß wie das alte. Dies ist eine Vorgehensweise, die für kleine und schnell wachsende Arrays eine clevere Lösung darstellt, großen Arrays jedoch schnell zum Verhängnis werden kann. Für den Fall, dass wir die Vergrößerung selbst bestimmen wollen, nutzen wir den Konstruktor `Vector(int initialCapacity, int capacityIncrement)`, der die Verdopplung ausschaltet und eine fixe Vergrößerung befiehlt. Die `ArrayList` verdoppelt nicht, sie nimmt die neue Größe mal 1,5. Bei ihr gibt es nicht das `capacityIncrement` im Konstruktor.

### Die Größe eines Arrays \*

Die interne Größe des Arrays kann mit `ensureCapacity(int)` geändert werden. Ein Aufruf von `ensureCapacity(int minimumCapacity)` bewirkt, dass die Liste insgesamt mindestens `minimumCapacity` Elemente aufnehmen kann, ohne dass ein Resizing nötig wird. Ist die aktuelle Kapazität der Liste kleiner als `minimumCapacity`, so wird mehr Speicher angefordert. Der Vektor verkleinert die aktuelle Kapazität nicht, falls sie schon höher als `minimumCapacity` ist. Um aber auch diese Größe zu ändern und somit ein nicht mehr wachsendes Vektor-Array so groß wie nötig zu machen, gibt es, ähnlich wie beim `String` mit Weißraum, die Methode `trimToSize()`. Sie reduziert die Kapazität des Vektors auf die Anzahl der Elemente, die gerade in der Liste sind. Mit `size()` lässt sich die Anzahl der Elemente in der Liste erfragen.

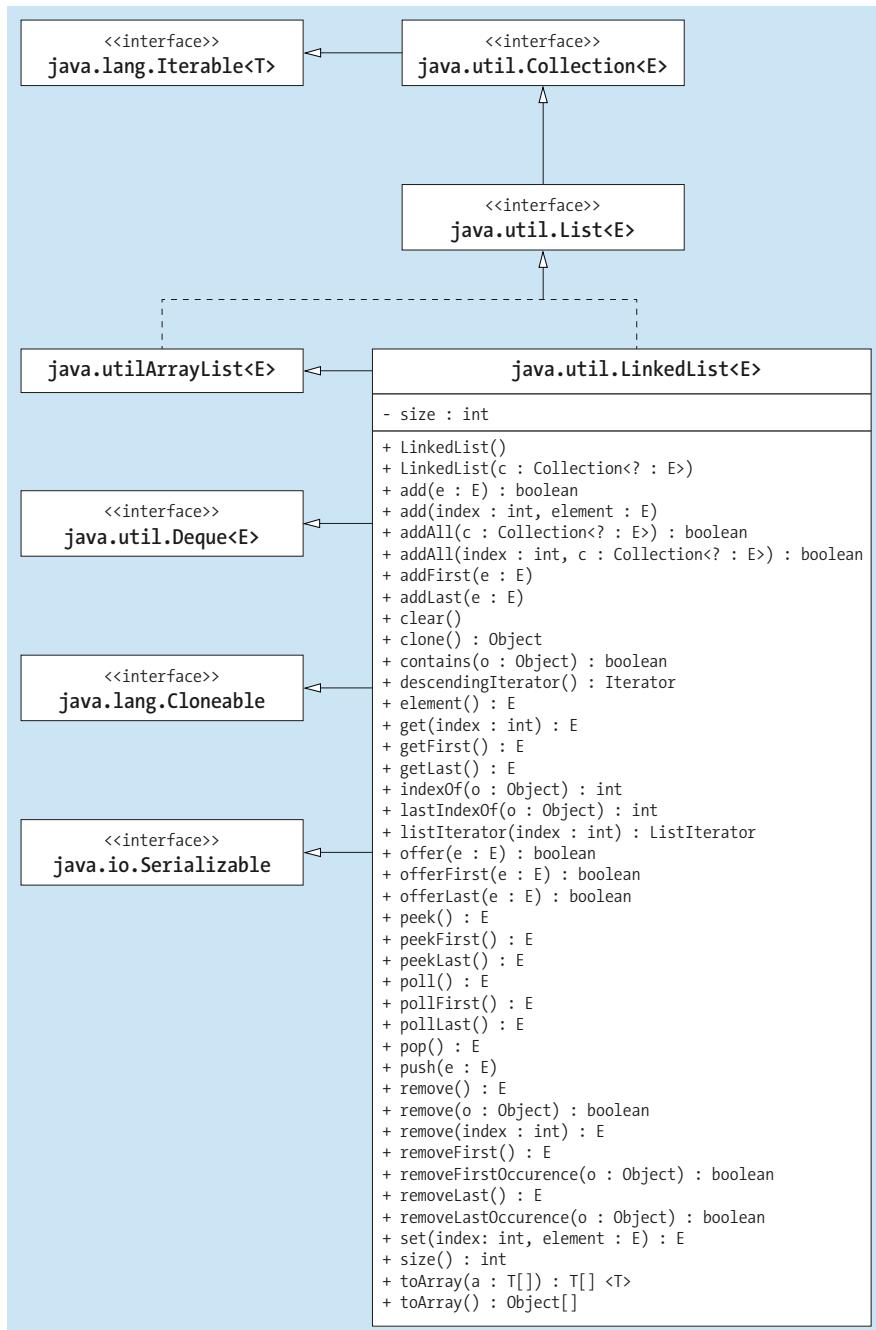
Bei der Klasse `Vector` lässt sich mit `setSize(int newSize)` auch die Größe der Liste verändern. Ist die neue Größe kleiner als die alte, werden die Elemente am Ende des Vektors abgeschnitten. Ist `newSize` größer als die alte Größe, werden die neu angelegten Elemente mit `null` initialisiert.<sup>3</sup> Vorsicht ist bei `newSize == 0` geboten, denn `setSize(0)` bewirkt das Gleiche wie `removeAllElements()`.

#### 17.1.5 LinkedList

Die Klasse `LinkedList` realisiert die Schnittstelle `List` als verkettete Liste und bildet die Elemente nicht auf ein Array ab. Die Implementierung realisiert die `LinkedList` als doppelt verkettete Liste, in der jedes Element – die Ränder lassen wir außen vor – einen Vorgänger und Nachfolger hat. (Einfach verkettete Listen haben nur einen Nachfolger, was die Navigation in beide Richtungen schwierig macht.)

---

<sup>3</sup> Zudem können `null`-Referenzen ganz normal als Elemente eines Vektors auftreten, bei den anderen Datenstrukturen gibt es Einschränkungen.

Abbildung 17.1 Klassendiagramm von `LinkedList` mit Vererbungsbeziehungen

Eine `LinkedList` hat neben den gegebenen Operationen aus der Schnittstelle `List` weitere Hilfsmethoden: Dabei handelt es sich um die Methoden `addFirst(...)`, `addLast(...)`, `getFirst()`,

`getLast()`, `removeFirst()` und `removeLast()`. Die implementierten Schnittstellen `Queue` und `Deque` sind nicht ganz unschuldig an diesen Methoden.

```
class java.util.LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

- `LinkedList()`  
Erzeugt eine neue leere Liste.
- `LinkedList(Collection<? extends E> c)`  
Kopiert alle Elemente der Collection `c` in die neue verkettete Liste.

### 17.1.6 Der Array-Adapter `Arrays.asList(...)`

Arrays von Objektreferenzen und dynamische Datenstrukturen passen nicht so richtig zusammen, obwohl sie schon häufiger zusammen benötigt werden. Die Java-Bibliothek bietet mit der statischen Methode `Arrays.asList(...)` an, ein existierendes Array als `java.util.List` zu behandeln. Der Parametertyp ist ein Vararg – das ja intern auf ein Array abgebildet wird –, sodass sich `asList(...)` auf zwei Arten verwenden lässt:

- ▶ `Arrays.asList(array)`: Die Variable `array` ist eine Referenz auf ein Array, und das Ergebnis ist eine Liste, die die gleichen Elemente wie das Array enthält.
- ▶ `Arrays.asList(e1, e2, e3)`: Die Elemente `e1, e2, e3` sind Elemente der Liste.

Das Entwurfsmuster, das die Java-Bibliothek bei der statischen Methode anwendet, nennt sich *Adapter*. Er passt die Schnittstelle eines Typs an eine andere Schnittstelle eines anderen Typs an.

#### Beispiel

[zB]

Ermittle die Anzahl der »:-)«-Smileys im String.

```
String s = "Oten :-) Bilat :-) Iyot";
int i = Collections.frequency( Arrays.asList(s.split("\\s")), ":" );
System.out.println( i ); // 2
```

In String gibt es keine Methode zum Zählen von Teil-Strings.

#### Beispiel

[zB]

Gib das größte Element eines Arrays aus:

```
Integer[] ints = { 3, 9, -1, 0 };
System.out.println( Collections.max( Arrays.asList( ints ) ) );
```

Zum Ermitteln des Maximums bietet die Utility-Klasse `Arrays` keine Methode, daher bietet sich die `max(...)`-Methode von `Collections` an. Auch etwa zum Ersetzen von Array-Elementen bietet `Arrays` nichts, aber `Collections`. Sortieren und füllen kann `Arrays` aber schon, hier muss `asList()` nicht einspringen.

### `class java.util.Arrays`

- `static <T> List<T> asList(T... a)`

Ermöglicht es, über die Schnittstelle `List` Zugriff auf ein Array zu erhalten. Die variablen Argumente sind sehr praktisch.



### Hinweis

Wegen der Generics ist der Parametertyp von `asList(...)` ein Objekt-Array, aber niemals ein primitives Array. In unserem Beispiel von eben würde so etwas wie

```
int[] ints = { 3, 9, -1, 0 };
Arrays.asList( ints );
```

zwar compilieren, aber die Rückgabe von `Arrays.asList(ints)` ist vom Typ `List<int[]>`, was bedeutet, die gesamte Liste besteht aus genau einem Element, und dieses Element ist das primitive Array. Zum Glück führt `Collections.max(Arrays.asList(ints))` zu einem Compilerfehler, denn von einer `List<int[]>`, also einer Liste von `Arrays`, kann `max(Collection<? extends T>)` kein Maximum ermitteln. Anders wäre das bei `Arrays.asList(3, 9, -1, 0)`, denn hier konvertiert der Compiler die Varargs-Argumente über Autoboxing schon direkt in Wrapper-Objekte, und es kommt eine Liste von `Integer`-Objekten heraus.

### Internes

Die Rückgabe von `asList(...)` ist kein konkreter Klassentyp wie `ArrayList` oder `LinkedList`, sondern irgendetwas Unbekanntes, was `asList(...)` als `List` herausgibt. Diese Liste ist nur eine andere Ansicht des Arrays.

Jetzt gibt es allerdings einen Interpretationsspielraum, was genau mit der Rückgabe möglich ist. Zudem ist es nicht ganz uninteressant, zu wissen, ob die Liste einen schnellen Punktzugriff zulässt (`RandomAccess` implementiert) bzw. ob optionale Operationen wie Veränderungen oder sogar totale Reorganisationen denkbar sind. Ein Blick auf die Implementierung verrät mehr. Das Ergebnis ist ein Adapter, der Listen-Methoden wie `get(index)` oder `set(index, element)` direkt auf das Array umleitet. Da Array-Längen `final` sind, führen Modifikationsmethoden wie `remove(...)` oder `add(...)` zu einer `UnsupportedOperationException`.



### Tipp

Sind Veränderungen an der `asList(...)`-Rückgabe erwünscht, so muss das Ergebnis in eine neue Datenstruktur kopiert werden, etwa so:

```
List<String> list = new ArrayList<>( Arrays.asList( "A", "B" ) );
list.add( "C" );
```

### 17.1.7 ListIterator \*

Die Schnittstelle `ListIterator` ist eine Erweiterung von `Iterator`. Diese Schnittstelle fügt noch Methoden hinzu, damit an der aktuellen Stelle auch Elemente eingefügt werden können. Die Stelle wird auch *Cursor* genannt. Es ist zu berücksichtigen, dass der Cursor vom Iterator immer zwischen den Elementen steht; am Anfang steht er vor dem ersten Element. Die Methoden `remove()` und `set(e)` sind nicht mit Cursor-Positionen verbunden; sie operieren auf dem letzten Element, das `next()` oder `previous()` liefert hat.

**Listing 17.6** src/main/java/com/tutego/insel/util/list/ListIteratorDemo.java, main()

```
List<String> list = new ArrayList<>();
Collections.addAll( list, "b", "c", "d" );

ListIterator<String> it = list.listIterator();

it.add( "a" );                                // Vorne anfügen
System.out.println( list );                  // [a, b, c, d]

it.next();                                    // Position vor
it.remove();                                  // Element löschen
System.out.println( list );                  // [a, c, d]

it.next();                                    // Position vor
it.set( "C" );                                // Element ersetzen
System.out.println( list );                  // [a, C, d]

it = list.listIterator( 1 );                // Neuen Iterator mit Startpos. 1
it.add( "B" );                                // B hinzufügen
System.out.println( list );                  // [a, B, C, d]

it = list.listIterator( list.size() );
it.previous();                                // Eine Stelle nach vorne
it.remove();                                  // Letztes Element löschen
System.out.println( list );                  // [a, B, C]
```

**Tipp**

Ein ListIterator kann die Elemente auch rückwärts verarbeiten:

```
List<String> list = new ArrayList<String>();
Collections.addAll( list, "1", "2", "3", "4" );
for ( ListIterator<String> it = list.listIterator( list.size() );
      it.hasPrevious(); )
    System.out.print( it.previous() + " " ); // 4 3 2 1
```

```
interface java.util.ListIterator<E>
extends Iterator<E>
```

- `boolean hasPrevious()`
- `boolean hasNext()`  
Liefert true, wenn es ein vorhergehendes/nachfolgendes Element gibt.
- `E previous()`
- `E next()`  
Liefert das vorangehende/nächste Element der Liste oder NoSuchElementException, wenn es das Element nicht gibt. Setzt dann die Cursorposition zurück bzw. weiter.
- `int previousIndex()`
- `int nextIndex()`  
Liefert den Index des vorhergehenden/nachfolgenden Elements. Geht previousIndex() vor die Liste, so liefert die Methode die Rückgabe »-1«. Geht nextIndex() hinter die Liste, liefert die Methode die Länge der gesamten Liste.
- `void remove()`  
Optional. Entfernt das letzte von `next()` oder `previous()` zurückgegebene Element.
- `void add(E o)`  
Optional. Fügt ein neues Objekt vor der aktuellen Position in die Liste ein.
- `void set(E o)`  
Optional. Ersetzt das Element, das `next()` oder `previous()` als Letztes zurückgegeben haben.

### 17.1.8 `toArray(...)` von Collection verstehen – die Gefahr einer Falle erkennen

Die `toArray()`-Methode aus der Schnittstelle `Collection` gibt laut Definition ein Array von Objekten zurück. Es ist wichtig, zu verstehen, welchen Typ die Einträge und das Array selbst haben. Eine Implementierung der `Collection`-Schnittstelle ist `ArrayList`.



## Beispiel

Diese Anwendung von `toArray()` überträgt alle Punkte der Sammlung in ein neues Array:

```
ArrayList<Point> list = new ArrayList<>();
list.add( new Point(13, 43) );
list.add( new Point(9, 4) );
Object[] points = list.toArray();
```

Wir erhalten ein Array mit Referenzen auf `Point`-Objekte, können jedoch nicht einfach `points[1].x` schreiben, um auf das Attribut des `Point`-Exemplars zuzugreifen, denn das Array `points` hat den deklarierten Elementtyp `Object`. Es fehlt die explizite Typumwandlung, und erst `((Point)points[1]).x` ist korrekt.

Vielleicht kommen wir spontan auf die Idee, einfach den Typ des Arrays in `Point[]` zu ändern; in dem Array befinden sich ja schließlich `Point`-Exemplare:

```
Point[] points = list.toArray(); // ☠ Compilerfehler
```

Doch damit meldet der Compiler den Fehler »Type mismatch: cannot convert from `Object[]` to `Point[]`«, weil der Rückgabewert von `toArray()` ein `Object[]` ist und kein `Point[]`. Jetzt können wir auf die Idee kommen, das mit einer Typumwandlung in ein `Point[]`-Objekt zu reparieren:

```
Point[] points = (Point[]) list.toArray(); // ☠ Problem zur Laufzeit
```

Zwar haben wir zur Übersetzungszeit kein Problem mehr, aber zur Laufzeit wird es immer knallen mit einer »`java.lang.ClassCastException: class [Ljava.lang.Object; cannot be cast to class [Ljava.awt.Point;`«, auch wenn sich im Array tatsächlich nur `Point`-Objekte befinden.

Diesen Programmierfehler müssen wir verstehen. Was wir falsch gemacht haben, ist einfach: Wir haben den Typ des Arrays mit den Typen der Array-Elemente durcheinandergebracht. Einem Array von Objektreferenzen können wir alles zuweisen:

```
Object[] os = new Object[ 3 ];
os[0] = new Point();
os[1] = "Trecker fahr'n";
os[2] = LocalDate.now();
```

Wir merken, dass der Typ des Arrays `Object[]` ist, und die Array-Elemente sind ebenfalls vom Typ `Object`. Hinter dem Schlüsselwort `new`, das das Array-Objekt erzeugt, steht der gemeinsame Obertyp für zulässige Array-Elemente. Bei `Object[]`-Arrays dürfen die Elemente Referenzen für beliebige Objekte sein. Klar ist, dass ein Array nur Objektreferenzen aufnehmen kann, die mit dem Typ für das Array selbst kompatibel sind, sich also auf Exemplare der angegebenen Klasse beziehen oder auf Exemplare von Unterklassen dieser Klasse:

```

/* 1 */ Object[] os = new Point[ 3 ];
/* 2 */ os[0] = new Point();
/* 3 */ os[1] = LocalDate.now();      // 💀 ArrayStoreException
/* 4 */ os[2] = "Trecker fahr'n";   // 💀 ArrayStoreException

```

Zeilen 3 und 4 sind vom Compiler erlaubt, führen aber zur Laufzeit zu einer `ArrayStoreException`.

Kommen wir wieder zur Methode `toArray()` zurück. Weil die auszulesende Datenstruktur alles Mögliche enthalten kann, muss der Typ der Elemente also `Object` sein. Wir haben gerade festgestellt, dass der Elementtyp des Array-Objekts, das die Methode `toArray()` als Ergebnis liefert, mindestens so umfassend sein muss. Da es keinen allgemeineren (umfassenderen) Typ als `Object` gibt, ist auch der Typ des Arrays `Object[]`. Dies muss so sein, auch wenn die Elemente einer Datenstruktur im Einzelfall einen spezielleren Typ haben. Einer allgemeingültigen Implementierung von `toArray()` bleibt gar nichts anderes übrig, als das Array vom Typ `Object[]` und die Elemente vom Typ `Object` zu erzeugen.

Fassen wir zusammen: Wir haben vorher `Object[] os = new Point[3];` geschrieben und gesehen, dass `Object[]` ein Basistyp von `Point[]` ist. Das geht, weil `Object` auch ein Basistyp von `Point` ist. Die Vererbungsbeziehung von Typen überträgt sich auf die Vererbungsbeziehung der Arrays dieser Typen. Das nennt sich *kovariant*. Wenn allerdings – wie durch das parameterlose `toArray()` – nur ein `Object[]`-Array aufbaut ist, lässt es sich nicht auf den Typ `Point[]` bringen, auch wenn jedes Element ein `Point` ist.

### Die Lösung für das Problem

Bevor wir nun eine Schleife mit einer Typumwandlung für jedes einzelne Array-Element schreiben oder eine Typumwandlung bei jedem Zugriff auf die Elemente vornehmen, sollten wir einen Blick auf die zweite `toArray(T[])`-Methode werfen. Sie akzeptiert als Parameter ein vorgefertigtes Array für das Ergebnis. Mit dieser Methode lässt sich erreichen, dass das Ergebnis-Array von einem spezielleren Typ als `Object[]` ist.



#### Beispiel

Wir fordern von der `toArray()`-Methode ein Array vom Typ `Point`:

```

List<Point> list = new ArrayList<>();
list.add( new Point(13,43) );
list.add( new Point(9,4) );
Point[] points = (Point[]) list.toArray( new Point[0] );

```

Die Listenelemente bekommen wir in ein Array kopiert, und der Typ des Arrays ist `Point[]` – passend zu den aktuell vorhandenen Listenelementen. Der Parameter zeigt dabei den Wunschtyp an, der hier das `Point`-Array ist.



### Performance-Tipp

Am besten ist es, bei `toArray(T[])` ein Array anzugeben, das so groß ist wie das Ergebnis-Array, also so groß wie die Liste. Dann füllt nämlich `toArray(T[])` genau dieses Array und gibt es zurück, anstatt ein neues Array aufzubauen:

```
ArrayList<Point> list = new ArrayList<>();
list.add( new Point(13,43) );
list.add( new Point(9,4) );
Point[] points = list.toArray( new Point[list.size()] );
// alternative Point[] points = list.toArray( Point[]::new );
```

### Arrays mit Reflection anlegen \*

Intern verwendet die Methode `toArray(T[])` *Reflection*, um dynamisch ein Array vom gleichen Typ wie das übergebene Array zu erzeugen. Es kommt eine Methode aus `java.lang.reflect.Array` zum Einsatz:

- `static Object newInstance(Class<?> componentType, int length)`



### Beispiel

Lege ein neues Array an, das den gleichen Typ wie `array` hat. Das neue leere Feld soll Platz für `len` Elemente haben:

```
Object[] array = { new Point(-1, -1), new Point( 1, 1 ) };
int len = 100;
Object[] b = (Object[]) Array.newInstance(
    array.getClass().getComponentType(), len);
```

Der Aufruf `array.getClass()` liefert ein `Class`-Objekt für das Array `array`, etwa ein Objekt, das den Typ `Point[]` repräsentiert. Mit `array.getClass().getComponentType()` erhalten wir ein `Class`-Objekt für den Elementtyp des Arrays, also `Point`.

Mit `newInstance(...)` können wir das dynamisch machen, was `new TYP[len]` in Java-Quellcode macht, wobei bei `new` der Elementtyp zur Übersetzungszeit festgelegt werden muss. Da der Rückgabewert von `newInstance()` ein allgemeines `Object` ist, muss letztendlich noch die Konvertierung in ein passendes Array stattfinden.

Ist das übergebene Array so groß, dass es alle Elemente der Sammlung aufnehmen kann, kopiert `toArray(T[])` die Elemente aus der Collection in das Array. Im Übrigen entspricht `toArray(new Object[0])` dem Aufruf von `toArray()`.

### 17.1.9 Primitive Elemente in Datenstrukturen verwalten

Jede Datenstruktur der Collection-API akzeptiert, auch wenn sie generisch verwendet wird, nur Referenzen. Primitive Datentypen nehmen die Sammlungen nicht auf, was zur Konsequenz hat, dass Wrapper-Objekte nötig sind (über das Boxing fügt Java scheinbar primitive Elemente ein, doch in Wahrheit sind es Wrapper-Objekte). Auch wenn es also

```
List<Double> list = new ArrayList<>();
list.add( 1.1 );
list.add( 2.2 );
```

heißt, sind es zwei neue Double-Objekte, die aufgebaut werden und in die Liste wandern. Anders und klarer geschrieben, sehen wir hier, was wirklich passiert:

```
List<Double> list = new ArrayList<>();
list.add( Double.valueOf(1.1) );
list.add( new Double(2.2) );
```

Dem Aufruf von `Double.valueOf(...)` ist die `new`-Nutzung nicht abzulesen, doch die Methode ist implementiert als: `Double valueOf(double d){ return new Double(d); }`.

### Spezialbibliotheken

Für performante Anwendungen und große Mengen von primitiven Elementen ist es sinnvoll, eine Klasse für den speziellen Datentyp einzusetzen. Anstatt so etwas selbst zu programmieren, kann der Entwickler auf drei Implementierungen zurückgreifen:

- ▶ *fastutil (<http://fastutil.di.unimi.it/>)*: Erweiterung um Datenstrukturen für (sehr viele) primitive Elemente und hochperformante Ein-/Ausgabe-Klassen
- ▶ *GNU Trove: High performance collections for Java (<https://bitbucket.org/trove4j/trove>)*: Stabil, und die Entwicklung ist relativ aktiv.
- ▶ *Eclipse Collections (<https://www.eclipse.org/collections/>)*: Neben primitiven Sammlungen reiche API. Ehemals *GS Collections*.

## 17.2 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 Binärbäume realisiert, die eine Sortierung ermöglichen.

- ▶ `LinkedHashSet`: schnelle Mengenimplementierung unter Beibehaltung der Einfügereihenfolge
- ▶ `EnumSet`: eine spezielle Menge ausschließlich für Aufzählungen
- ▶ `CopyOnWriteArrayList`: schnelle Datenstruktur für viele lesende Operationen

### 17.2.1 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 17.7** src/main/java/com/tutego/insel/util/set/WhereHaveYouBeen.java

```
package com.tutego.insel.util.set;

import java.text.BreakIterator;
import java.util.*;

public class WhereHaveYouBeen {
    public static void main( String[] args ) {
        // Menge mit Städten aufbauen

        Set<String> allCities = new HashSet<>();
        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<>();

        // 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.substring( first, last );
        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: " + String.join( ", ", visitedCities ) );
    Set<String> unvisitedCities = new TreeSet<>( allCities );
    unvisitedCities.removeAll( visitedCities );
    System.out.println( "Unbesuchte Städte: " + String.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 visitedCities merkt sich die besuchten Städte. Auch dieses Set ist schnell, hat aber 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 Städten 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 Rückgabetypr 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 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.



### Hinweis

Auch reguläre Ausdrücke wären eine Option zum Zerlegen von Sätzen und kommen an anderen Stellen im Buch auch vor. Allerdings hat der BreakIterator den Vorteil, dass er jedes einzelne Unicode-Zeichen korrekt einordnen kann.

### 17.2.2 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 dort noch kein `equals`-gleiches Objekt vorliegt. Liefert `true` bei erfolgreichem Einfügen.
- `boolean addAll(Collection<? extends E> c)`  
Fügt alle Elemente von `c` in das Set ein. Ist `c` ein anderes Set, so steht `addAll(...)` für die Mengenvereinigung, im Kern ein `for (E e : c) add(e);`. Die Rückgabe ist `true`, wenn sich die Sammlung in irgendeiner Weise verändert hat.
- `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?
- `static <E> Set<E> copyOf(Collection<? extends E> coll)`  
Erzeugt eine unmodifizierbare Kopie.
- `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, andernfalls `false`, wenn es kein `equals`-gleiches Objekt in der Menge gab.
- `boolean removeAll(Collection<?> c)`  
Löscht alle Elemente der Collection aus dem Set und liefert `true` bei erfolgreichem Löschen.
- `boolean retainAll(Collection<?> c)`  
Die Menge behält (engl. *retain*) alle Elemente, die auch in `c` sind. Anders gesagt: Alle Elemente der eigenen Menge werden gelöscht, die nicht auch in `c` vorhanden sind. An der Datenstruktur `c` gibt es keine Änderung. Die eigene Menge ist dann die Schnittmenge mit `c`, aber `c` kann noch viel mehr Elemente enthalten.

- `int size()`  
Gibt die Anzahl der Elemente in der Menge zurück.
- `Object[] toArray()`  
Erzeugt zunächst ein neues Array, in dem alle Elemente der Menge Platz finden, und kopiert anschließend die Elemente in das Array.
- `<T> T[] toArray(T[] a)`  
Ist das übergebene Array groß genug für alle Elemente der Menge, dann werden alle Elemente der Menge in das Array kopiert und zurückgegeben. Ist das Array zu klein, wird ein neues Array vom Typ `T` angelegt, und alle Elemente werden von der Menge in das Array kopiert und zurückgegeben.
- `default <T> T[] toArray(IntFunction<T[]> generator)`  
Wie in `Collection` dokumentiert.

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.

### Ein Element erneut hinzunehmen

Ist ein Element in der Menge noch nicht vorhanden, fügt `add(...)` es ein und liefert als Rückgabe `true`. Ist es schon vorhanden, macht `add(...)` nichts und liefert `false` (das ist bei einer `Map` anders, denn dort überschreibt `put(...)` den Schlüssel). Ob ein hinzuzufügendes Element mit einem existierenden in der Menge übereinstimmt, bestimmt die `equals(...)`-Methode, also die Gleichheit und nicht die Identität:

**Listing 17.8** src/main/java/com/tutego/insel/util/set/HashSetDoubleAdd.java, main()

```
Set<Point> set = new HashSet<>();
Point p1 = new Point(), p2 = new Point();
System.out.println( set.add(p1) );      // true
System.out.println( set.add(p1) );      // false
System.out.println( set.add(p2) );      // false
System.out.println( set.contains(p1) ); // true
System.out.println( set.contains(p2) ); // true
```

### 17.2.3 HashSet

Ein `java.util.HashSet` verwaltet die Elemente in einer schnellen hashbasierten Datenstruktur. Dadurch sind die Elemente schnell eingesortiert und schnell zu finden. Falls eine Sortierung der Elemente vom `HashSet` nötig ist, müssen die Elemente zum Beispiel in eine `List` oder `TreeSet` umkopiert und dann sortiert werden.

```
class java.util.HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

- `HashSet()`  
Erzeugt ein leeres `HashSet`-Objekt.
- `HashSet(Collection<? extends E> c)`  
Erzeugt aus der Sammlung `c` ein neues unsortiertes `HashSet`.
- `HashSet(int initialCapacity)`
- `HashSet(int initialCapacity, float loadFactor)`  
Die beiden Konstruktoren sind zur Optimierung gedacht –  
`HashSet` basiert intern auf der `HashMap`.

### 17.2.4 TreeSet – die sortierte Menge

Die Klasse `java.util.TreeSet` implementiert ebenfalls wie `HashSet` die Set-Schnittstelle, verfolgt aber eine andere Implementierungsstrategie. Ein `TreeSet` verwaltet die Elemente immer sortiert (intern werden die Elemente in einem balancierten Binärbaum gehalten). Speichert `TreeSet` ein neues Element, so fügt `TreeSet` das Element automatisch sortiert in die Datenstruktur ein. Das kostet zwar etwas mehr Zeit als ein `HashSet`, doch ist diese Sortierung dauerhaft. Daher ist es auch nicht zeitaufwändig, alle Elemente geordnet auszugeben. Die Suche nach einem einzigen Element ist aber etwas langsamer als im `HashSet`. Der Begriff »langsamer« muss jedoch relativiert werden: Die Suche ist logarithmisch und daher nicht wirklich »langsam«. Beim Einfügen und Löschen muss bei bestimmten Konstellationen eine Reorganisation des Baums in Kauf genommen werden, was die Einfüge-/Löschezeit verschlechtert. Doch auch beim Re-Hashing gibt es diese Kosten, die sich dort jedoch durch die passende Startgröße vermeiden lassen.

```
class java.util.TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable
```

- `TreeSet()`  
Erzeugt ein neues, leeres `TreeSet`.

- `TreeSet(Collection<? extends E> c)`  
Erzeugt ein neues TreeSet aus der gegebenen Collection.
- `TreeSet(Comparator<? super E> c)`  
Erzeugt ein leeres TreeSet mit einem gegebenen Comparator, der für die Sortierung der internen Datenstruktur die Vergleiche übernimmt.
- `TreeSet(SortedSet<E> s)`  
Erzeugt ein neues TreeSet, und übernimmt alle Elemente von s und auch die Sortierung von s. (Einen Konstruktor mit NavigableSet gibt es nicht.)

[zB]

### Beispiel

Teste, ob eine Liste von Datumswerten – die nur einmal in der Liste vorkommen dürfen – aufsteigend sortiert ist:

```
List<Instant> dates = Arrays.asList( Instant.ofEpochMilli( 2L ),
                                         Instant.ofEpochMilli( 3L ) );
boolean isSorted = new ArrayList<>( new TreeSet<>( dates ) ).equals( dates );
```

Nimmt der Konstruktor von TreeSet eine andere Sammlung entgegen, so entsteht eine sortierte Sammlung aller Elemente. Diese Sammlung kann wiederum in einen anderen Konstruktor gegeben werden, der Collection-Objekte annimmt, wie zum Beispiel eine ArrayList. Unser Beispiel vergleicht zwei List-Exemplare mit `equals(...)`, wobei Listen eine Ordnung haben. Stimmt die Ordnung nach dem Sortieren mit der vor der Sortierung überein, war die Liste schon sortiert.

Natürlich gibt es bessere Implementierungen, zum Beispiel mit einem Iterator die Sammlung abzulaufen und zu schauen, ob das nächste Element immer größer oder gleich ist.

### Bedeutung der Sortierung

Durch die interne sortierte Speicherung gibt es zwei ganz wichtige Bedingungen:

- Die Elemente müssen sich vergleichen lassen. Kommen zum Beispiel Player-Objekte in das TreeSet, aber implementiert Player nicht die Schnittstelle Comparable, löst TreeSet eine Ausnahme aus, da TreeSet nicht weiß, in welcher Reihenfolge die Spieler stehen.
- Die Elemente müssen vom gleichen Typ sein. Wie sollte sich ein Kirchen-Objekt mit einem Staubsauger-Objekt vergleichen lassen?

[zB]

### Beispiel

Sortiere Strings in eine Menge ein, wobei die Groß-/Kleinschreibung und vorangestellter bzw. nachfolgender Weißraum keine Rolle spielen. Anders gesagt: Wörter sollen auch dann als gleich angesehen werden, wenn sie sich in der Groß-/Kleinschreibweise unterscheiden oder etwa Weißraum am Anfang und Ende besitzen:

```
Comparator<String> comparator = (s1, s2) ->
    String.CASE_INSENSITIVE_ORDER.compare( s1.trim(), s2.trim() );
Set<String> set = new TreeSet<>( comparator );
Collections.addAll( set, "xxx", "XXX", "tang", "xXx", "QUEEF" );
System.out.println( set ); // [ QUEEF , tang, xxx ]
```

### Die Methode equals(...) und die Vergleichsmethoden

Die Methode `equals(...)` spielt für Datenstrukturen eine große Rolle. Beim `TreeSet` ist das anders, denn es nutzt zur Einordnung einen externen `Comparator` bzw. die `compareTo(...)`-Eigenschaft, wenn die Elemente `Comparable` sind. Gibt die Vergleichsmethode 0 zurück, so sind die Elemente gleich, und gleiche Elemente sind in der Menge nicht erlaubt. `equals(...)` wird dabei nicht gefragt, wenngleich natürlich die `equals`-Implementierung `true` liefern sollte, wenn `compare(...)/compareTo(...)` 0 liefert.

Nehmen wir als Beispiel den `Comparator` aus dem vorangegangenen Beispiel für `String`-Objekte, der unabhängig von der Groß-/Kleinschreibung und Weißraum vergleicht. Dann sind laut `equals(...)` die `Strings` "xxx" und "XXX" sicherlich nicht gleich, der `Comparator` würde aber Gleichheit anzeigen. Dies führt dazu, dass tatsächlich nur eines der beiden Objekte in das `TreeSet` kommt und eine Anfrage nach einem `Comparator`-gleichen Objekt daher das Element liefert:

```
Comparator<String> comparator = (s1, s2) ->
    String.CASE_INSENSITIVE_ORDER.compare( s1.trim(), s2.trim() );
Set<String> set = new TreeSet<>( comparator );
```

### 17.2.5 Die Schnittstellen `NavigableSet` und `SortedSet`

`TreeSet` implementiert die Schnittstelle `NavigableSet` und bietet darüber Methoden, die insbesondere zu einem gegebenen Element das nächsthöhere/-kleinere liefern. Somit sind auf Mengen nicht nur die üblichen Abfragen über Mengenzugehörigkeit denkbar, sondern auch Abfragen wie »Gib mir das Element, das größer oder gleich einem gegebenen Element ist«.

Folgendes Beispiel reiht in ein `TreeSet` drei `LocalDate`-Objekte ein – die Klasse `LocalDate` implementiert `Comparable<LocalDate>`, denn die Objekte haben eine natürliche Ordnung. Die Methoden `lower(...)`, `ceiling(...)`, `floor(...)` und `higher(...)` wählen aus der Menge das angefragte Objekt aus:

**Listing 17.9** src/main/java/com/tutego/insel/util/set/SortedSetDemo.java

```
NavigableSet<LocalDate> set = new TreeSet<>();
set.add( LocalDate.of( 2018, Month.MARCH, 10 ) );
set.add( LocalDate.of( 2018, Month.MARCH, 12 ) );
```

```

set.add( LocalDate.of( 2018, Month.MARCH, 14 ) );

LocalDate cal1 = set.lower( LocalDate.of( 2018, Month.MARCH, 12 ) );
System.out.printf( "%tF%n", cal1 ); // 2018-03-10

LocalDate cal2 = set.ceiling( LocalDate.of( 2018, Month.MARCH, 12 ) );
System.out.printf( "%tF%n", cal2 ); // 2018-03-12

LocalDate cal3 = set.floor( LocalDate.of( 2018, Month.MARCH, 12 ) );
System.out.printf( "%tF%n", cal3 ); // 2018-03-12

LocalDate cal4 = set.higher( LocalDate.of( 2018, Month.MARCH, 12 ) );
System.out.printf( "%tF%n", cal4 ); // 2018-03-14

```

Eine Methode wie tailSet(...) ist insbesondere bei Datumsobjekten sehr praktisch, da sie alle Zeitpunkte liefern kann, die nach einem Startdatum liegen.

TreeSet implementiert die Schnittstelle NavigableSet, die ihrerseits SortedSet erweitert – ein historisches Erbe. Insgesamt bietet NavigableSet 15 Operationen, wobei sie aus SortedSet die Methoden headSet(...), tailSet(...) und subSet(...) um die überladene Version der Methoden ergänzt, die die Grenzen exklusiv oder inklusiv erlauben.

```

interface java.util.NavigableSet<E>
extends SortedSet<E>

```

- NavigableSet<E> headSet(E toElement)
- NavigableSet<E> tailSet(E fromElement)
 

Liefert eine Teilmenge von Elementen, die echt kleiner/größer als toElement/fromElement sind.
- NavigableSet<E> headSet(E toElement, boolean inclusive)
- NavigableSet<E> tailSet(E fromElement, boolean inclusive)
 

Bestimmt gegenüber den oberen Methoden zusätzlich, ob das Ausgangselement zur Ergebnismenge gehören darf.
- NavigableSet<E> subSet(E fromElement, E toElement)
 

Liefert eine Teilmenge im gewünschten Bereich.
- E pollFirst()
- E pollLast()
 

Holt und entfernt das erste/letzte Element. Die Rückgabe ist null, wenn das Set leer ist.
- E higher(E e)

- `E lower(E e)`  
Liefert das folgende/vorangehende Element im Set, das echt größer/kleiner als E ist, oder null, falls ein solches Element nicht existiert.
- `E ceiling(E e)`
- `E floor(E e)`  
Liefert das folgende/vorangehende Element im Set, das größer/kleiner oder gleich E ist, oder null, falls ein solches Element nicht existiert.
- `Iterator<E> descendingIterator()`  
Liefert die Elemente in umgekehrter Reihenfolge.

Aus der Schnittstelle SortedSet erbt NavigableSet im Grunde nur drei Operationen, denn `subSet(...)`, `headSet(...)` und `tailSet(...)` werden mit kovariantem Rückgabetyp in NavigableSet re-definiert.

```
interface java.util.SortedSet<E>
extends Set<E>
```

- `E first()`  
Liefert das kleinste Element in der Liste.
- `E last()`  
Liefert das größte Element.
- `Comparator<? super E> comparator()`  
Liefert den mit der Menge verbundenen Comparator. Die Rückgabe kann null sein, wenn sich die Objekte mit Comparable selbst vergleichen können.
- `SortedSet<E> subSet(E fromElement, E toElement)`
- `SortedSet<E> headSet(E toElement)`
- `SortedSet<E> tailSet(E fromElement)`

Anders als HashSet liefert der Iterator beim TreeSet die Elemente aufsteigend sortiert. Davon profitieren auch die beiden `toArray(...)`-Methoden – implementiert in AbstractCollection –, da sie den Iterator nutzen, um ein sortiertes Array zurückzugeben.

### Beispiel

Eine Variable `contacts` ist vom Typ `Map<Long, String>` und assoziiert IDs vom Typ `long` mit Strings. Ein neuer Kontakt soll eine ID bekommen, die um 1 höher ist als die höchste ID des Assoziativspeichers:

```
contact.setId( new TreeSet<Long>( contacts.keySet() ).last() + 1L );
```



### 17.2.6 LinkedHashSet

Ein `LinkedHashSet` vereint die Reihenfolgentreue einer Liste und die hohe Performance für Mengenoperationen vom `HashSet`. Dabei bietet die Klasse keine Listen-Methoden wie `first()` oder `get(int index)`, sondern ist eine Implementierung ausschließlich der Set-Schnittstelle, in der der Iterator die Elemente in der Einfügereihenfolge liefert:

**Listing 17.10** src/main/java/com/tutego/insel/util/set/LinkedHashSetDemo.java, main()

```
Set<Integer> set = new LinkedHashSet<>(
    Arrays.asList( 9, 8, 7, 6, 9, 8 )
);

for ( Integer i : set )
    System.out.print( i + " " );      // 9 8 7 6

System.out.printf( "%n%s", set ); // [9, 8, 7, 6]
```

Da ein Set jedes Element nur einmal beinhalten kann, bekommen wir als Ergebnis jedes Element nur einmal, aber gleichzeitig geht die Reihenfolge des Einfügens nicht verloren. Der Iterator liefert die Elemente genau in der Einfügereihenfolge.



#### Beispiel

Dass ein `LinkedHashSet` eine Menge ist, die Elemente nur einmal enthält, sich aber beim Einfügen wie eine Liste verhält, ist nützlich, um doppelte Elemente aus einer Liste zu löschen:

```
public static <T> List<T> removeDuplicate( List<T> list ) {
    return new ArrayList<>( new LinkedHashSet<>( list ) );
}
```

Das Ergebnis ist eine neue Liste, und `list` selbst wird nicht modifiziert. Es ergibt zum Beispiel `removeDuplicate( Arrays.asList( 1,2,1,3,1,2,4 ) )` die Liste `[1, 2, 3, 4]`.

### LinkedHashSet und Iterator

Mit einem Iterator lässt sich jedes Element von `LinkedHashSet` nach der Reihenfolge des Einfügens auflisten. Der Iterator von `LinkedHashSet` unterstützt auch die `remove()`-Methode. Sie kann eingesetzt werden, um die ältesten Einträge zu löschen und nur noch die neuesten zwei Elemente beizubehalten:

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();
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]

```

## 17.3 Java Stream API

### 17.3.1 Deklaratives Programmieren

Die API ist im funktionalen Stil, und Programme lesen sich damit sehr kompakt (die einzelnen Methoden werden in diesem Kapitel detailliert vorgestellt):

```

Object[] words = { " ", '3', null, "2", 1, "" };
Arrays.stream( words )                                // Erzeugt neuen Stream
    .filter( Objects::nonNull )                      // Belasse Nicht-null-Referenzen im Stream
    .map( Objects::toString )                        // Konvertiere Objekte in Strings
    .map( String::trim )                            // Schneide Weißraum ab
    .filter( s -> ! s.isEmpty() )                  // Belasse nicht-leere Elemente im Stream
    .map( Integer::parseInt )                       // Konvertiere Strings in Ganzzahlen
    .sorted()                                       // Sortiere die Ganzzahlen
    .forEach( System.out::println ); // 1 2 3

```

Während die Klassen aus der Collection-API optimale Speicherformen für Daten realisieren, ist es Aufgabe der Stream-API, die Daten komfortabel zu erfragen. Gut ist hier zu erkennen, dass die Stream-API das *Was* betont, nicht das *Wie*. Das heißt, Durchläufe und Iterationen kommen im Code nicht vor, sondern die Fluent-API beschreibt deklarativ, wie das Ergebnis aussehen soll. Die Bibliothek realisiert schlussendlich das *Wie*. So kann eine Implementierung zum Beispiel entscheiden, ob die Abarbeitung sequenziell oder parallel erfolgt, ob die Reihenfolge eine Rolle spielen muss oder alle Daten zwecks Sortierung zwischengespeichert werden müssen usw.

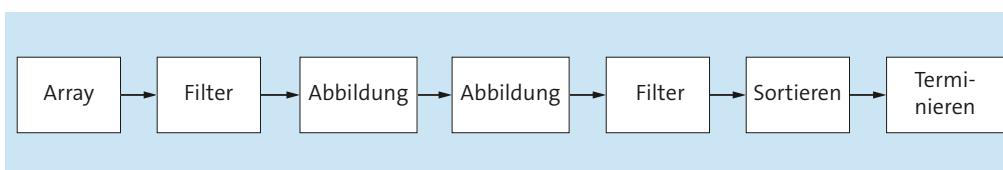


Abbildung 17.2 Pipeline-Prinzip bei Streams vom vorangehenden Beispiel

### 17.3.2 Interne versus externe Iteration

Als Erstes fällt bei der Stream-API auf, dass die klassische Schleife fehlt. Normalerweise gibt es Schleifen, die durch Daten laufen und dann Abfragen auf den Elementen vornehmen.

Traditionelle Schleifen sind immer sequenziell und laufen von Element zu Element, und zwar vom Anfang bis zum Ende. Das Gleiche gilt auch für einen Iterator. Die Stream-API verfolgt einen anderen Ansatz, hier wird die so genannte *externe Iteration* (durch Schleifen vom Entwickler gesteuert) durch eine *interne Iteration* (die Stream-API holt sich Daten) abgelöst. Wenn etwa `forEach(...)` nach Daten fragt, wird die Datenquelle abgezapft und ausgesaugt, aber erst dann. Der Vorteil ist, dass wir zwar bestimmen, welche Datenstruktur abgelaufen werden soll, aber wie das intern geschieht, kann die Implementierung selbst bestimmen und dahingehend optimieren. Wenn wir selbst die Schleife schreiben, läuft die Verarbeitung immer Element für Element, während die interne Iteration auch von sich aus parallelisieren und Teilprobleme von mehreren Ausführungseinheiten berechnen lassen kann.



### Hinweis

An verschiedenen Sammlungen hängt eine `forEach(...)`-Methode, die über alle Elemente läuft und eine Methode auf einem übergebenen Konsumenten aufruft. Das heißt jetzt nicht, dass die klassische `for`-Schleife – etwa über das erweiterte `for` – damit überflüssig wird. Neben der einfachen Schreibweise und dem einfachen Debuggen hat die übliche Schleife immer noch einige Vorteile. `forEach(...)` bekommt den auszuführenden Code in der Regel über einen Lambda-Ausdruck, und der hat Einschränkungen. So darf er etwa keine lokalen Variablen beschreiben (alle vom Lambda-Ausdruck adressierten lokalen Variablen sind effektiv `final`), und Lambda-Ausdrücke dürfen keine geprüften Ausnahmen auslösen – im Inneren einer Schleife ist das alles kein Thema. Im Übrigen gibt es für Schleifenabbrüche das `break`, das in Lambda-Ausdrücken nicht existiert (ein `return` im Lambda entspricht `continue`).

#### 17.3.3 Was ist ein Stream?

Ein Strom ist eine Sequenz von Daten – aber keine Datenquelle an sich –, die Daten wie eine Datenstruktur speichert. Die Daten vom Strom werden in einer Kette von nachgeschalteten Verarbeitungsschritten

- ▶ gefiltert (engl. *filter*),
- ▶ transformiert/abgebildet (engl. *map*) und
- ▶ komprimiert/reduziert (engl. *reduce*).

Die Verarbeitung entlang einer Kette nennt sich *Pipeline* und besteht aus folgenden drei Komponenten:

- ▶ Am Anfang steht eine Datenquelle, wie etwa ein Array, eine Datenstruktur oder ein Generator.
- ▶ Es folgen diverse Verarbeitungsschritte wie Filterungen (Elemente verschwinden aus dem Strom) oder Abbildungen (ein Datentyp kann auch in einen Datentyp konvertiert wer-

den); diese Veränderungen auf dem Weg nennen sich *intermediäre Operationen* (engl. *intermediate operations*). Ergebnis einer intermediären Operation ist wieder ein Stream.

- ▶ Am Schluss wird das Ergebnis eingesammelt, das Ergebnis ist kein Stream mehr. Eine Reduktion wäre zum Beispiel eine Maximumsbildung oder die Konkatenation von Strings.

Die eigentliche Datenstruktur wird nicht verändert, vielmehr steht am Ende der intermediären Operationen eine terminale Operation, die das Ergebnis erfragt. So eine terminale Operation ist etwa `forEach(...)`; sie steht am Ende der Kette, und der Strom bricht ab.

Viele terminale Operationen reduzieren die durchlaufenden Daten auf einen Wert, anders als etwa `forEach(...)`. Dazu gehören etwa Methoden zum einfachen Zählen der Elemente oder zum Summieren; das nennen wir *reduzierende Operationen*. In der API gibt es für Standardreduktionen – wie Bildung der Summe, des Maximums, des Durchschnitts – vorgefertigte Methoden, doch sind allgemeine Reduktionen über eigene Funktionen möglich, etwa statt der Summe das Produkt.

### **Lazy Love**

Alle intermediären Operationen sind »faul« (engl. *lazy*), weil sie die Berechnungen so lange hinausschieben, bis sie benötigt werden. Am ersten Beispiel ist das gut abzulesen: Wenn die Elemente aus dem Array genommen werden, werden sie der Reihe nach weitergereicht zum nächsten Verarbeitungsschritt. Entfernt der Filter Elemente aus dem Strom, sind sie weg und müssen in einem späteren Schritt nicht mehr berücksichtigt werden. Es ist also nicht so, dass die Daten mehrfach existieren, zum Beispiel in einer Datenstruktur mit allen Elementen ohne `null`, dann alle Objekte, die in Strings konvertiert werden, dann alle getrimmten Strings usw.

Im Gegensatz zu den fortführenden Operationen stehen terminale Operationen, bei denen das Ergebnis vorliegen muss: Sie sind »begierig« (engl. *eager*). Im Prinzip wird alles so lange aufgeschoben, bis ein Wert gebraucht wird, das heißt, bis eine terminale Operation auf das Ergebnis wirklich zugreifen möchte.

### **Zustand ja oder nein**

Intermediäre Operationen können einen Zustand haben oder nicht. Eine Filteroperation zum Beispiel hat keinen Zustand, weil sie zur Erfüllung ihrer Aufgabe nur das aktuelle Element betrachten muss, nicht aber die vorangehenden. Eine Sortierungsoperation hat dagegen einen Status, sie »will«, dass alle anderen Elemente gespeichert werden, denn das aktuelle Element reicht für die Sortierung nicht aus, sondern auch alle vorangehenden werden benötigt.

## 17.4 Stream erzeugen

Das Paket `java.util.stream` deklariert diverse Typen rund um Streams. Im Zentrum steht für Objektströme eine generisch deklarierte Schnittstelle `Stream`. Ein konkretes Exemplar wird immer von einer Datenquelle erzeugt, u. a. stehen folgende Stream-Erzeuger zur Verfügung:

Typ	Rückgabe	Methode
<code>Collection&lt;E&gt;</code>	<code>Stream&lt;E&gt;</code>	<code>stream()</code>
<code>Arrays</code>	<code>Stream&lt;T&gt;</code>	<code>stream(T[] array) (statisch)</code>
	<code>Stream&lt;T&gt;</code>	<code>stream(T[] array, int start, int end) (statisch)</code>
<code>Stream</code>	<code>Stream&lt;T&gt;</code>	<code>empty() (statisch)</code>
	<code>Stream&lt;T&gt;</code>	<code>of(T... values) (statisch)</code>
	<code>Stream&lt;T&gt;</code>	<code>of(T value) (statisch)</code>
	<code>Stream&lt;T&gt;</code>	<code>generate(Supplier&lt;T&gt; s)(statisch)</code>
	<code>Stream&lt;T&gt;</code>	<code>iterate(T seed, UnaryOperator&lt;T&gt; f)(statisch)</code>
	<code>Stream&lt;T&gt;</code>	<code>iterate(T seed, Predicate&lt;? super T&gt; hasNext, UnaryOperator&lt;T&gt; next) (statisch)</code>
	<code>Stream&lt;T&gt;</code>	<code>ofNullable(T t) (statisch)</code>
<code>Optional&lt;T&gt;</code>	<code>Stream&lt;T&gt;</code>	<code>stream()</code>
<code>Scanner</code>	<code>Stream&lt;String&gt;</code>	<code>tokens()</code>
<code>String</code>	<code>Stream&lt;String&gt;</code>	<code>lines() (ab Java 11)</code>
<code>Files</code>	<code>Stream&lt;String&gt;</code>	<code>lines(Path path)</code>
	<code>Stream&lt;String&gt;</code>	<code>lines(Path path, Charset cs)</code>
	<code>Stream&lt;Path&gt;</code>	<code>list(Path dir)</code>
	<code>Stream&lt;Path&gt;</code>	<code>walk(Path start, FileVisitOption... options)</code>
	<code>Stream&lt;Path&gt;</code>	<code>walk(Path start, int maxDepth, FileVisitOption... options)</code>

Tabelle 17.1 Methoden, die Stream-Exemplare liefern

Typ	Rückgabe	Methode
Files (Forts.)	Stream<Path>	find(Path start, int maxDepth, BiPredicate<Path, BasicFileAttributes> matcher, FileVisitOption... options)
BufferedReader	Stream<String>	lines()
Pattern	Stream<String>	splitAsStream(CharSequence input)
ZipFile	Stream<? extends ZipEntry>	stream()
	Stream<JarEntry>	stream()
	Stream<JarEntry>	versionedStream() (ab Java 10)

Tabelle 17.1 Methoden, die Stream-Exemplare liefern (Forts.)

Die Methode `Stream.empty()` liefert wie erwartet einen Strom ohne Elemente.

### Stream.ofXXX(...)

`Stream` hat einige Fabrikmethoden für neue Ströme (etwa `of(...)`), alles andere sind Objektmethoden anderer Klassen, die Ströme liefern. Die `of(...)`-Methoden sind als statische Schnittstellenmethoden von `Stream` implementiert, und `of(T... values)` ist nur eine Fassade für `Arrays.stream(values)`. Die Methode `static <T> Stream<T> ofNullable(T t)` liefert, wenn `t == null` ist, einen leeren Stream, und wenn `t != null` ist, einen Stream mit genau dem Element `t`. Die Methode ist nützlich für Teilströme, die integriert werden.

#### Beispiel

Produziere einen Stream aus gegebenen Ganzzahlen, entferne die Vorzeichen, sortiere das Ergebnis und gib es aus:

```
Stream.of( -4, 1, -2, 3 )
    .map( Math::abs )
    .sorted()
    .forEach( System.out::println ); // 1 2 3 4
```



### Stream.generate(...)

`generate(...)` produziert Elemente aus einem `Supplier`; der Stream ist unendlich.



## Beispiele

Erzeuge ein Array von zehn Zufallszahlen nach Normalverteilung:

```
Random random = new Random();
double[] randoms = Stream.generate( random::nextGaussian )
    .limit( 10 ).mapToDouble( e -> e ).toArray();
System.out.println( Arrays.toString( randoms ) );
```

Erzeuge einen Strom von Fibonacci-Zahlen<sup>4</sup>:

```
class FibSupplier implements Supplier<BigInteger> {
    private final Queue<BigInteger> fibs =
        new LinkedList<>( Arrays.asList( BigInteger.ZERO, BigInteger.ONE ) );
    @Override public BigInteger get() {
        fibs.offer( fibs.remove().add( fibs.peek() ) );
        return fibs.peek();
    }
};
Stream.generate( new FibSupplier() )
    .limit( 1000 ).forEach( System.out::println );
```

Die Implementierung nutzt keinen Lambda-Ausdruck, sondern eine almodische Klassenimplementierung, da wir uns für die Fibonacci-Folgen die letzten beiden Elemente merken müssen. Sie speichert eine `LinkedList`, die als `Queue` genutzt wird. Bei der Anfrage an ein neues Element nehmen wir das erste Element heraus, sodass das zweite nachrutscht, und addieren es mit dem Kopfelement, was die Fibonacci-Zahl ergibt. Die hängen wir im zweiten Schritt hinten an und geben sie zurück. Parallel Zugriffe sind nicht gestattet.

## Stream.iterate(...)

Die zwei statischen `iterate(...)`-Methoden generieren einen Stream aus einem Startwert und einer Funktion, die das nächste Element produziert. Bei `iterate(T seed, UnaryOperator<T> f)` ist der Strom unendlich, bei der zweiten Methode `iterate(..., Predicate<? super T> hasNext, ...)` beendet ein erfülltes Prädikat den Strom und erinnert an eine klassische `for`-Schleife. Der Abbruch über ein Prädikat ist sehr flexibel, denn bei der ersten `iterate(...)`-Methode ist der Strom unendlich, und so folgt oftmals ein `limit(...)` oder `takeWhile(...)` zum Limitieren der Elemente.

---

<sup>4</sup> <https://de.wikipedia.org/wiki/Fibonacci-Folge>



### Beispiel 1

Produziere Permutationen eines Strings.

```
UnaryOperator<String> shuffleOp = s -> {
    char[] chars = s.toCharArray();
    for ( int index = chars.length - 1; index > 0; index-- ) {
        int rndIndex = ThreadLocalRandom.current().nextInt( index + 1 );
        if ( index == rndIndex ) continue;
        char c = chars[ rndIndex ];
        chars[ rndIndex ] = chars[ index ];
        chars[ index ] = c;
    }
    return new String( chars );
};
String text = "Sie müssen nur den Nippel durch die Lasche ziehn";
Stream.iterate( text, shuffleOp ).limit( 10 ).forEach( System.out::println );
```

Die Ganzzahl-Zufallszahlen stammen diesmal nicht von einem Random-Objekt, sondern von ThreadLocalRandom. Diese spezielle Klasse ist mit dem aktuellen Thread verbunden und kann ebenfalls einzelne Zufallszahlen oder einen Strom liefern. Bei Nebenläufigkeit bietet diese Variante eine bessere Performance.



### Beispiel 2

Erzeuge einen endlosen Stream aus BigInteger-Objekten, der bei 10.000.000 beginnt und in Einerschritten weitergeht, bis mit hoher Wahrscheinlichkeit eine Primzahl erscheint und der Strom damit endet.

```
Predicate<BigInteger> isNotPrime = i -> ! i.isProbablePrime( 10 );
UnaryOperator<BigInteger> incBigInt = i -> i.add( BigInteger.ONE );
Stream.iterate( BigInteger.valueOf( 10_000_000 ), isNotPrime, incBigInt )
    .forEach( System.out::println );
```

## 17.4.1 Parallele oder sequenzielle Streams

Ein Stream kann parallel oder sequenziell sein, das heißt, es ist möglich, dass Threads gewisse Operationen nebenläufig durchführen, wie zum Beispiel die Suche nach einem Element. Stream liefert über isParallel() die Rückgabe true, wenn ein Stream Operationen nebenläufig durchführt.

Alle Collection-Datenstrukturen können mit der Methode parallelStream() einen potenziell nebenläufigen Stream liefern.

Typ	Rückgabe	Methode
Collection	Stream<E>	parallelStream()
Collection	Stream<E>	stream()

Tabelle 17.2 Parallele und nichtparallele Streams von jeder Collection erfragen

Jeder parallele Stream lässt sich mithilfe der Stream-Methode sequential() in einen sequenziellen Stream konvertieren. Parallele Streams nutzen intern das Fork-&-Join-Framework, doch sollte nicht automatisch jeder Stream parallel sein, da das nicht zwingend zu einem Performance-Vorteil führt; wenn zum Beispiel keine Parallelisierung möglich ist, bringt es wenig Threads einzusetzen.

## 17.5 Terminale Stream-Operationen

Wir haben gesehen, dass sich Operationen in intermediär und terminal unterscheiden lassen. Die Schnittstelle Stream bietet insgesamt 18 terminale Operationen, und die Rückgaben der Methoden sind etwa void oder ein Array. Bei den intermediären Operationen sind die Rückgaben allesamt neue Stream-Exemplare. Wir schauen uns nacheinander die terminalen Operationen an.

### 17.5.1 Anzahl Elemente

Die vielleicht einfachste terminale Operation ist count(): Sie liefert ein long mit der Anzahl der Elemente im Strom.

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- long count()



### Beispiele

Wie viele Elemente hat ein Array von Referenzen, von null einmal abgesehen?

```
Object[] array = { null, 1, null, 2, 3 };
long size = Stream.of( array )
    .filter( Objects::nonNull ).count();
System.out.println( size ); // 3
```

### 17.5.2 Und jetzt alle – forEachXXX(...)

Die Stream-API bietet zwei forEachXXX(...)-Methoden zum Ablaufen der Ergebnisse:

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- void forEach(Consumer<? super T> action)
- void forEachOrdered(Consumer<? super T> action)

Übergeben wird immer ein Codeblock vom Typ Consumer. Die funktionale Schnittstelle Consumer hat einen Parameter, und forEachXXX(...) übergibt beim Ablaufen Element für Element an den Consumer.

Das Ablaufen mit forEach(...) ist ein Implementierungsdetail. Die Reihenfolge ist nicht zwingend deterministisch, was insbesondere bei parallelen Streams auffällt. Anders verhält sich forEachOrdered(...), das die Reihenfolge der Stream-Quelle respektiert, selbst wenn zwischen durch der Stream parallel verarbeitet wird. Es ist aber gut möglich, dass das Stream-Framework keine interne Parallelisierung nutzt, wenn forEachOrdered(...) die Elemente abfragt.

#### Beispiel

[zB]

Im ersten Fall ist die Ausgabe der Werte ungeordnet (z. B. rfAni), im zweiten Fall geordnet (Afrin).

```
"Afrin".chars().parallel().forEach( c -> System.out.print( (char)c ) );
System.out.println();
"Afrin".chars().parallel().forEachOrdered( c -> System.out.print( (char)c ) );
```

### 17.5.3 Einzelne Elemente aus dem Strom holen

Produziert der Strom mehrere Daten in Reihenfolge, so liefert findFirst() das erste Element im Strom, wohingegen findAny() irgendein Element vom Strom liefern kann. Letztere Methode ist insbesondere bei parallelen Operationen interessant, wobei es völlig offen ist, welches Element das ist. Beide Methoden haben den Vorteil, dass sie abkürzende Operationen sind, das heißt, sie ersparen einem einiges an Arbeit.

Da ein Strom von Elementen leer sein kann, ist die Rückgabe der Methoden immer Optional:

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- Optional<T> findFirst()
- Optional<T> findAny()



### Beispiel

```
Consumer<Character> print = System.out::println;
List<Character> chars = List.of( '1', 'a', '2', 'b', '3', 'c' );
chars.parallelStream().findFirst().ifPresent( print ); // 1
chars.parallelStream().findAny().ifPresent( print ); // b
```

Es liefert `findFirst()` immer das erste Element »1«, aber `findAny()` könnte jedes liefern, in der Ausgabe etwa »b«.

Hängen andere zustandsbehaftete Operationen dazwischen, ist eine Optimierung mit `findAny()` mitunter hinfällig. So wird `stream.sorted().findAny()` nicht die Sortierung umgehen können.

#### 17.5.4 Existenztests mit Prädikaten

Ob Elemente eines Stroms eine Bedingung erfüllen, zeigen drei Methoden:

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`

Die Bedingung wird immer als `Predicate` formuliert.



### Beispiel

Teste in einem Stream mit zwei Strings, ob entweder alle, irgendein oder kein Element leer ist:

```
System.out.println( Stream.of("", "") .allMatch( String::isEmpty ) ); // true
System.out.println( Stream.of("", "a") .anyMatch( String::isEmpty ) ); // true
System.out.println( Stream.of("", "a") .noneMatch( String::isEmpty ) ); // false
```

#### 17.5.5 Strom reduzieren auf kleinstes/größtes Element

Ein Strom kann aus beliebig vielen Elementen bestehen, die durch Reduktionsfunktionen auf einen Wert reduziert werden können. Ein bekanntes Beispiel ist die `Math.max(a, b)`-Methode, die zwei Werte auf das Maximum abbildet. Diese Maximum-/Minimum-Methoden gibt es auch im Stream:

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- `Optional<T> min(Comparator<? super T> comparator)`
- `Optional<T> max(Comparator<? super T> comparator)`

Interessant ist, dass immer ein Comparator übergeben werden muss und die Methoden nie auf die natürliche Ordnung zählen. Hilfreich ist hier eine Comparator-Utility-Methode, die genau so einen natürlichen Comparator liefert. Die Methoden liefern ein `Optional.empty()`, wenn der Stream leer ist.

### Beispiel

Was ist die größte Zahl im Stream?

```
System.out.println( Stream.of( 9, 3, 4, 11 ).max( Comparator.naturalOrder() ).get() );
```

[zB]

Die Bestimmung des Minimums/Maximums sind nur zwei Beispiele einer Reduktion. Allgemein kann jedes Paar von Werten auf einen Wert reduziert werden. Das sieht im Prinzip für die Minimum-Funktion wie folgt aus:

Stream (4, 2, 3, 1)	Funktionsanwendung	Resultat (Minimum)
4	-	4
2	<code>min(4, 2)</code>	2
3	<code>min(2, 3)</code>	2
1	<code>min(2, 1)</code>	1

Tabelle 17.3 Minimum-Berechnung beim Stream durch Reduktion

Im ersten Fall wird keine Funktion angewendet, und es wird nicht reduziert; das einzelne Element bleibt vorhanden.

### 17.5.6 Strom mit eigenen Funktionen reduzieren

Java greift für Reduktionen auf die funktionalen Schnittstellen `BiFunction<T,U,R>` bzw. `BinaryOperator<T>` zurück (`BinaryOperator<T>` ist durch Vererbung eine `BiFunction<T,T,T>`).

Der Stream-Typ deklariert drei `reduce(...)`-Methoden, die mit diesen funktionalen Schnittstellen arbeiten. Wichtig ist, dass diese Reduktionsfunktion assoziativ ist, also  $f(f(a, b), c) = f(f(a, b), c)$  gilt, denn insbesondere bei nebenläufiger Verarbeitung können beliebige Paare gebildet und reduziert werden.

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- `Optional<T> reduce(BinaryOperator<T> accumulator)`  
Reduziert alle Paare von Werten schrittweise mit dem `accumulator` auf einen Wert. Ist der Stream leer, liefert die Methode `Optional.empty()`. Gibt es nur ein Element, bildet das die Rückgabe.
- `T reduce(T identity, BinaryOperator<T> accumulator)`  
Reduziert die Werte, wobei das erste Element mit `identity` über den `accumulator` reduziert wird. Ist der Strom leer, bildet `identity` die Rückgabe.
- `<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`  
Die beiden anderen Methoden liefern als Ergebnistyp immer den Elementtyp `T` des Streams. Mit dieser Methode ist der Ergebnistyp nicht `T`, sondern `U`, da der `accumulator` die Typen `(U, T)` auf `U` abbildet. Der `combiner` ist *nur* dann nötig, wenn in der parallelen Verarbeitung zwei Ergebnisse zusammengelegt werden, sonst ist der `combiner` überflüssig. Da er aber nicht `null` sein darf, haben wir es mit dem unschönen Fall zu tun, irgend etwas übergeben zu müssen, auch wenn das bei der sequenziellen Verarbeitung unnötig ist. Der Combiner kommt immer erst nach der Funktion.



### Beispiel

Was ist das größte Element im Stream von positiven Zahlen?

```
System.out.println( Stream.of( 9, 3, 4, 11 ).reduce( 0, Math::max ) ); // 11
```

Anders als die `max(...)`-Methode vom Stream liefert `reduce(...)` direkt ein Integer und kein `Optional`, weil bei leerem Stream 0 – die Identität – zurückgegeben wird. Das ist problematisch, denn 0 ist nicht das größte Element eines leeren Streams.



### Beispiel

Reduziere einen Strom von Dimension-Objekten auf die Summe der Flächen:

```
Dimension[] dims = { new Dimension( 10, 10 ), new Dimension( 100, 100 ) };
BiFunction<Integer, Dimension, Integer> accumulator =
    (area, dim) -> area + dim.height * dim.width;
BinaryOperator<Integer> combiner = Integer::sum;
System.out.println(
    Arrays.stream( dims ).reduce( 0, accumulator, combiner )
); // 10100
```

Das Ergebnis einer Reduktion ist immer ein neuer Wert, wobei »Wert« natürlich alles sein kann, eine Zahl, ein String, ein Datum etc.

### 17.5.7 Ergebnisse in einen Container schreiben, Teil 1: collect(...)

Während die `reduce(...)`-Methoden durch den Akkumulator immer neue Werte in jedem Verarbeitungsschritt produzieren, können die Stream-Methoden `collect(...)` etwas anders arbeiten. Vereinfacht gesagt erlauben sie es, Daten eines Streams in einen Container (Datenstruktur oder auch String) zu setzen und diesen Container zurückzugeben.

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- <R> R collect(Supplier<R> resultFactory, BiConsumer<R, ? super T> accumulator, BiConsumer<R,R> combiner)

Sammelt die Elemente in einem Container. Die drei Parameter haben alle unterschiedliche Aufgaben. `resultFactory`: Der `Supplier` baut den Container auf, das ist auch der Rückgabetyp. `Accumulator`: Der `BiConsumer` bekommt zwei Argumente, den Container und das Element, und muss dann das Element dem Container hinzufügen. `combiner`: Er ist nur bei Parallelverarbeitung nötig und ist dafür verantwortlich, zwei Container zusammenzulegen.

#### Beispiel

Vier Zahlen eines Streams sollen im `LinkedHashSet` landen:

```
LinkedHashSet<Integer> set =
    Stream.of( 2, 3, 1, 4 )
        .collect( LinkedHashSet::new, LinkedHashSet::add, LinkedHashSet::addAll );
System.out.println( set ); // [2, 3, 1, 4]
```

Zur allgemeinen Initialisierung eines `LinkedHashSet` ist das natürlich noch etwas zu viel Code, hier ist es einfacher, mit `Collection.addAll(...)` die Elemente hinzuzufügen.

[zB]

Wie bei `reduce(...)` bekommt die `collect(...)`-Methode jedes Element, doch verbindet es dieses Element nicht zu einem Kombinat, sondern setzt es in Container. Daher sind auch die verwendeten Typen nicht `BinaryOperator`/`BiFunction` (bekommen etwas und liefern etwas zurück), sondern `BiConsumer` (bekommt etwas, liefert aber nichts zurück). Der Zustand sitzt damit im Container und nicht in den Zwischenverarbeitungsschritten.



### Beispiel

Eine eigene Klasse macht diese Zustandshaltung nötig, wenn zum Beispiel bei einem Strom von Ganzzahlen am Ende die Frage steht, wie viele Zahlen positiv, negativ oder null waren. Der Kollektor sieht dann so aus:

```
class NegZeroPosCollector {
    public long neg, zero, pos;

    public void accept( int i ) {
        if ( i > 0 ) pos++; else if ( i < 0 ) neg++; else zero++;
    }

    public void combine( NegZeroPosCollector other ) {
        neg += other.neg; zero += other.zero; pos += other.pos;
    }
}
```

Bei der Nutzung muss der Kollektor aufgebaut und müssen Akkumulator und Kombinierer angegeben werden:

```
NegZeroPosCollector col = Stream.of( 1, -2, 4, 0, 4 )
    .collect( NegZeroPosCollector::new,
              NegZeroPosCollector::accept,
              NegZeroPosCollector::combine );
System.out.printf( "-:%d, 0:%d, +:%d", col.neg, col.zero, col.pos ); // -:1, 0:1, +:3
```

#### 17.5.8 Ergebnisse in einen Container schreiben, Teil 2: Collector und Collectors

Es gibt zwei Varianten der `collect(...)`-Methode, und die zweite ist:

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- <R> R collect(Collector<? super T, R> collector)

Ein `Collector` fasst Supplier, Akkumulator und Kombinierer zusammen. Die statische `Collector.of(...)`-Methode baut einen `Collector` auf, den wir ebenso an `collect(Collector)` übergeben können – unser Beispiel von eben ist also mit folgendem identisch:

```
.collect( Collector.of( LinkedHashSet::new,
                      LinkedHashSet::add, LinkedHashSet::addAll ) );
```

Vorteil vom `Collector` ist also, dass dieser drei Objekte zusammenfasst (genau genommen noch ein vierter, `Collector.Characteristics` kommt dazu). Die Schnittstelle deklariert zwei

statische of(...)-Methoden und weitere Methoden, die jeweils die Funktionen vom Collector erfragen, also accumulator(), characteristics(), combiner(), finisher() und supplier(). Es ist Collector.Characteristics ein Aufzählungstyp mit den drei Elementen CONCURRENT, IDENTITY\_FINISH und UNORDERED, die Eigenschaften eines Collector beschreiben.

### Utility-Klasse Collectors

Für Standardfälle muss nicht extra mit Collector.of(...) ein Collector zusammengebaut werden, sondern es gibt eine Utility-Klasse `Collectors` mit knapp 40 statischen Methoden, die jeweils Collector-Exemplare liefern. Die gruppieren sich grob wie folgt:

- ▶ `toSet()`, `toList()`, `toXXXMap(...)` usw. für Sammlungen
- ▶ `toUnmodifiableXXX()` für unveränderbare Sammlungen, bei denen Elemente nicht hinzugefügt, gelöscht oder ersetzt werden können (seit Java 10)
- ▶ `groupingByXXX(...)` und `partitioningBy(...)` zum Bilden von Gruppen
- ▶ einfache Rückgaben wie bei `averagingLong(...)`
- ▶ Zeichenketten-Collectoren wie im Fall von `joining(...)`, die Strings eines Streams zusammenhängen.

### Beispiel

[zB]

Füge Ganzzahlen in einem Strom zu einer Zeichenkette zusammen:

```
String s = Stream.of( 192, 0, 0, 1 )
    .map( Integer::toUnsignedString )
    .collect( Collectors.joining(".")) );
System.out.println( s ); // 192.0.0.1
```

Bau ein Set<String> mit zwei Startwerten auf:

```
Set<String> set = Stream.of( "a", "b" ).collect( Collectors.toSet() );
```

Bau eine Map<Integer, String> auf, die aus zwei Schlüssel-Wert-Paaren besteht:

```
Map<Integer, String> map =
    Stream.of( "1=one", "2=two" )
    .collect( Collectors.toMap( k -> Integer.parseInt(k.split("=")[0]),
        v -> v.split("=")[1] ) );
```

```
final class java.util.stream.Collectors
```

- static <T> Collector<T,?,List<T>> `toList()`
- static <T> Collector<T,?,Set<T>> `toSet()`

Die weiteren Methoden sollten Leser in der Javadoc studieren.



### Hinweis

Ein Collector ist das einzige vernünftige Mittel, um die Stream-Elemente in einen Container zu transferieren. Von einem Seiteneffekt aus irgendeiner intermediären oder terminalen Operation sollten Entwickler absehen, da erstens der Vorteil der funktionalen Programmierung dahin ist (Operationen haben dort keine Seiteneffekte) und zweitens bei paralleler Verarbeitung bei nichtsynchronisierten Datenstrukturen Chaos ausbrechen würde. Folgendes ist also *keine* Alternative zum charmanten `Collectors.joining(".")`:

```
StringBuilder res = new StringBuilder();
Stream.of( 192, 0, 0, 1 )
    .map( Integer::toUnsignedString )
    .forEach( s ->
        res.append( res.length() == 0 ? "" : "." ).append( s ) );
System.out.print( res );
```

Wenn ein Lambda-Ausdruck Schreibzugriffe auf äußere Variablen macht, sollten die Alarmglocken schrillen. Das Gleiche gilt im Übrigen auch für Reduktionen. Zwar kann im Prinzip in einem `forEach(...)` alles zum Beispiel in einen threadsicheren Atomic-Datentyp geschrieben werden, aber wenn es auch funktional geht, dann richtig mit `reduce(...)`.

### 17.5.9 Ergebnisse in einen Container schreiben, Teil 3: Gruppierungen

Oftmals lassen sich Objekte dadurch in Gruppen einteilen, dass referenzierte Unterobjekte eine Kategorie bestimmen. Da das furchtbar abstrakt klingt, ein paar Beispiele:

- ▶ Konten haben einen Kontotyp, und gesucht ist eine Aufstellung aller Konten nach Kontotyp. Mit dem Kontotyp sind also  $n$  Konten verbunden, die alle den gleichen Kontotyp haben.
- ▶ Menschen haben einen Beruf, und gesucht ist eine Liste aller Menschen nach Berufen.
- ▶ Threads können in unterschiedlichen Zuständen (wartend, schlafend ...) sein; gesucht ist eine Assoziation zwischen Zustand und Threads in diesem Zustand.

Um dies mit der Stream-API zu lösen, lässt sich `collect(...)` mit einem besonderen Collector nutzen, den die `groupingByXXX(...)`-Methoden von `Collectors` liefern. Wir wollen uns nur mit der einfachsten Variante beschäftigen.



### Beispiel

Gib alle laufenden Threads aus:

```
Map<Thread.State, List<Thread>> map =
    Thread.getAllStackTraces().keySet().stream()
        .collect( Collectors.groupingBy( Thread::getState ) );
System.out.println( map.get( Thread.State.RUNNABLE ) );
```

Zunächst liefert `Thread.getAllStackTraces().keySet()` ein Set<Threads> mit allen aktiven Threads. Aus der Menge leiten wir einen Strom ab und nutzen einen Kollektor, der nach Thread-Status gruppiert. Das Ergebnis ist eine Assoziation zwischen `Thread.State` und einer List<Threads>.

Wenn es Methoden in der Java-API gibt, die Entwicklern Generics nur so um die Ohren hauen, dann sind es die gruppierenden Collector-Exemplare – in der Übersicht:

```
final class java.util.stream.Collectors
```

- `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`
- `static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`
- `static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)`
- `static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)`
- `static <T,K,A,D> Collector<T,?,ConcurrentMap<K,D>> groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`
- `static <T,K,A,D,M extends ConcurrentHashMap<K,D>> Collector<T,?,M> groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)`

Die Funktion, die den Schlüssel für den Assoziativspeicher extrahiert, ist in der API der `classifier`. Er muss immer angegeben werden. Wir haben uns die einfachste Methode – die erste – an einem Beispiel angeschaut; sie liefert immer eine Map von Listen. Die anderen Methoden können auch eine Map mit anderen Datenstrukturen liefern (bzw. statt Listen einfache akkumulierte Werte), dafür ist der `downstream` gedacht. Auch können die konkreten Map-Implementierungen bestimmt werden (etwa `TreeMap`), das gibt die `mapFactory` an – sonst wählt `groupingBy(Function)` für uns eine Map-Klasse aus.

### Beispiel

Sammle die Namen aller Threads nach Status ein:

```
Map<Thread.State, List<String>>> map =
    Thread.getAllStackTraces().keySet().stream()
        .collect( Collectors.groupingBy(
            Thread::getState,
```

[zB]

```
Collectors.mapping( Thread::getName, Collectors.toList() ) );
System.out.println( map.get( Thread.State.RUNNABLE ) );
```

Baue einen Assoziativspeicher auf, der den Thread-Status nicht mit den Threads selbst verbindet, sondern einfach nur mit der Anzahl an Threads:

```
Map<Thread.State, Long> map =
    Thread.getAllStackTraces().keySet().stream()
        .collect( Collectors.groupingBy( Thread::getState,
                                         Collectors.counting() ) );
System.out.println( map.get( Thread.State.RUNNABLE ) ); // 3
```

Assoziiere jeden Thread-Status mit einem Durchschnittswert der Prioritäten derjenigen Threads mit demselben Status:

```
Map<Thread.State,Double> map =
    Thread.getAllStackTraces().keySet().stream().collect(
        Collectors.groupingBy( Thread::getState,
                               Collectors.averagingInt(Thread::getPriority) ) );
System.out.println( map.get( Thread.State.RUNNABLE ) ); // 7.25
System.out.println( map.get( Thread.State.WAITING ) ); // 8.0
```

### 17.5.10 Stream-Elemente in Array oder Iterator übertragen

Kümmern wir uns abschließend um die letzten drei aggregierenden Methoden. Zwei toArray(...)-Methoden von Stream übertragen die Daten des Stroms in ein Array. iterator() auf dem Stream wiederum liefert einen Iterator mit all den Daten (die Methode stammt aus dem Oberotyp BaseStream).

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- Object[] toArray()
- <A> A[] toArray(IntFunction<A[]> generator)
- Iterator<T> iterator()

Die zwei toArray(...)-Methoden sind nötig, da im ersten Fall unklar ist, was für ein Typ das Array hat – es kommt standardmäßig nur Object[] heraus. Der IntFunction wird die Größe des Streams übergeben, und das Ergebnis ist in der Regel ein Array dieser Größe. Konstruktor-Referenzen kommen hier sehr schön zum Zuge. Primitive Arrays können nie die Rückgabe bilden, hierfür müssen wir spezielle primitive Stream-Klassen nutzen.



### Beispiel

Setze alle Elemente eines Streams in ein String-Array:

```
String[] strings = Stream.of( "der", "die", "das" ).toArray( String[]::new );
```



## 17.6 Intermediäre Stream-Operationen

Alle terminalen Operationen führen zu keinem neuen veränderten Stream, sondern beenden den aktuellen Stream. Anders sieht das bei intermediären Operationen aus, sie verändern den Stream, indem sie etwa Elemente herausnehmen, abbilden auf andere Werte und Typen oder sortieren. Jede der intermediären Methoden liefert ein neues Stream-Objekt – das haben wir in den ersten Schreibweisen durch die Konkatenation etwas verschleiert, aber vollständig sieht es so aus:

Vollständige Schreibweise	Kaskadierte Schreibweise
<pre>Stream&lt;Object&gt; s1 =     Stream.of(" ",'3',null,"2",1,""); Stream&lt;Object&gt; s2 =     s1.filter( Objects::nonNull ); Stream&lt;String&gt; s3 =     s2.map( Objects::toString ); Stream&lt;String&gt; s4 =     s3.map( String::trim ); Stream&lt;String&gt; s5 =     s4.filter( s -&gt; !s.isEmpty() ); Stream&lt;Integer&gt; s6 =     s5.map( Integer::parseInt ); Stream&lt;Integer&gt; s7 = s6.sorted(); s7.forEach( System.out::println );</pre>	<pre>Stream.of(" ",'3',null,"2",1,"")     .filter( Objects::nonNull )     .map( Objects::toString )     .map( String::trim )     .filter( s -&gt; ! s.isEmpty() )     .map( Integer::parseInt )     .sorted()     .forEach( System.out::println );</pre>



Tabelle 17.4 Ausführliche und kompakte Schreibweise im Vergleich

### Hinweis

Keine Strommethode darf die Stromquelle modifizieren, da das Ergebnis sonst unbestimmt ist. Veränderungen treten nur entlang der Kette von einem Strom zum nächsten auf. Wir haben das schon in [Abschnitt 17.5.8, »Ergebnisse in einen Container schreiben, Teil 2: Collector und Collectors«](#), diskutiert.

### 17.6.1 Element-Vorschau

Eine einfache intermediäre Operation ist `peek(...)`; sie darf sich das aktuelle Element während des Durchlaufs anschauen.

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- `Stream<T> peek(Consumer<? super T> action)`



#### Beispiel

Schau in den Strom vor und nach einer Sortierung:

```
System.out.println( Stream.of( 9, 4, 3 )
    .peek( System.out::println ) // 9 4 3
    .sorted()
    .peek( System.out::println ) // 3 4 9
    .collect( Collectors.toList() ) );
```

### 17.6.2 Filtern von Elementen

Eine der wichtigsten Stream-Methoden ist `filter(...)`: Sie liefert alle Elemente im Stream, die einem Kriterium genügen, und ignoriert alle anderen, die nicht diesem Kriterium genügen.

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- `Stream<T> filter(Predicate<? super T> predicate)`



#### Beispiel

Gib alle Wörter aus, bei denen zwei beliebige Vokale hintereinander vorkommen:

```
Stream.of( "Moor", "Aha", "Meister" )
    .filter( Pattern.compile( "[aeiou]{2}" ).asPredicate() )
    .forEach( System.out::println );      // Moor Meister
```

### 17.6.3 Statusbehaftete intermediäre Operationen

Die allermeisten intermediären Operationen können Elemente direkt während des Durchlaufs bewerten und verändern, sodass keine speicherintensive Zwischenspeicherung nötig ist. Einige intermediäre Operationen haben jedoch einen Status, dazu zählen `limit(long) –`

begrenzt den Strom auf eine gewisse Anzahl maximaler Elemente –, skip(long) – überfährt eine Anzahl Elemente –, distinct() – löscht alle doppelten Elemente – und sorted(...) – sortiert den Strom.

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- Stream<T> limit(long maxSize)
- Stream<T> skip(long n)
- Stream<T> distinct()
- Stream<T> sorted()
- Stream<T> sorted(Comparator<? super T> comparator)

### Beispiel 1

[zB]

Was ist das längste Wort der Liste?

```
String longest =
Stream.of( "Autos", "können", "nicht", "versaut", "genug", "sein" )
    .sorted( Comparator.comparing( String::length ).reversed() )
    .findFirst().get();
System.out.println( longest ); // versaut
```

Alternativ mit max(...):

```
String longest = ...
    .max( Comparator.comparing( String::length ) )
    .get();
```

### Beispiel 2

[zB]

Zerlege eine Zeichenkette in Teilzeichenketten, bilde diese auf das ersten Zeichen ab, und lösche aus dem Stream doppelte Elemente:

```
List<String> list = Pattern.compile( " " ).splitAsStream( "Pu Po Aha La" )
    .map( s -> s.substring(0, 1) )
    .peek( System.out::println ) // P P A L
    .distinct()                // \/\/\/
    .peek( System.out::println ) //  P A L
    .collect( Collectors.toList() );
System.out.print( list );                                // [P, A, L]
```

peek(...) macht gut deutlich, wie die Elemente vor und nach dem Anwenden von distinct() aussehen.

**Tipp**

Eine Methode wie `skip(...)` sieht auf den ersten Blick unschuldig aus, kann aber ganz schön auf den Speicherbedarf und die Performance gehen, wenn parallele Ströme mit Ordnung (etwa durch Sortierung) ins Spiel kommen. Bei einem `stream.parallel().sorted().skip(10000)` müssen trotzdem alle Elemente erst sortiert werden, damit die ersten 10.000 übersprungen werden können. Sequenzielle Ströme bzw. Ströme ohne Ordnung (die liefert die Stream-Methode `unordered()`) sind ungleich schneller, aber natürlich nicht in jedem Fall möglich.

#### 17.6.4 Präfix-Operation

Unter einem *Präfix* verstehen wir eine Teilfolge eines Streams, die beim ersten Element beginnt. Wir können mit `limit(long)` selbst ein Präfix generieren, doch im Allgemeinen geht es darum, eine Bedingung zu haben, die alle Elemente eines Präfix-Streams erfüllen müssen, und wenn die Bedingung für ein Element nicht mehr gilt, dann den Stream zu beenden. Dafür gibt es die Methoden `takeWhile(...)` und `dropWhile(...)`:

- ▶ default Stream<T> `takeWhile(Predicate<? super T> predicate)`
- ▶ default Stream<T> `dropWhile(Predicate<? super T> predicate)`

Die deutsche Übersetzung von `takeWhile(...)` wäre »nimm, solange predicate gilt« und `dropWhile(...)` »lass fallen, solange predicate gilt«.

**Beispiel**

Der Stream soll bei Eintreffen des Wortes »Trump« sofort enden:

```
new Scanner( "Dann twitterte Trump am 7. Nov. 2012: "
    + "'It's freezing and snowing in New York--we need global warming!' " )
    .useDelimiter( "\\\P{Alpha}+" ).tokens()
    .takeWhile( s -> !s.equalsIgnoreCase( "trump" ) )
    .forEach( System.out::println ); // Dann twitterte
}
```

Der Stream soll nach dem längsten Präfix beginnen, der dann endet, wenn eine Zahl kleiner gleich 0 ist:

```
Stream.of( 1, 2, -1, 3, 4, -1, 5, 6 )
    .dropWhile( i -> i > 0 )           // !(i>0), also i<=0 wird übersprungen
    .forEach( System.out::println ); // -1 3 4 -1 5 6
```

Das Element, das das Prädikat erfüllt, ist selbst das erste Element im neuen Stream. Wir können es mit `skip(1)` überspringen.

Erfüllt schon bei `takeWhile(...)` das erste Element nicht das Prädikat, so ist der Stream leer. `takeWhile(...)` und `dropWhile(...)` können zusammen verwendet werden: So liefert `Stream.of(1, 2, -1, 3, 4, -1, 5, 6).dropWhile(i -> i > 0).skip(1).takeWhile(i -> i > 0).forEach(System.out::println);` die Ausgaben 3 4.

### Tipp

Präfixe sind nur für geordnete Streams sinnvoll. Und wenn Streams parallel sind, müssen sie für die Präfixberechnung wieder in Reihe gebracht werden, das ist eine eher teure Operation.



## 17.6.5 Abbildungen

Die Abbildung von Elementen im Strom auf neue Elemente ist eines der mächtigsten Werkzeuge der Stream-API. Zu unterscheiden sind drei Typen von Abbildungsmethoden:

- ▶ Die einfachste Variante ist `map(Function)`. Die Funktion bekommt das Stromelement als Argument und liefert ein Ergebnis, das dann in den Strom kommt. Hierbei kann sich der Typ ändern, wenn die Funktion nicht den gleichen Typ gibt, wie sie nimmt. Die Funktion wird nach und nach auf jedem Element angewendet, die Reihenfolge ergibt sich aus der Ordnung des Stroms.
- ▶ Die drei Methoden `mapToInt(IntFunction)`, `mapToLong(LongFunction)` und `maptoDouble(DoubleFunction)` arbeiten wie `map(Function)`, nur liefern die Funktionen einen primitiven Wert, und das Ergebnis ist ein primitiver Stream – diese Typen schauen wir uns später an.
- ▶ `flatMapXXX(XXXFunction)`-Methoden ersetzen ebenfalls Elemente im Stream, mit dem Unterschied, dass die Funktionen alle selbst Stream-Objekte liefern, deren Elemente dann in den Ergebnisstrom gesetzt werden. Der Begriff »flat« (engl. für »flach«) kommt daher, dass die »inneren« Streams nicht selbst alle als Elemente in den Ursprungs-Stream kommen (sozusagen ein `Stream<Stream>`), sondern flachgeklopft werden. Anwendungsbeispiel sind in der Regel Szenarien wie »*n* Spieler assoziieren *m* Gegenstände, gesucht ist ein Strom aller Gegenstände aller Spieler«. Die Funktion kann `null` liefern, wenn nichts in den Stream gelegt werden soll.

```
interface java.util.stream.Stream<T>
extends BaseStream<T, Stream<T>>
```

- ▶ `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
- ▶ `IntStream mapToInt(IntFunction<? super T> mapper)`
- ▶ `LongStream mapToLong(LongFunction<? super T> mapper)`
- ▶ `DoubleStream mapToDouble(DoubleFunction<? super T> mapper)`
- ▶ `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`

- ▶ IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper)
- ▶ LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper)
- ▶ DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper)

[zB]

### Beispiel

Konvertiere ein String-Array mit Zahlen in ein long-Array:

```
String[] numbers = { "1", "2", "3" };
long[] parseInts = Stream.of( numbers ).mapToLong( Long::parseLong ).toArray();
```

[zB]

### Beispiel

Die Methode `Locale.getAvailableLocales()` liefert ein Array von `Locale`-Objekten, die ein System unterstützt. Wir interessieren uns für alle Ländercodes:

```
Stream.of( Locale.getAvailableLocales() )
    .map( Locale::getCountry )
    .distinct()
    .forEach( System.out::println );
```

Über ein `Locale`-Objekt erfragt `DateFormatSymbols.getInstance(locale).getWeekdays()` die Namen der Wochentage. Wir interessieren uns als Ergebnis für alle Wochentage aller installierten Gebiete:

```
Stream.of( Locale.getAvailableLocales() )
    .flatMap( l -> Stream.of( DateFormatSymbols.getInstance( l ).getWeekdays() ) )
    .filter( s -> ! s.isEmpty() )
    .distinct()
    .forEach( System.out::println );
```

[+]

### Tipp

Die an `flatMap(...)` übergebene Funktion muss als Ergebnis einen Stream liefern oder null, wenn nichts passieren soll. Es ist nicht verkehrt, immer einen Stream zu liefern und statt null der Funktion einen leeren Stream mit `Stream.empty()` zurückgeben zu lassen. Anwendungen für diese leeren Ströme kommen aus folgendem Szenario: In der API gibt es Methoden, die statt Arrays/Sammlungen nur null geben. Nehmen wir an, `result` ist so eine Rückgabe, die null oder ein Array sein kann, dann bekommen wir einen Stream wie folgt:

```
Stream<...> flatStream = Optional.ofNullable( result )
    .map( Stream::of ).orElse( Stream.empty() );
```

Die Fallunterscheidung, ob die Sammlung null ist, ist hier funktional mit `Optional` gelöst, alternativ `result != null ? result : Stream.empty()`. Das statische `Stream.of(...)`, das wir hier über eine Methodenreferenz nutzen, funktioniert für ein Array; für einen Stream aus einer

Sammlung wäre `result.stream()` nötig. Fassen wir das in einem Beispiel zusammen. Die Methode `getEnumConstants()` auf ein Class-Objekt liefert ein Array von Konstanten, wenn das Class-Objekt eine Aufzählung repräsentiert – andernfalls ist das Array nicht etwa leer, sondern `null` (so lässt sich unterscheiden, ob das Class-Objekt entweder keine Elemente hat oder überhaupt kein Aufzählungstyp ist). Wir wollen von drei Class-Objekten alle Konstanten einsammeln und ausgeben:

```
Stream.of( Object.class, Thread.State.class, DayOfWeek.class )
    .flatMap( clazz -> Optional.ofNullable( clazz.getEnumConstants() )
        .map( Stream::of ).orElse( Stream.empty() ) )
    .forEach( System.out::println );
```

`null` ist oft ein ungebetener Gast, und die Stream-Methode `ofNullable(...)` hilft, Fehler zu vermeiden. Nehmen wir Folgendes, um alle Koordinaten in einen Stream zu bekommen:

```
Stream.of( null, new Point( 1, 2 ), null, new Point( 3, 4 ) )
    .flatMap( q -> Stream.of( q.x, q.y ) )
    .forEach( System.out::println );
```

Es wird eine `NullPointerException` folgen bei `flatMap(...)`. Mit `filter(...)` könnten wir im Vorfeld jede `null` ausblenden, doch eine andere Lösung wäre:

```
Stream.of( null, new Point( 1, 2 ), null, new Point( 3, 4 ) )
    .flatMap( p -> Stream.ofNullable( p ).flatMap( q -> Stream.of( q.x, q.y ) ) )
    .forEach( System.out::println );
```

## 17.7 Zum Weiterlesen

Der Band »Java SE 9 Standard-Bibliothek« behandelt ausführlich Datenstrukturen und Algorithmen und diskutiert gleichzeitig, wie Threads und Datenstrukturen effizient zusammenarbeiten. Ein weiteres Kapitel präzisiert die Stream-API.



# Kapitel 18

## Einführung in grafische Oberflächen

»Wenn die Reklame keinen Erfolg hat, muss man die Ware ändern.«  
– Edgar Faure (1908–1988)

### 18.1 GUI-Frameworks

*Benutzeroberfläche* oder *User Interface*, kurz *UI*, nennen wir die Schnittstelle zwischen Mensch und Anwendung.

#### 18.1.1 Kommandozeile

Am Anfang der Mensch-Maschine-Interaktion stand die *Kommandozeile* (engl. *command line interface*, kurz *CLI*). Java bietet hierfür nur rudimentäre Unterstützung:

- ▶ Die Ausgaben auf der Kommandozeile/Shell laufen über `System.out` und `System.err`, die Eingaben über `System.in`.
- ▶ Einige Shells bieten Positionierungen eines Cursors oder Farben, doch da dies nicht plattformunabhängig ist, bietet Java keine Unterstützung an. Das kann durch quelloffene Erweiterungen jedoch ermöglicht werden.
- ▶ Die an das Java-Programm übergebenen Kommandozeilenargumente landen bei der Methode `main(String[] args)` in der Parametervariablen `args`.
- ▶ Rückgaben vom Java-Programm an die Shell sind über einen Methodenaufruf `System.exit(int)` möglich.
- ▶ Es fehlen das komfortable Parsen der Programmargumente und Auswerten der Optionen. Quelloffene Bibliotheken schaffen hier Abhilfe.

#### 18.1.2 Grafische Benutzeroberfläche

Das Aufkommen von grafischen Benutzeroberflächen oder Graphical User Interfaces, kurz *GUI*, ab den 1970er Jahren verdrängte immer mehr die Kommandozeile, und Benutzer konnten die Anwendungen mit Fenstern, Symbolen und Interaktionskomponenten bedienen.

Die Java-Plattform hat sich zum Ziel gesetzt, plattformunabhängige Softwareentwicklung zu unterstützen, und dazu gehört auch eine Bibliothek zur Gestaltung grafischer Oberflächen. Eine Bibliothek sollte dabei im Wesentlichen die folgenden Bereiche abdecken:

- ▶ Sie beherrscht das Zeichnen grafischer Grundelemente wie Linien und Polygone, setzt performant Grafiken, ermöglicht das Setzen von Farben und bietet eine ordentliche Darstellung von Text.
- ▶ 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.

Zur GUI-Entwicklung in Java gibt es Frameworks, also Bibliotheken, mit denen sich grafische Oberflächen programmieren lassen. Drei Frameworks bringt Java gleich mit.

### 18.1.3 Abstract Window Toolkit (AWT)

Als Sun Java 1995 veröffentlichte, gehörte ein Framework zur Gestaltung (einfacher) grafischer Oberflächen schon gleich mit zur Standard-API: das *Abstract Window Toolkit* (AWT). Es bietet Methoden für Primitivoperationen zum Zeichnen, zur Ereignisbehandlung und einen einfachen Satz von GUI-Komponenten, wie Beschriftungen, Schaltflächen, Textfelder. Das AWT ist eine portable GUI-Bibliothek, die auf nativen Komponenten des Betriebssystems basiert und folglich auch hervorragende Performance und im Grunde gute Portabilität bietet.

### 18.1.4 Java Foundation Classes und Swing

Das AWT bietet nur lächerliche acht Komponenten,<sup>1</sup> und anspruchsvolle Oberflächen sind damit nur mit viel Handarbeit möglich. Sun entschloss sich daher schon früh, das AWT zu neuern, und so flossen neue Komponenten – die so genannten *Swing-Komponenten* – sowie eine neue Java 2D-API und weitere Dienste fest in Java 1.2 ein (zur Erinnerung, das war Ende 1998). Die Gesamtheit von AWT und Swing bilden die *Java Foundation Classes*, kurz JFC. Sie sind bis heute Teil der Java SE und seit Java 9 ein eigenes Modul.

### 18.1.5 JavaFX

Seit etwa 20 Jahren realisieren Swing bzw. die Java Foundation Classes grafische Anwendungen unter Java. Zwar sind die Java Foundation Classes mächtig, aber sie haben sich in den letzten Jahren nicht großartig weiterentwickelt. Insbesondere gibt es Lücken im Bereich ausgereifter Komponenten sowie Medienunterstützung und Animation, etwas, was bei modernen grafischen Oberflächen heutzutage gefragt ist. Anstatt in die Weiterentwicklung der JFC zu investieren, hat sich Sun/Oracle für eine komplette Neuentwicklung der GUI-Ebene entschieden, die nichts mehr mit Swing/AWT gemeinsam hat – herausgekommen ist *JavaFX*.

---

<sup>1</sup> Button, Checkbox, Choice, Label, List, Scrollbar, TextArea, TextField, die Container und eine generische Zeichenfläche Canvas einmal außen vor gelassen

JavaFX ist eine Komplettlösung mit einer API für:

- ▶ GUI-Komponenten
- ▶ HTML/CSS/JavaScript mit eingebettetem Webbrowser
- ▶ Animationen
- ▶ Video
- ▶ Audio
- ▶ 2D und 3D

Da JavaFX komplett alle APIs für moderne Oberflächen anbietet und auch nicht von AWT/Swing abhängig ist, bildet JavaFX einen kompletten *Media-Stack*. Die Betonung liegt auf »Media«, denn die Java Foundation Classes können keine Videos einbinden oder abspielen. Anders als AWT ist die JavaFX-Implementierung auf der Höhe der Zeit und greift direkt auf alle 2D-/3D-Fähigkeiten moderner Grafikkarten zurück. So kann mit JavaFX alles das programmiert werden, was bisher eher mit Flash gemacht wurde, wohl aber fehlen noch die tollen Entwicklertools. Es gibt Plugins für Adobe Photoshop und Illustrator, mit denen Grafiken und Pfade exportiert werden können, aber eben keine ganzen Animationen, die etwa mit Adobe Flash erzeugt wurden.

### Geschichte von JavaFX: JavaFX 1, JavaFX 2, JavaFX 8, OpenJFX 11

Ursprünglich wollte Sun/Oracle JavaFX als Flash-Ersatz im Internet positionieren, doch dafür ist die Kombination *HTML5 + CSS3 + JavaScript* zu attraktiv. JavaFX ist in erster Linie eine großartige GUI-Bibliothek für klassische Client-Anwendungen, die langsam auch auf mobile Endgeräte vorrückt. So nahm die Entwicklung auch unterschiedliche Richtungen.

JavaFX ist schon sehr lange in Entwicklung, und viele interne Swing-Entwickler wurden bei Sun/Oracle auf das Projekt angesetzt – umgekehrt passierte bei den Java Foundation Classes nicht mehr viel, Bugfixes kommen aber immer noch brav. Im Jahr 2007 wurde JavaFX auf der JavaOne-Konferenz vorgestellt, Ende 2008 erschien das Release JavaFX 1.0 zusammen mit der Programmiersprache *JavaFX Script*. Die besondere Sprache machte es möglich, auf einfache Weise hierarchische Objektgraphen aufzubauen, und bot eine nette Syntax für Object-Binding, sodass Zustände synchronisiert werden konnten.

Im Oktober 2011 erschien JavaFX 2.0 mit vielen Neuerungen und auch Änderungen. So wurde JavaFX Script entfernt, denn Oracle wollte keine weitere Programmiersprache aufbauen, sondern eine pure Java-API, die Entwickler dann von unterschiedlichen existierenden Skriptsprachen aus ansprechen können. Das war sicherlich eine gute Entscheidung, denn unter JavaScript und Groovy sieht das sehr schlank aus, fast wie mit JavaFX Script. Mit dem Update auf JavaFX 2.0 änderte sich auch die API, sodass alter Code angefasst werden musste. Heute ist JavaFX 1 veraltet, genauso wie die Literatur.

JavaFX entwickelte sich immer mehr zur Alternative zu Swing/AWT, und so integrierte Oracle im August 2012 das Java FX 2.2 SDK und die Runtime in das JRE/JDK 7 Update 6. Der Schritt war ungewöhnlich, denn so große Ergänzungen waren bisher als Update im JRE/JDK noch nie gemacht worden. Neben der Integration bewegte sich auch das ehemals geschlossene JavaFX in Richtung Open Source und mündete in der Implementierung *OpenJFX*. Mit dem OpenJDK und OpenJFX lässt sich ein komplett freies Java-System mit GUI-Stack unter der GPL bauen, was strikte Linux-Distributionen freuen wird. Die Offenheit führte schon zu sehr spannenden Experimenten, etwa einer Java-Version für iOS.

Mit Java 8 zog auch JavaFX 8 ein, der nächste Sprung mit 3D-Unterstützung. In Java 11 wiederum wurde JavaFX entfernt, und die Entwicklung findet im OpenJFX statt – das wird als Modul eingebunden und ist eine externe Bibliothek wie jede andere auch.

### 18.1.6 SWT (Standard Widget Toolkit) \*

Es gibt Anwendungsfälle, in denen weder AWT noch Swing noch JavaFX passen. Swing ist zwar sehr mächtig, doch kostet es auch einige Ressourcen. AWT ist schlank und recht schnell, bietet aber wenige Komponenten. Das AWT ist bei Geräten mit wenig Speicher, etwa Handheld-Geräten, noch eine interessante Variante, aber alles sieht so aus, als ob die Entwicklung bei JavaFX weitergeht.

Im Jahr 1998 begann IBM mit der Entwicklung des Nachfolgers von *VisualAge for Java* (das in Smalltalk geschrieben war). Für die neue Entwicklungsumgebung – aus der später *Eclipse* wurde – benötigte das Entwicklerteam Oberflächenelemente. Swing war weit davon entfernt, zügig, speichereffizient und korrekt zu sein, und Sun führte schon damals die AWT-Entwicklung nicht erkennbar weiter. So entschied IBM, mit dem *SWT (Standard Widget Toolkit)* eine Alternative zum AWT zu entwickeln, die Möglichkeiten eines nativen grafischen Systems nutzen kann. Dazu zählen neben Komponenten wie Schaltflächen und Tabellen auch integrierte Fähigkeiten wie Drag & Drop, Zwischenablage, Drucken und ActiveX-Integration (unter Windows). Dabei nutzt SWT keinen Anteil von AWT, nicht einmal die grafischen Primitivoperationen oder Java 2D. Wegen dieser engen Kopplung an das System müssen die SWT-Applikationen mit einer dynamischen Bibliothek (DLL unter Windows) ausgeliefert werden. Eine Umsetzung gibt es heute außer für Windows (auch Windows CE) und macOS etwa auf diversen Motif-Plattformen, GTK (kein Qt) und QNX/Photon. (Wer nur unter GNOME und GTK+ programmiert, der sollte einen Blick auf Java-GNOME unter <http://java-gnome.sourceforge.net> werfen.)

Ursprünglich war das SWT (<http://www.eclipse.org/swt>) ausschließlich für die neue Entwicklungsumgebung gedacht. Mittlerweile erfreut sich die Swing-Alternative aber größerer Beliebtheit und ist inzwischen von der Entwicklungsumgebung losgelöst. Insbesondere für mobile Endgeräte (Windows CE) ist das SWT sehr performant, da die Bibliothek auf einem kleinen nativen Kern aufbaut und recht leichtgewichtig ist. Die Eclipse *Rich Client Platform*

(RCP) ist ein großes Framework für komplexe grafische Oberflächen auf der Basis von SWT, das Aufgaben wie Konfigurationen für Benutzer oder Verteilung und Aktualisierung von Haus aus realisiert.

Das immer wieder vorgebrachte Argument, dass mit SWT die Java-Oberflächen genauso aussehen wie native Oberflächen der jeweiligen Plattform, ist fragwürdig. SWT-Oberflächen sehen anders aus, etwa bei den Reitern oder Menüs, und Swing kommt dem nativen Aussehen – insbesondere ab Java Version 6 unter Windows – eigentlich näher. Dem Anwender ist das ziemlich egal. Selbst für konservativere Bereiche wie Office haben sich Anwender daran gewöhnt, dass Microsoft in jeder Produktversion die Optik anpasst. *Microsoft Office 2007* sieht völlig anders aus als *Office 2003*, das sieht wieder anders aus als *Office 2016*, und der *Windows Media Player* folgt auch nicht der Designsprache. Die Avantgarde der Anwender passte schon immer die Oberflächen an; das »Skinning« unterstützen heute bereits viele Anwendungen.

Ob die IBM-Entwickler mit dem mittlerweile schnellen Swing-Framework die gleiche Entscheidung treffen werden, wird immer wieder diskutiert – ebenso wie die Frage, ob JavaFX irgendwann einmal Eclipse realisiert, ein Prototyp existiert schon. Von der Geschwindigkeit und vom Speicherverbrauch her wäre es denkbar, aber sehr unwahrscheinlich, dass die Entwickler Mühe in eine komplette Umstellung investieren, zumal in SWT auch Swing-Applikationen und Java 2D ans Laufen gebracht werden können.

## 18.2 Deklarative und programmierte Oberflächen

Grundsätzlich können grafische Oberflächen über eine Programm-API aufgebaut werden oder in einer deklarativen Beschreibung spezifiziert werden:

- ▶ *Programmierte Oberflächen*: Der traditionelle 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 und mithilfe eines Layouts explizit angeordnet werden. Komplexe Oberflächen bestehen dann aus fast unerwartbaren Mengen von Programmcode zum Aufbau der Komponenten, zum Setzen der Eigenschaften und Platzieren auf dem Container. Die Änderung des Layouts ist natürlich sehr schwierig, da mitunter auch für kleinste Änderungen viel Quellcode bewegt wird. In der Versionsverwaltung sieht das mitunter schrecklich aus.
- ▶ *Deklarative Oberflächen* stehen im Gegensatz zu den programmierten Oberflächen. Bei ihnen sind die Beschreibung des Layouts und die Anordnung der Komponenten nicht in Java ausprogrammiert, sondern in einer externen Ressourcendatei beschrieben. Das Format kann etwa XML sein und spiegelt wider, wie das Objektgeflecht aussieht. Eine Ablaufumgebung liest die Ressourcendatei und übersetzt die Deklarationen in ein Geflecht von

GUI-Komponenten. Im Hauptspeicher steht dann am Ende das Gleiche wie bei der programmierten GUI: ein Objektgraph.

### Das andere Ufer

Microsoft erkannte ebenfalls die Notwendigkeit einer deklarativen Beschreibung von Oberflächen und nutzt intensiv XAML (*Extensible Application Markup Language*). Gleichzeitig gibt es leistungsfähige Tools und Designer für XAML. Die Firma Soyatec versucht sich mit *eFace* an einer Java-Realisierung (<http://www.soyatec.com/eface>).

#### 18.2.1 GUI-Beschreibungen in JavaFX

JavaFX unterstützt beide Möglichkeiten zum Aufbau von grafischen Oberflächen. Zum einen ist da die klassische API, die Knoten in einen Baum hängt, viel interessanter ist aber der deklarative Ansatz, der sehr schön Präsentation und Logik trennt. JavaFX selbst bietet eine Beschreibung auf XML-Basis, genannt *FXML*. XML ist selbst hierarchisch, kann also die grundlegende hierarchische Gliederung einer GUI in Containern und Komponenten sehr gut abbilden.

Neben FXML gibt es weitere proprietäre Beschreibungen und Mischformen. Eine davon ist *FXGraph* vom Projekt *efxclipse* (<http://www.eclipse.org/efxclipse>), einer JavaFX-Unterstützung in Eclipse. Die Beschreibung ist eine DSL<sup>2</sup> und definiert den Objektgraphen, der im Hintergrund in FXML umgesetzt wird. FXGraph ist kompakter als FXML und erinnert entfernt an JSON. Auch kann die JavaFX-API in alternativen Sprachen angesprochen werden; JavaScript und weitere Skriptsprachen wie Groovy (mit der Hilfe von GroovyFX<sup>3</sup>) oder Scala (zusammen mit ScalaFX<sup>4</sup>) zeigen interessante Wege auf. Allerdings mischt sich dann doch wieder schnell die Deklaration der GUI mit Logik, was die Trennung zwischen Präsentation und Logik aufweicht. Es ist guter Stil, die Beschreibung der Benutzerschnittstelle und der Logik zu trennen, um auch Tests leichter zu realisieren.

#### 18.2.2 Deklarative GUI-Beschreibungen für Swing?

Für AWT und Swing hat sich für deklarative Oberflächen in den letzten Jahren kein Standard gebildet, und Oberflächen werden heute noch so programmiert wie vor 15 Jahren. Dass Swing-Oberflächen immer programmiert werden müssen, hält auf, auch wenn ein GUI-BUILDER heutzutage die Schmerzen minimiert. Über die WYSIWYG-Oberfläche (*What You See Is What You Get*) wird in der Regel das Layout mit allen Komponenten zusammengeklickt, und

2 Eine *Domain Specific Language* (DSL) ist eine »Spezialsprache«, die ein exklusives Format für einen klar abgegrenzten Anwendungsfall definiert.

3 <http://groovyfx.org>

4 <http://www.scalafx.org>

im Hintergrund erzeugt der GUI-Builder den Programmcode. Für die Laufzeitumgebung hat sich also nichts verändert, aber für uns schon.

Um auch in Swing in die Richtung von deklarativen Oberflächen zu kommen, gibt es unterschiedliche Open-Source-Lösungen, da Oracle nichts im Angebot hat.

- ▶ *Swixml* (<http://www.swixml.org>) nutzt das XML-Format zur Beschreibung von GUIs und bildet jede Swing-Klasse auf ein XML-Element ab. Später nutzt Swixml dann SAX und JDOM, um die XML-Datei einzulesen und zu repräsentieren und um zur Laufzeit einen Swing-Komponentenbaum aufzubauen. Die Folien unter <http://www.swixml.org/slides.html> geben einen Einblick in die Möglichkeiten. Seit Mitte 2011 wird Swixml nicht mehr erweitert.
- ▶ Eine weitere Lösung zur deklarativen Beschreibung von Swing-Oberflächen bietet der *Swing JavaBuilder* (<https://github.com/jacek99/javabuilders>). Die Open-Source-Bibliothek steht unter der Apache-Lizenz und nutzt statt XML das kompaktere YAML-Format, dessen Schreibweise noch weiter verkürzt wurde. Das letzte Release stammt von Ende 2011, eine Weiterentwicklung ist unwahrscheinlich.

Die Inaktivität lässt sich entweder damit erklären, dass die Produkte perfekt sind oder dass sich Entwickler mit den klassischen codegenerierenden GUI-Buildern anfreunden konnten oder dass die Tools mit dem Aufkommen von JavaFX einfach unattraktiv werden.

## 18.3 GUI-Builder

Mit einem GUI-Builder lassen sich grafische Oberflächen über ein grafisches Werkzeug einfach aufbauen. In der Regel bietet ein GUI-Builder eine Zeichenfläche und eine Symbolleiste mit Komponenten, die per Drag & Drop angeordnet werden. Zentral bei dem Ansatz ist das WYSIWYG-Prinzip, dass nämlich im Designschritt schon abzulesen ist, wie die fertige Oberfläche aussieht.

Ein GUI-Builder erzeugt eine Repräsentation der grafischen Oberfläche, die im Prinzip auch von Hand zu erstellen wäre – allerdings ist der Aufwand sehr groß und für jeden nachvollziehbar, der schon einmal in HTML eine neue Tabellenspalte hinzugefügt hat. Es gibt immer wieder Diskussionen über das Für und Wider, doch ist es wie mit allen Tools: Richtig eingesetzt kann ein GUI-Builder viel Arbeit sparen.

### 18.3.1 GUI-Builder für JavaFX

Der *JavaFX Scene Builder* war ein Werkzeug von Oracle und »A Visual Layout Tool for JavaFX Applications«. Da Oracle jedoch die Lust an Client-Technologien verloren hat, wurde das Programm quelloffen und vom Unternehmen Gluon aufgegriffen, das es seitdem weiterent-

wickelt. Die in *Scene Builder* umbenannte Software kann unter <http://gluonhq.com/products/scene-builder/> bezogen und installiert werden. Danach stehen komfortable Werkzeuge zum Entwurf von Oberflächen und deren Verschönerung mit CSS zur Verfügung.

Eclipse und IntelliJ bieten von Haus aus keine Unterstützung beim Entwurf grafischer Oberflächen. Es lässt sich allerdings der Scene Builder aus der IDE heraus aufrufen. Das gibt uns direkte Möglichkeiten, JavaFX spielerisch zu erfahren.<sup>5</sup>



Für Eclipse hilft bei der Integration das quelloffene Eclipse-Plugin *e(fx)clipse* unter <https://www.eclipse.org/efxclipse/>.<sup>6</sup> Das Plugin bietet erweitert gibt weitere Quick-Fixes und einen Editor zum Umgang mit FXML-Dateien.

### 18.3.2 GUI-Builder für Swing

Eclipse integriert standardmäßig keinen GUI-Builder, doch auf dem Markt gibt es Plugins. Der *WindowBuilder* (<https://www.eclipse.org/windowbuilder/>) von Google hat sich hier als De-facto-Standard etabliert. Über den Update-Mechanismus von Eclipse wird er installiert. Eine Installationsanleitung findet sich auf der Webseite. Neben Swing bringt der WindowBuilder gleich noch GWT, SWT und XWT (Eclipse XML Window Toolkit) mit.



IntelliJ bringt einen GUI-Builder mit, <http://www.jetbrains.com/help/idea/gui-designer-basics.html> dokumentiert die Arbeit.

## 18.4 Mit dem Eclipse WindowBuilder zur ersten Swing-Oberfläche

Ohne uns intensiv mit den Swing-Klassen auseinanderzusetzen, wollen wir ein erstes Beispiel programmieren und das Swing-Wissen sozusagen im Vorbeigehen mitnehmen. Wir wollen im Folgenden ein Programm *Bing* entwickeln. Mit einem Schieberegler können wir die Zeit einstellen, und nach dem Aufruf erscheint eine Meldung auf dem Bildschirm. Wer vor lauter Java immer vergisst, den Teebeutel aus der Tasse zu nehmen, für den ist diese Applikation genau richtig.

### 18.4.1 WindowBuilder installieren

Einige Entwicklungsumgebungen bringen zur Gestaltung grafischer Oberflächen einen GUI-Builder mit. Eclipse hat kein Werkzeug mit an Bord, doch es gibt es freies Plugin, das wir installieren können.

<sup>5</sup> Didaktiker nennen das »exploratives Lernen«.

<sup>6</sup> [http://docs.oracle.com/javafx/scenebuilder/1/use\\_java\\_ides/sb-with-eclipse.htm](http://docs.oracle.com/javafx/scenebuilder/1/use_java_ides/sb-with-eclipse.htm)

Nach dem Start von Eclipse wählen wir im Menü HELP • ECLIPSE MARKETPLACE.

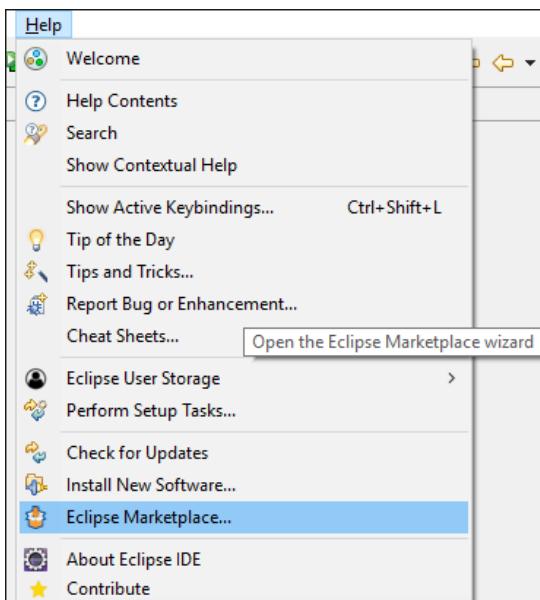


Abbildung 18.1 »Marketplace über« das Hilfemenü öffnen

Anschließend suchen wir nach »WindowBuilder«.

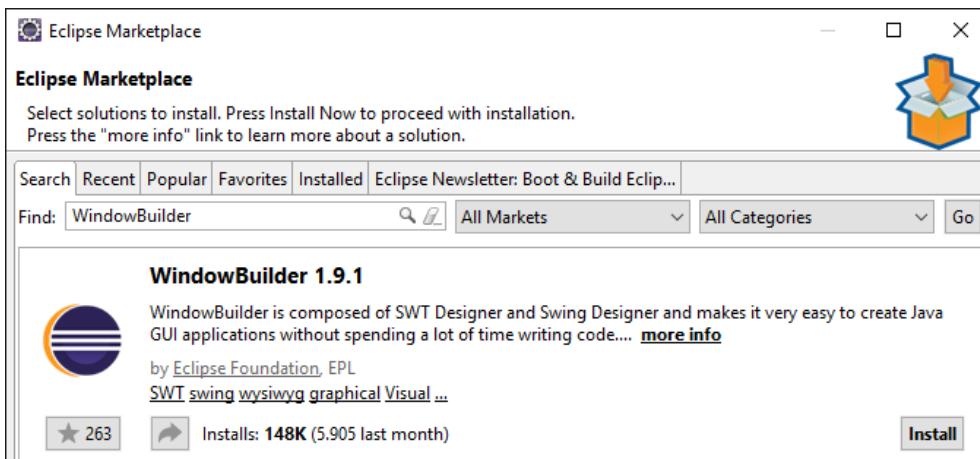


Abbildung 18.2 WindowBuilder über den Eclipse-Marktplatz installieren

Ein Klick auf INSTALL startet die Installation. Nach einem Neustart von Eclipse ist das Plugin aktiviert, und es kann mit dem GUI-Builder losgehen.

### 18.4.2 Mit WindowBuilder eine GUI-Klasse hinzufügen

Der WindowBuilder legt neue Klassen an, ein Java-Projekt muss schon vorhanden sein. Wir gehen zunächst auf FILE • NEW • OTHER.

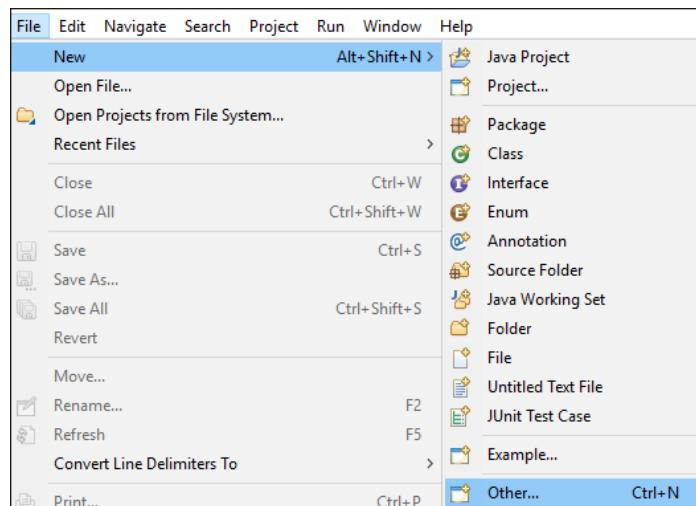


Abbildung 18.3 GUI erstellen, Teil 1

In dem Dialog wählen wir WINDOWBUILDER • SWING DESIGNER • APPLICATION WINDOW.

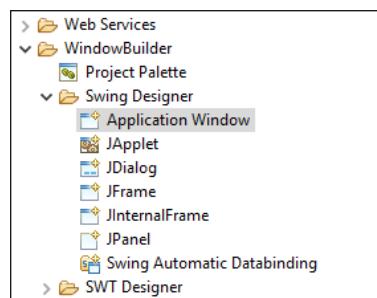


Abbildung 18.4 GUI erstellen, Teil 2

Nachdem wir auf NEXT gedrückt haben, erscheint ein Dialog, den wir vom Anlegen neuer Klassen kennen. Wir geben das Paket und einen Klassennamen an:

WindowBuilder erzeugt eine neue Klasse und öffnet standardmäßig den grafischen Editor. Wir können das Programm starten, aber ohne Funktion bleibt das langweilig. Zwischen GUI-Builder und Quellcode lässt sich jederzeit umschalten mit den kleinen Reitern SOURCE und DESIGN.

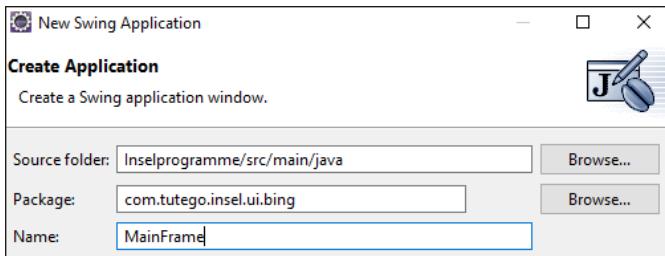


Abbildung 18.5 GUI erstellen, Teil 3

Sind wir in der Ansicht DESIGN, können wir per Drag & Drop Komponenten auf die Zeichenfläche setzen und hin und her schieben.

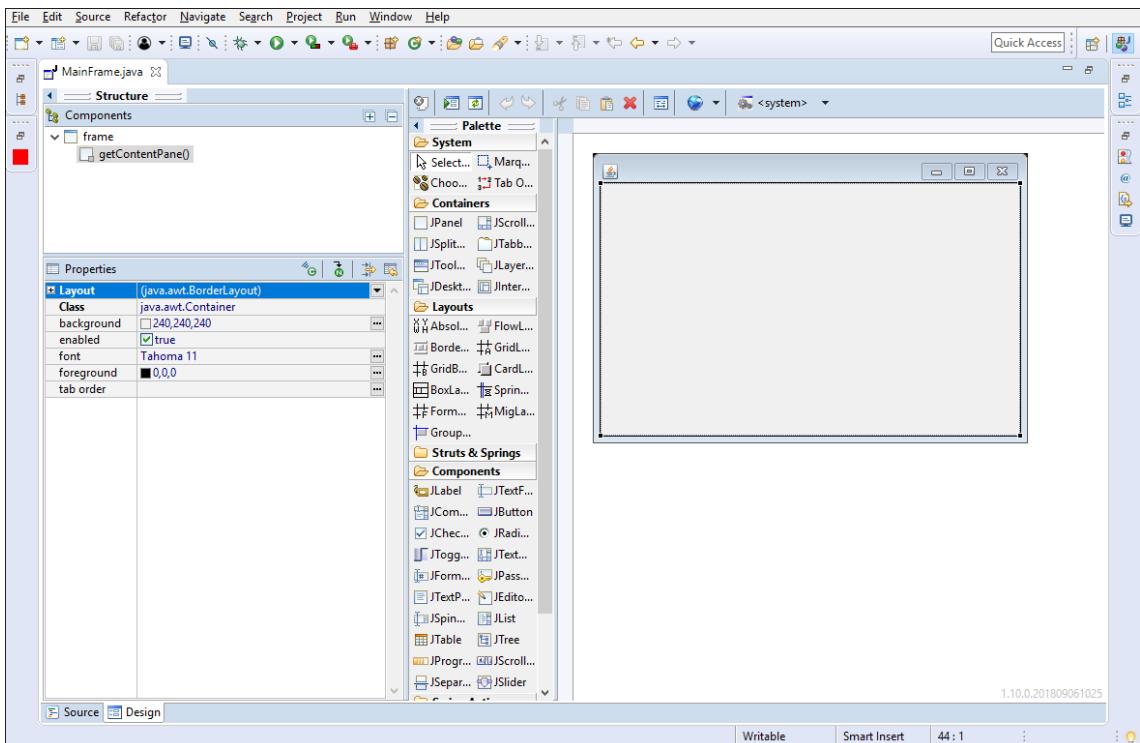
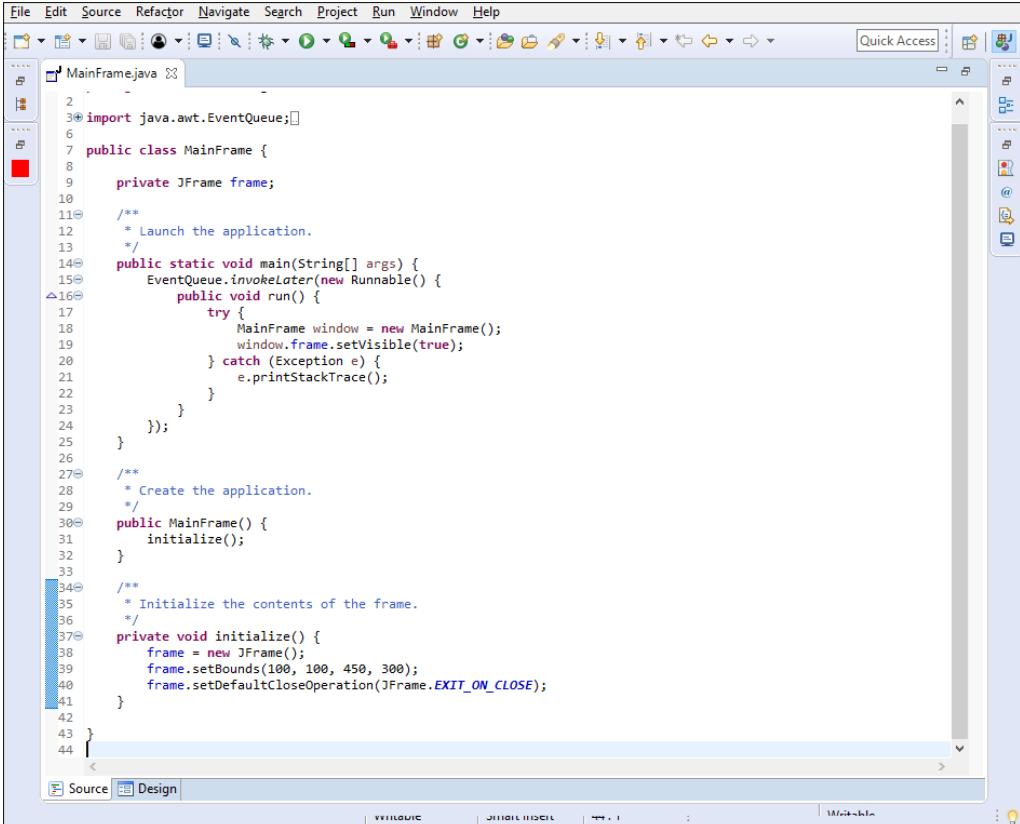


Abbildung 18.6 Layout

Mittig sind unter PALETTE die Komponenten aufgelistet, die wir per Drag & Drop auf den Formulardesigner ziehen können. Links bei PROPERTIES finden wir die Eigenschaften der Komponenten, etwa den Titel des Fensters.

Der WindowBuilder gießt das Layout direkt in Quellcode, den wir unter SOURCE einsehen können. In einem gewissen Rahmen können wir auch Code ändern, und das äußert sich in der Änderung der GUI.



```

File Edit Source Refactor Navigate Search Project Run Window Help
MainFrame.java
1
2
3 import java.awt.EventQueue;
4
5 public class MainFrame {
6
7     private JFrame frame;
8
9     /**
10      * Launch the application.
11     */
12     public static void main(String[] args) {
13         EventQueue.invokeLater(new Runnable() {
14             public void run() {
15                 try {
16                     MainFrame window = new MainFrame();
17                     window.frame.setVisible(true);
18                 } catch (Exception e) {
19                     e.printStackTrace();
20                 }
21             }
22         });
23     }
24 }
25
26 /**
27  * Create the application.
28 */
29 public MainFrame() {
30     initialize();
31 }
32
33 /**
34  * Initialize the contents of the frame.
35 */
36 private void initialize() {
37     frame = new JFrame();
38     frame.setBounds(100, 100, 450, 300);
39     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
40 }
41
42 }
43
44

```

The screenshot shows an IDE interface with the MainFrame.java file open in the central editor pane. The code implements a Java Swing application with a single window. It includes a main method that creates and displays the window using Java's Event Queue mechanism. The window is initialized with a title bar, a close button, and a blank white interior.

Abbildung 18.7 Source

### 18.4.3 Layoutprogramm starten

Im Quellcode lässt sich ablesen, dass die Klasse eine `main(String[])`-Methode hat, sodass wir MainFrame starten können. Mit **F11** springt dann ein unspektakuläres leeres Fenster auf.

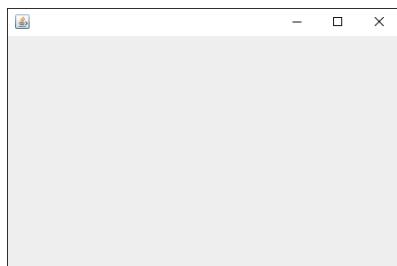


Abbildung 18.8 Leeres Fenster

Eine Vorschau gibt es auch. ÜBER DER PALETTE gibt es ein kleines Fenster mit grünem Play-Symbol, das über einen Klick einen ersten Eindruck vom Design vermittelt.

#### 18.4.4 Grafische Oberfläche aufbauen

Es ist an der Zeit, der Oberfläche ein paar Komponenten zu geben. Wenn wir in der Sicht Components den Zweig getContentPane() auswählen, können wir bei den Properties einen Layout-Manager wählen. Die Aufgabe eines Layout-Managers ist die Anordnung der Kinder. Hier gibt es diverse Techniken. Wir wählen GroupLayout, da es für GUI-Builder am leistungsfähigsten und am besten geeignet ist.

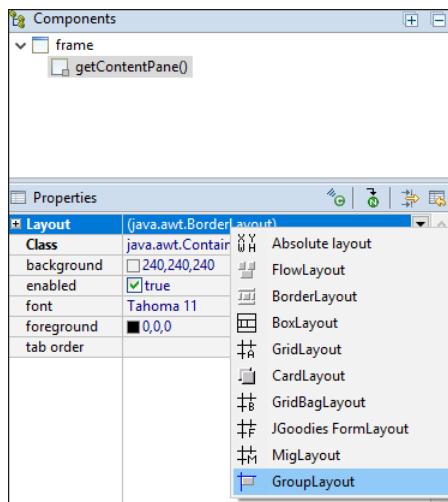


Abbildung 18.9 GroupLayout

In der Palette bei COMPONENTS suchen wir **JLABEL** und ziehen es per Drag & Drop auf die graue Designerfläche. Bemerkenswert ist, dass WindowBuilder vorgibt, was eine gute Position für die Beschriftung ist. Positionieren wir sie links oben, so rastet sie quasi ein.

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.

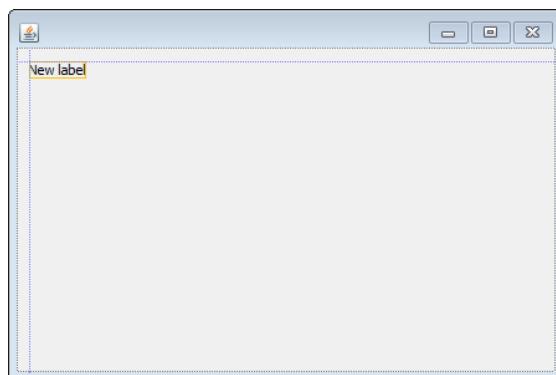


Abbildung 18.10 Erstes Label

Nachdem das Label positioniert ist, geben wir ihm einen Namen. Dazu kann links bei den PROPERTIES der Text verändert werden oder auch im Designer über einen Doppelklick auf den Text.

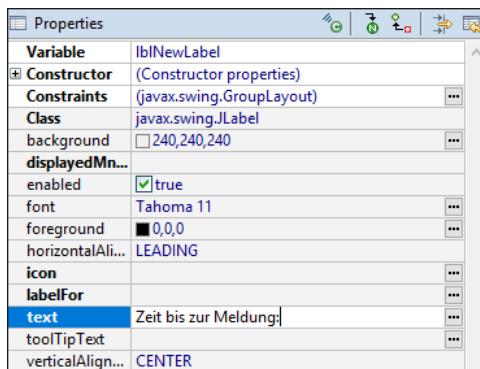


Abbildung 18.11 Label umbenennen

Jetzt, wo die erste Beschriftung steht, komplettieren wir das GUI. Unter der BESCHRIFTUNG setzen wir einen SLIDER (Schieberegler), allerdings nicht auf die ganze Breite, sondern etwa bis zur Hälfte. Unter den PROPERTIES auf der linken Seite gibt es Eigenschaften für MINIMUM (0) und MAXIMUM (100). Das MINIMUM 0 erhöhen wir auf 1 und das MAXIMUM auf 1.440 (die Minuten von 24 Stunden sollten reichen).

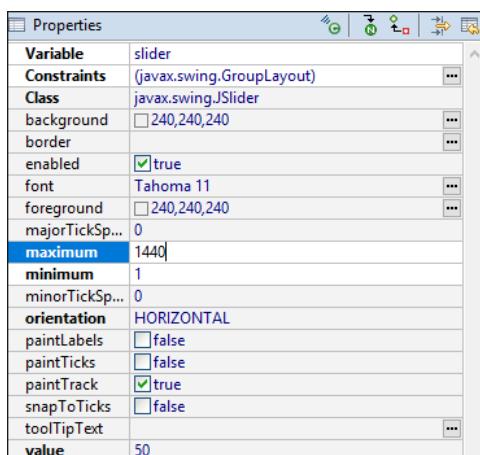


Abbildung 18.12 Properties vom JSlider

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. 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 die Vorschau sollte es nun so aussehen wie in Abbildung 18.13.



Abbildung 18.13 Oberfläche 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 eine Schaltfläche hinzu.

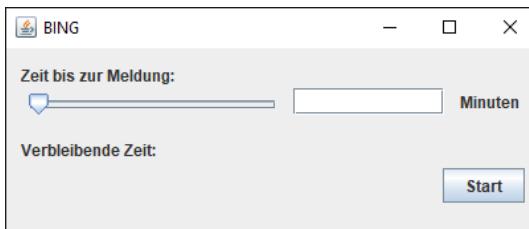


Abbildung 18.14 Vollständige Anwendung

Zwei Labels stehen nebeneinander (das zweite enthält nur ein Leerzeichen) unter dem SLIDER. Startet später die Anwendung, soll in der jetzt unsichtbaren Beschriftung die Restzeit stehen. Die Schaltflächen sind bei den SWING CONTROLS als JButton aufgeführt. Der Button dient zum Starten der Applikation. Über den Property-Editor geben wir gleichzeitig dem Fenster noch einen Titel (BING), und fertig ist die Oberfläche.

#### 18.4.5 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 Standard-Konstruktor: Er ruft `initialize()` auf. Eigene Funktionalität können wir hier hinzuschreiben.
- ▶ Die statische `main(String[])`-Methode könnte das Fenster gleich starten, denn sie baut ein Exemplar der eigenen Klasse auf, die ja Unterklasse von `JFrame` ist, und zeigt es mit `setVisible(true)` an.
- ▶ In der Methode selbst finden sich Variablen. Es sind keine Objektvariablen, sondern lokale Variablen. Das lässt sich jedoch individuell ändern.

Es lässt sich ablesen, dass für Schaltflächen die Klasse `JButton`, für Beschriftungen die Klasse `JLabel`, für den Schieberegler ein `JSlider` und für einfache Textfelder die Klasse `JTextField` zum Einsatz kommt.

### Variablen umbenennen

Die vom WindowBuilder automatisch generierten Namen machen semantisch nicht klar, worum es geht. Daher wollen wir die Variablennamen ändern, sodass klarer wird, was welche Komponenten macht. Hier kommt wieder die Sicht **COMPONENTS** ins Spiel. Sie zeigt die hierarchische Struktur der Komponenten an. Mit einem Klick neben den Punkt **Variable** in unseren Properties lässt sich jeder Variablenname ändern. Das wollen wir machen:

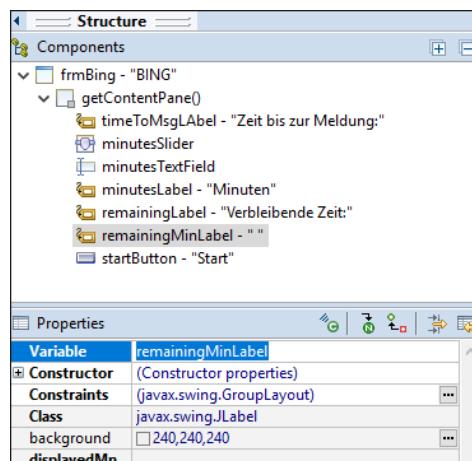


Abbildung 18.15 Umbenennung der Variablen in den Properties

In unserer `initialize()`-Methode wollen wir die Schiebereglerposition auf 1 setzen und das Textfeld ebenfalls mit 1 vorbelegen:

```
private void initialize() {
    JSlider minutesSlider = new JSlider();
    minutesSlider.setMaximum(1440);
    minutesSlider.setMinimum(1);
    minutesSlider.setValue( 1 );

    minutesTextField = new JTextField();
    minutesTextField.setColumns(10);
    minutesTextField.setText( "1" );
}
```

Die Methode `setValue(int)` erwartet einen numerischen Wert für den Schieberegler, und `setText(String)` erwartet einen String für das Textfeld.

### 18.4.6 Funktionalität geben

Nachdem die Variablen gut benannt sind, soll es an die Implementierung der Funktionalität gehen. Folgendes gilt es zu realisieren:

1. Das Bewegen des Sliders aktualisiert das Textfeld.
  2. Das Verändern des Textfeldes aktualisiert den SLIDER.
  3. Nach dem Start läuft das Programm, und die verbleibende Zeit wird aktualisiert.
- Ist die Zeit um, erscheint eine Dialogbox.

#### Wert vom Schieberegler und Textfeld synchronisieren

Als Erstes halten wir den Wert vom Schieberegler und den Wert im Textfeld synchron. Zurück in der DESIGN-Ansicht selektieren wir den SLIDER und finden im Kontextmenü den Menüpunkt ADD EVENT HANDLER. Das sind alle Ereignisse, die der Schieberegler auslösen kann. Wir interessieren uns für CHANGE:

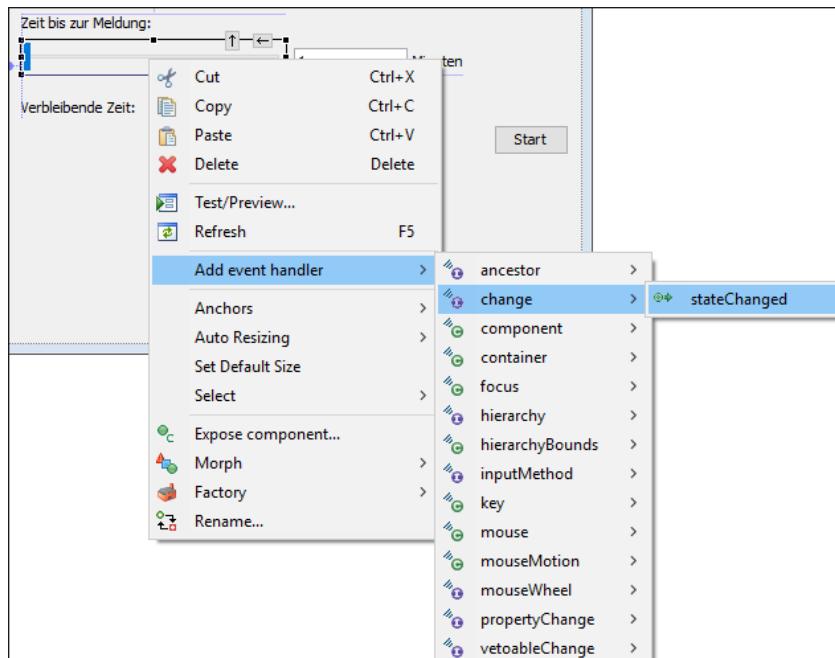


Abbildung 18.16 Im Formulardesigner einen Ereignisbehandler hinzufügen

Nach dem Aktivieren des Menüpunktes bekommen wir vom WindowBuilder Quellcode generiert (den generierten Quellcode müssen wir unter unser `minutesSlider.getValue(1);` verfrachten). Die Listener-Methode wird immer dann aufgerufen, wenn der Anwender den Schieberegler bewegt. Lesen wir einfach den aktuellen Wert aus, und schreiben wir ihn in das Textfeld:

```

minutesSlider.getValue();
minutesSlider.addChangeListener(new ChangeListener(){
    public void stateChanged(ChangeEvent e) {
        minutesTextField.setText( "" + minutesSlider.getValue() );
    }
});

```

Die `JSlider`-Methode `getValue()` liefert den aktuell eingestellten Wert als Ganzzahl, und `setText(String)` vom `JTextField` setzt einen String in die Textzeile.

Nach dem Start des Programms können wir den Schieberegler bewegen, und im Textfeld steht die ausgewählte Zahl. Der erste Schritt ist gemacht.

Kommen wir zum umgekehrten Fall: Wenn das Textfeld mit bestätigt wird, soll der Wert ausgelesen und damit die `JSlider`-Position gesetzt werden. Mit einem Rechtsklick gehen wir im Designer auf das Textfeld, wählen unter Add event handler dann action und klicken auf actionPerformed. Das fügt Programmcode für den Listener ein. Füllen wir ihn wie folgt:

```

public void actionPerformed(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. Falsche Wertebereiche ignorieren wir.



### Hinweis

Eine Rückkopplung zwischen Textfeld und Schieberegler kann in diesem Fall nicht auftreten. Bewegen wir den Schieberegler und setzen mit `setText(...)` den Text im Textfeld, so löst das *kein* ActionEvent aus.

Nachdem jetzt Änderungen im Textfeld und Schieberegler 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:

```

public void actionPerformed(ActionEvent e) {

    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();
}

```

Unser Programm ist fertig!

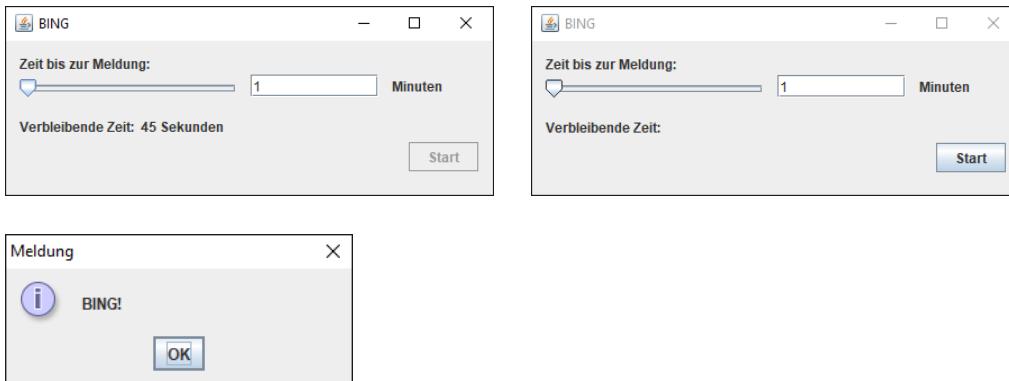


Abbildung 18.17 Die Anwendung in Aktion

## 18.5 Grundlegendes zum Zeichnen

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.

### 18.5.1 Die `paint(Graphics)`-Methode für das AWT-Frame

Als einleitendes Beispiel soll uns genügen, einen Text zu platzieren. Dafür überschreiben wir die Methode `paint(Graphics)` 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(Graphics)` aufrufen und das Zeichnen anstoßen:

**Listing 18.1** src/main/java/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 );
    }
}
```

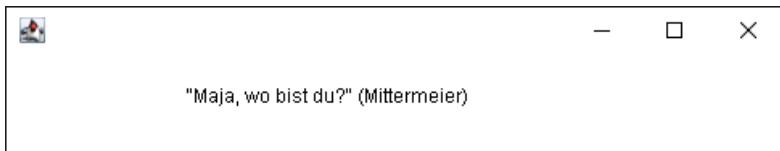


Abbildung 18.18 Ein Fenster mit gezeichnetem Inhalt

### Der Grafikkontext Graphics

Das Grafiksystem ruft von unserem Programm die `paint(Graphics)`-Methode auf und über gibt ein Objekt vom Typ `Graphics` – bzw. `Graphics2D`, wie wir später sehen werden. Dieser Gra fikkontext bietet verschiedene Methoden zum Setzen von Zeichenzuständen 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 Hinter grundgrafiken.

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 außer halb des Clipping-Bereichs werden nicht angezeigt. Daher wird ein Clipping-Bereich auch Beschnittbereich genannt.
- ▶ die aktuelle Schriftart (`java.awt.Font`) und die aktuelle Farbe (`java.awt.Color`)
- ▶ die Pixeloperation (XOR<sup>7</sup> oder Paint)
- ▶ die Translation – eine Verschiebung vom Nullpunkt

Wir können nur in der `paint(Graphics)`-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 Grafikkontex tes schreibt. Im Folgenden werden wir noch weitere Methoden kennenlernen.

#### Hinweis

Etwas ungewöhnlich ist die Tatsache, dass der Nullpunkt nicht oben links in den freien sicht baren 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 sicht baren Bereich fallen, wird ein `java.awt.Insets`-Objekt benötigt. Ist `f` ein `Frame`-Objekt, liefert `f.getInsets().top` die Höhe der Titelleiste.




---

<sup>7</sup> Zur Bewegung des Grafik-Cursors wird gern eine XOR-Operation eingesetzt. Obwohl dies absolut einfach erscheint, ist die Realisierungsidee in den USA patentiert.

### 18.5.2 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 Beispiel<sup>8</sup> aus der C128-Zeit:

```

10 COLOR 0,1 :REM SELECT BACKGROUND COLOR
20 COLOR 1,3 :REM SELECT FOREGROUND COLOR
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 Grafiksystemen) nicht mehr. Auch mit Objektorientierung hat sie nicht viel zu tun!

In Java führt ein Repaint-Ereignis zum Aufruf der `paint(Graphics)`-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(Graphics)`-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(Graphics)`-Methode geschehen bzw. in Methoden, die von `paint(Graphics)` aufgerufen werden. Wenn wir aber selbst etwas zeichnen wollen, wie lässt sich `paint(Graphics)` dann parametrisieren?

---

<sup>8</sup> Commodore 128 System Guide, Commodore Business Machines, Inc. 1985, online zugänglich unter <http://rvbelzen.tripod.com/c128sg/toc.htm>

Um mit diesem Problem umzugehen, müssen wir der `paint(Graphics)`-Methode Informationen mitgeben. Diese Informationen kann die Methode aus den Objektattributen beziehen. Daher implementieren wir eine Unterklasse einer Komponente, die eine `paint(Graphics)`-Methode besitzt. Anschließend können wir Objektzustände ändern, sodass `paint(Graphics)` neue Werte bekommt und somit gewünschte Inhalte zeichnen kann.

### 18.5.3 Zeichnen von Inhalten auf einen JFrame

Wenn Swing eine Komponente zeichnet, ruft es automatisch die Methode `paint(Graphics)` auf. Um eine Grafik selbst in ein Fenster zu zeichnen, ließe sich von `JFrame` eine Unterklasse bilden und `paint(Graphics)` 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(Graphics)`-Methode auf. Allerdings überschreiben eigene Unterklassen von Swing-Komponenten im Regelfall nicht `paint(Graphics)`, sondern `paintComponent(Graphics)`. Das liegt daran, dass Swing in `paint(Graphics)` zum Beispiel noch Rahmen zeichnet und sich um eine Pufferung des Bildschirm Inhalts zur Optimierung kümmert. So ruft `paint(Graphics)` die drei Methoden `paintComponent(Graphics)`, `paintBorder(Graphics)` und `paintChildren(Graphics)` 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(Graphics)`:

**Listing 18.2** src/main/java/com/tutego/insel/ui/graphics/DrawFirstLine.java, Ausschnitt 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(Graphics)` 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 18.3** src/main/java/com/tutego/insel/ui/graphics/DrawFirstLine.java, Ausschnitt 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.



#### Tipp

Für einfache Tests lässt sich auch `JOptionPane.showMessageDialog(null, new DrawPanel());` einsetzen.

#### 18.5.4 Auffordern zum Neuzeichnen mit `repaint(...)`

Die Methode `repaint(...)` kann von außen aufgerufen werden, um ein Neuzeichnen zu erzwingen. Wenn die Komponente wie unsere Unterklasse von `JPanel` eine Swing-Komponente ist, dann wird die `paint(Graphics)`-Methode der Komponente aufgerufen. Im Fall einer AWT-Komponente, wie `Frame`, wird `update(Graphics)` aufgerufen, das ja automatisch `paint(Graphics)` aufruft.

```
abstract class java.awt.Component
implements ImageObserver, MenuContainer, Serializable
```

- `void repaint()`  
Erbittet sofortiges Neuzeichnen der Komponente. Entspricht `repaint(0, 0, 0, width, height)`.
- `void repaint(long tm)`  
Erbittet Neuzeichnen in höchstens `tm` Millisekunden. Entspricht `repaint(tm, 0, 0, width, height)`.
- `void repaint(int x, int y, int width, int height)`  
Erbittet Neuzeichnen der Komponente im angegebenen Bereich.
- `void repaint(long tm, int x, int y, int width, int height)`  
Erbittet Neuzeichnen der Komponente nach höchstens `tm` Millisekunden im angegebenen Bereich. Die Zeitbeschränkung wird in Swing nicht wirklich verwendet.

### 18.5.5 Java 2D-API

Seit dem JDK 1.2 – und das ist nun schon etwas her – hat sich beim Zeichnen einiges getan. So wird der `paint(Graphics)`-Methode – und `paintComponent(Graphics)` ebenso – nicht mehr nur ein `Graphics`-Objekt übergeben, sondern eine Unterklasse von `Graphics`, `Graphics2D`. Die Klasse wurde im Rahmen der Java 2D-API aus den *Java Foundation Classes (JFC)* eingeführt und bietet erweiterte Zeichenmöglichkeiten, die mit der Sprache PostScript vergleichbar sind. Als wichtige Ergänzung sind Transformationen auf beliebig geformten Objekten sowie Füllmustern und Kompositionen zu nennen. Die gezeichneten Objekte sind optional weichgezeichnet.

Da die Java-Entwickler die Signatur der `paintXXX(Graphics)`-Methoden nicht ändern wollten, blieb `Graphics` als Parametertyp stehen, und wir müssen es, um die erweiterte Funktionalität nutzen zu können, im Rumpf auf `Graphics2D` anpassen:

```
@Override protected void paintComponent( Graphics g ) {
    Graphics2D g2 = (Graphics2D) g;
    ...
}
```

Obwohl `Graphics2D` selbst im `java.awt`-Paket untergebracht ist, befinden sich viele der 2D-Klassen im Paket `java.awt.geom`.

#### Hinweis

Das Grafiksystem übergibt uns in der `paintXXX(Graphics)`-Methode zwar immer ein Objekt vom Typ `Graphics2D`, aber wir werden in den Beispielprogrammen nur dann eine Typumwandlung vornehmen, wenn wir wirklich die Erweiterungen von `Graphics2D` nutzen.



## 18.6 Zum Weiterlesen

Der Band »Java SE 11 Standard-Bibliothek« bietet drei Extrakapitel zu grafischer Programmierung, die Swing für grafische Oberflächen, das AWT und Java 2D für die Zeichenfunktionalität sowie JavaFX vorstellen.

# Kapitel 19

## 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*<sup>1</sup>) 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.

### 19.1 Alte und neue Welt in `java.io` und `java.nio`

Java bringt schon seit Urzeiten Klassen und Methoden zur Dateiverarbeitung mit, und das `java.io`-Paket gibt es schon seit Java 1.0. In späteren Java-Versionen kamen neue Typen und neue Pakete hinzu.

#### 19.1.1 `java.io`-Paket mit `File`-Klasse

Das `java.io`-Paket deklariert eine zentrale Klasse `File`, die für eine Datei oder ein Verzeichnis auf dem aktuellen Dateisystem steht. `File` bietet zahlreiche dateiorientierte Operationen, wie Dateien anlegen, löschen, umbenennen, Verzeichnisse auflisten usw.

#### File-Probleme

Bei `File` konzentriert sich alles auf eine Klasse, und sie kann alltägliche Probleme nicht wirklich lösen:

---

<sup>1</sup> Das englische Wort »file« geht auf das lateinische Wort »filum« zurück. Dies bezeichnete früher eine auf Draht aufgereihte Sammlung von Schriftstücken.

- ▶ 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 uns ein Callback informiert, sobald sich eine Datei verändert hat?
- ▶ Wie lässt sich einfache Verzeichnis rekursiv ablaufen?
- ▶ Wie lässt sich eine symbolische Verknüpfung anlegen und verfolgen?
- ▶ Die `File`-Klasse wurde immer mehr zum Sammelbecken für alle möglichen Anfragemethoden wie Lesbarkeit, Änderungsdatum usw. Ein Problem ist dabei, dass gewisse Dinge nicht wirklich auf jedem System identisch sind – etwa die Dateirechte.
- ▶ Wie lässt sich realisieren, dass die `File`-Operationen abstrahiert werden und nicht nur auf dem lokalen Dateisystem basieren? Wünschenswert ist eine Abstraktion, sodass die gleiche API auch ein virtuelles Dateisystem im Hauptspeicher, entfernte Dateisysteme wie FTP oder ein Repository anspricht.

### 19.1.2 NIO.2 und `java.nio`-Paket

Alternativ zum Paket `java.io` mit der Klasse `File` gibt es das Paket `java.nio.file` mit weiteren Typen, wie `Path`, `Paths`, `Files`, `FileSystem`.

#### Geschichte

Die `File`-Probleme waren lange bekannt, und so begannen 2003 die Arbeiten an dem JSR 203, »More New I/O APIs for the Java™ Platform (›NIO.2‹)«. Doch erst Mitte 2011, in Java 7, kam es zum großen Wurf.

### 19.1.3 `java.io.File` oder `java.nio.*`?

Die `File`-Klasse ist heutzutage an den wenigsten Stellen nötig, und zwar eigentlich nur aus zwei Gründen:

- ▶ Alte APIs erwarten als Parameter `File`-Objekte. Dafür gibt es Brücken zwischen den APIs, doch ersetzen die neuen NIO.2-Typen die `File`-Klasse im Grunde vollständig, und langfristig sollten `File`-Objekt nicht mehr erwartet werden.
- ▶ `File`-Objekte sind nötig, wenn wirklich Dateien auf dem eigenen Datensystem repräsentiert werden sollen und nicht unabhängig vom Dateisystem; ein gutes Beispiel ist Desktop mit der `open(File)`-Methode.

## 19.2 Dateisysteme und Pfade

Im Zentrum von NIO.2 stehen die Typen `FileSystem` und `Path`:

- ▶ `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 die alte Klasse `File`, doch der große Unterschied ist, dass `File` selbst die Datei oder das Verzeichnis repräsentiert und Abfragemethoden wie `isDirectory()` oder `lastModified()` deklariert, während `Path` nur den Pfad repräsentiert und nur pfadbezogene Methoden anbietet. Modifikationsmethoden gehören nicht dazu; dazu dienen Extra-Typen wie `BasicFileAttributes` für Attribute.

### 19.2.1 FileSystem und Path

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/" );
```

Mit einer Abkürzung:

```
Path p = Paths.get( "C:/Windows/Fonts/" );
```

[zB]

Da der Ausdruck `FileSystems.getDefault().getPath(...)` etwas unhandlich ist, gibt es drei kürzere Wege:

- ▶ Es gibt die Utility-Klasse `Paths`, und ein Aufruf von `Paths.get(...)` liefert den `Path`.
- ▶ Seit Java 11 gibt es in `Path` zwei statische `of(...)`-Methoden.
- ▶ Auch aus einem `File`-Objekt lässt sich mit `toPath()` ein `Path` ableiten.

Wir werden die Vereinfachung mit `Paths.get(...)` im Folgenden nutzen. Die `Path.of(...)`-Variante ist noch etwas neu ...

```
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 einem URI.

```
final class java.nio.file.Path
```

- static Path of(String first, String... more) Seit Java 11
- static Path of(URI uri) Seit Java 11

Jedes Path-Objekt hat auch eine Methode getFileSystem(), mit der wir wieder an das FileSystem kommen.

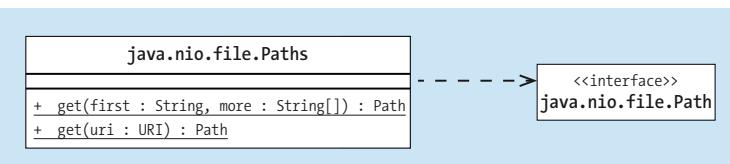


Abbildung 19.1 Abhängigkeiten der Typen Paths und Path

### Hinweis

Der Pfad-String darf unter Java kein \u0000 enthalten, andernfalls gibt es eine Ausnahme. So führt Paths.get("my.php\u0000.jpg") zur Ausnahme »java.nio.file.InvalidPathException: Illegal char <\u0000> at index 6: my.php«. Das ist ein wichtiges Sicherheitsmerkmal, um zum Beispiel einen Webserver davor zu schützen, falsche Dateien anzunehmen. Der Dateiname sieht über "my.php\u0000.jpg".endsWith(".jpg") wie eine JPG-Datei aus, aber würde alles nach dem Null-String abgeschnitten (was einige Dateisysteme machen), wäre plötzlich eine Datei *my.php* angelegt.

### Path-Eigenschaften erfragen

Der Path-Typ deklariert diverse getXXX(...)-Methoden (und eine isAbsolute()-Methode), die eine gewisse Ähnlichkeit mit den Methoden aus File haben. Ein paar Beispiele zur Anwendung:

#### Listing 19.1 src/main/java/com/tutego/insel/nio2/FileSystemPathFileDemo1.java, main()

```
Path p = Paths.get( "C:/Windows/Fonts/" );
System.out.println( p.toString() ); // C:\Windows\Fonts
```

```

System.out.println( p.isAbsolute() );           // true
System.out.println( p.getRoot() );             // C:\ 
System.out.println( p.getParent() );           // Fonts
System.out.println( p.getNameCount() );         // 2
System.out.println( p.getName(p.getNameCount()-1) ); // Fonts

```

Methoden wie getPath(), getRoot() und getParent() liefern alle wiederum Path-Objekte aus den Bestandteilen eines gegebenen Pfades. Es gibt drei Methoden, die das Ergebnis nicht als Path weiterverarbeiten:

- ▶ `toString()` liefert eine String-Repräsentation,
- ▶ die Methode `toUri()` einen URI und
- ▶ `toFile()` ein traditionelles File-Objekt.

Dadurch, dass Path eine hierarchische Liste von Namen für den Pfad speichert, lässt sich jedes Segment des Pfades erfragen; das ist die Aufgabe von `getName(int n)`, das wiederum einen Path liefert. Die Methode `subpath(int beginIndex, int endIndex)` liefert einen Path mit den Segmenten des angegebenen Bereichs. Path implementiert die Iterable-Schnittstelle, was eine Methode `iterator()` vorschreibt – das wiederum bedeutet, dass Path rechts vom Doppelpunkt im erweiterten for auftauchen kann.

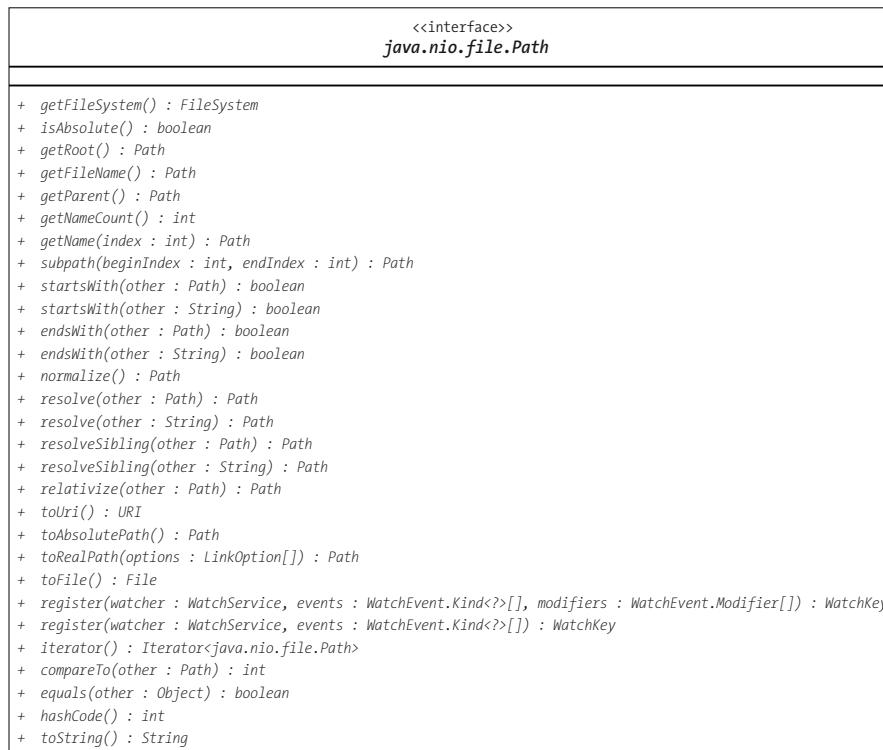


Abbildung 19.2 Klassendiagramm von Path

Praktisch sind die Prüfmethoden `startsWith(Path other)` und `endsWith(Path other)`, die feststellen, ob der Pfad mit einem bestimmten anderen Pfad beginnt oder endet. Aus `Object` wird `equals(...)` überschrieben. Da `Path` die Schnittstelle `Comparable<Path>` realisiert, wird zudem `compareTo(Path)` implementiert. Die Methode `equals(...)` löst die Pfade nicht auf, sondern betrachtet nur den Namen; die statische Methode `isSameFile(Path, Path)` der Klasse `Files` macht diesen Test und löst relative Bezüge auf. Neben `equals(...)` überschreibt `Path` auch `hashCode()`.

```
interface java.nio.file.Path
extends Comparable<Path>, Iterable<Path>, Watchable
```

- `String toString()`
- `File toFile()`
- `URI toUri()`
- `Path getFileName()`
- `Path getParent()`
- `Path getRoot()`
- `boolean isAbsolute()`
- `int getNameCount()`
- `Path getName(int index)`
- `Iterator<Path> iterator()`
- `Path subpath(int beginIndex, int endIndex)`
- `boolean endsWith(Path other)`
- `boolean endsWith(String other)`
- `boolean startsWith(Path other)`
- `boolean startsWith(String other)`
- `boolean equals(Object other)`
- `int compareTo(Path other)`
- `int hashCode()`

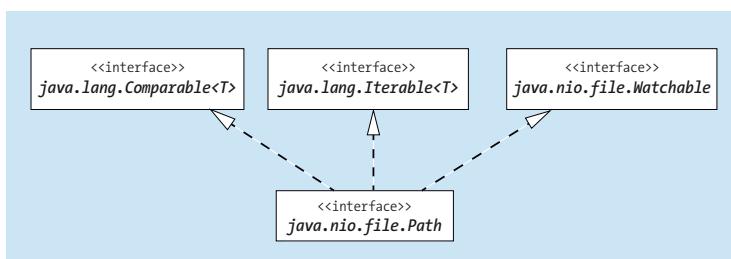


Abbildung 19.3 Vererbungsbeziehung von Path

### Hinweis

Die Methode `getFileName()` liefert keinen String, sondern ein Path-Objekt nur mit dem Dateinamen, bei dem also `getNameCount() == 1` ist. Daher führt `path.getFileName().endsWith(".xml")` zum Testen, ob ein Dateiname wie `trainings.xml` auf »`.xml`« endet, nicht zum Ziel, denn `endsWith(...)` testet, ob das letzte Segment im Pfad – in diesem Fall der komplette Dateiname, nur ohne Ordner – exakt diesen Namen trägt. Als Lösung ist zum Beispiel `path.getFileName().toString().endsWith(".xml")` gültig.

### Hinweis

Unterschiedliche Dateisysteme haben ihre ganz eigenen Beschränkungen und Schreibweisen. Mal ist das Ordner trennzeichen ein »/«, mal ein »\«. Eine Wikipedia-Seite<sup>2</sup> gibt einen Überblick, wie unterschiedliche Betriebssysteme Separatoren oder das Wurzelverzeichnis definieren. Java akzeptiert unter Windows auch den Slash als Ordner trenner.

## Neue Pfade aufbauen

Die `resolveXXX(...)`-Methoden bauen aus gegebenen Pfaden neue Pfade zusammen; die Methoden akzeptieren die Parametertypen `String` und `Path`.

### Beispiel

Hänge das Benutzerverzeichnis mit dem Bilderverzeichnis zusammen:

```
Path picturePath = Paths.get( System.getProperty("user.home") )
    .resolve( "Pictures" )
    .resolve( "Cora" );
System.out.println( picturePath ); // z. B. C:\Users\Chris\Pictures\Cora
```

Die Unterverzeichnisse lassen sich sonst auch direkt in `get(...)` eintragen.

Es ist im gleichen Ordner wie `stormyDaniels.jpg` ein neuer Pfad für die Datei `pee_tape.tar` gefragt:

```
Path stormy = Paths.get( "d:/geheimakten/usa/stormyDaniels.jpg" );
Path peepee = stormy.resolveSibling( "pee_tape.tar" );
System.out.println( peepee ); // d:\geheimakten\usa\pee_tape.tar
```

Eine interessante Methode ist auch `relativize(Path)` – sie liefert aus einer Basisangabe einen relativen Pfad, der zu einem anderen Pfad führt.

<sup>2</sup> [https://en.wikipedia.org/wiki/Path\\_\(computing\)#Representations\\_of\\_paths\\_by\\_operating\\_system\\_and\\_shell](https://en.wikipedia.org/wiki/Path_(computing)#Representations_of_paths_by_operating_system_and_shell)



### Beispiel

Von *C:/Windows/Fonts* nach *C:/Windows/Cursors* führt der relative Pfad ..\Cursors:

```
System.out.println( Paths.get( "C:/Windows/Fonts" )
                    .relativize( Paths.get("C:/Windows/Cursors") ) )
); // ..\Cursors
```

```
interface java.nio.file.Path
extends Comparable<Path>, Iterable<Path>, Watchable
```

- Path relativize(Path other)
- Path resolve(Path other)
- Path resolve(String other)
- Path resolveSibling(Path other)
- Path resolveSibling(String other)

### Normalisierung und Pfadauflösung

Genauso wie die `File`-Klasse symbolisiert die `Path`-Klasse einen Pfad, aber dieser muss nicht auf eine konkrete Datei oder ein konkretes Verzeichnis zeigen. Daher liefern die vorgestellten Methoden lediglich Informationen, die sich aus dem vorgegebenen Namen erschließen lassen, ohne auf das Dateisystem zurückzugreifen. Bei relativen Pfaden liefern die Anfrage-methoden daher wenig Spannendes:

**Listing 19.2** src/main/java/com/tutego/insel/nio2/FileSystemPathFileDemo2.java, main()

```
Path p = Paths.get( "../.." );
System.out.println( p.toString() );           // ..
System.out.println( p.isAbsolute() );          // false
System.out.println( p.getRoot() );             // null
System.out.println( p.getParent() );           // ..
System.out.println( p.getNameCount() );         // 2
System.out.println( p.getName(p.getNameCount()-1) ); // ..
```

Um ein wenig Ordnung in relative Pfadangaben zu bringen, bietet die `Path`-Klasse die Methode `normalize()`, die ohne Zugriff auf das Dateisystem die Bezüge »..« und »...« entfernt.

Zum Auflösen der relativen Adressierung mit Zugriff auf das Dateisystem bietet die `Path`-Klasse die beiden Methoden `toAbsolutePath()` bzw. `toRealPath(...)` an.

**Listing 19.3** src/main/java/com/tutego/insel/nio2/RealAndAbsolutePath.java, main()

```
Path p2 = Paths.get( "../.." );
System.out.println( p2.toAbsolutePath() );
```

```
// C:\Users\Christian\Documents\Insel\programme\2_06_Files\...\...
try {
    System.out.println( p2.toRealPath( LinkOption.NOFOLLOW_LINKS ) );
    // C:\Users\Christian\Documents\Insel
}
catch ( IOException e ) { e.printStackTrace(); }
```

Die erste Methode, `toAbsolutePath()`, normalisiert nicht, sondern löst einfach nur den relativen Pfad in einen absoluten Pfad auf. Die Auflösung vom `..../` erledigt `toRealPath(LinkOption...)`, wobei das (optionale) Argument ausdrückt, ob Verknüpfungen verfolgt werden sollen oder nicht.

### Hinweis

Die Methode `toRealPath(...)` löst eine Ausnahme aus, wenn eine Auflösung eines Pfades zu einer Datei versucht wird, die nicht existiert. So führt zum Beispiel `Paths.get("../0x").toRealPath()` zur »java.nio.file.NoSuchFileException: C:\Users\Chris\0x«, egal ob mit `LinkOption.NOFOLLOW_LINKS` oder ohne. `Paths.get("../0x").toAbsolutePath()` führt zu keinem Fehler.



```
interface java.nio.file.Path
extends Comparable<Path>, Iterable<Path>, Watchable
```

- `Path normalize()`
- `Path toAbsolutePath()`
- `Path toRealPath(LinkOption... options)`

Es gibt noch zwei `register(...)`-Methoden in `Path`, doch die haben etwas mit dem Anmelden eines Horchers zu tun, der auf Änderungen im Dateisystem reagiert. Sie werden später vor gestellt.

## 19.2.2 Die Utility-Klasse Files

Da die Klasse `Path` nur Pfade, aber keine Dateiinformationen wie die Länge oder Änderungszeit repräsentiert, und da `Path` auch keine Möglichkeit bietet, Dateien anzulegen und zu löschen, übernimmt die Klasse `Files` diese Aufgaben.

Eine einfache Methode ist `size(...)`, die die Länge der Datei liefert. Anders als bei `java.io.File` führen nahezu alle `Files`-Methoden zu einer `IOException`, wenn es Probleme bei den Ein-/Ausgabe-Operationen gibt.

```
final class java.nio.file.Files
```

- static long size(Path path) throws IOException  
Liefert die Größe der Datei.

### Einfaches Einlesen und Schreiben von Dateien

Mit den Methoden `readAllBytes(...)`, `readAllLines(...)`, `readString(...)`, `lines(...)` und `write(...)` und `writeString(..)` kann `Files` einfach einen Dateinhalt einlesen oder Strings bzw. ein Byte-Feld schreiben.

**Listing 19.4** src/main/java/com/tutego/insel/nio2/ListAllLines.java, main()

```
URI uri = ListAllLines.class.getResource( "/lyrics.txt" ).toURI();
Path p = Paths.get( uri );
System.out.printf( "Datei '%s' mit Länge %d Byte(s) hat folgende Zeilen:%n",
    p.getFileName(), Files.size( p ) );
int lineCnt = 1;
for ( String line : Files.readAllLines( p ) )
    System.out.println( lineCnt++ + ": " + line );
```

```
final class java.nio.file.Files
```

- static byte[] readAllBytes(Path path) throws IOException  
Liest die Datei komplett in ein Byte-Feld ein.
- static List<String> readAllLines(Path path) throws IOException
- static List<String> readAllLines(Path path, Charset cs) throws IOException  
Liest die Datei Zeile für Zeile ein und liefert eine Liste dieser Zeilen. Optional ist die Angabe einer Kodierung, standardmäßig ist es StandardCharsets.UTF\_8.
- static String readString(Path path) throws IOException
- static String readString(Path path, Charset cs) throws IOException  
Liest eine Datei komplett aus und liefert den Inhalt als String. Ohne Angabe der Kodierung gilt standardmäßig UTF-8. Beide Methoden neu in Java 11.
- static Path write(Path path, byte[] bytes, OpenOption... options) throws IOException  
Schreibt ein Byte-Array in eine Datei.
- static Path write(Path path, Iterable<? extends CharSequence> lines, OpenOption... options) throws IOException
- static Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options) throws IOException  
Schreibt alle Zeilen aus dem Iterable in eine Datei. Optional ist die Kodierung, die StandardCharsets.UTF\_8 ist, so nicht anders angegeben.

- static Path writeString(Path path, CharSequence csq, OpenOption... options) throws IOException
  - static Path writeString(Path path, CharSequence csq, Charset cs, OpenOption... options) throws IOException
- Schreibt eine Zeichenfolge in die genannte Datei. Der übergebene path wird zurückgegeben. Ohne Angabe der Kodierung gilt standardmäßig UTF-8. Beide Methoden neu in Java 11.

Die Aufzählung OpenOption ist ein Vararg, und daher sind Argumente nicht zwingend nötig. StandardOpenOption ist eine Aufzählung vom Typ OpenOption mit Konstanten wie APPEND, CREATE usw.

### Beispiel

[zB]

Lies eine UTF-8-kodierte Datei ein:

```
String s = Files.readString( path );
```

Bevor die praktische Methode in Java 11 einzog, sah eine Alternative so aus:

```
String s = new String( Files.readAllBytes( path ), StandardCharsets.UTF_8 );
```

### Hinweis

[<<]

Auch wenn es naheliegt, die Files-Methode zum Einlesen mit einem Path-Objekt zu füttern, das einen 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«.

```
URI uri = new URI( "http://tutego.de/javabuch/aufgaben/bond.txt" );
Path path = Paths.get( uri );      // ☠
List<String> content = Files.readAllLines( path );
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.

### Strom von Zeilen

Die bisherigen Files-Methoden lesen und schreiben als Ganzes, was ein Speicherproblem werden kann. Soll das Einlesen zeilenweise erfolgen, so ist statt readAllLines(), das im Speicher alle Zeilen als String-Objekte vorhält, ein Stream<String> eine gute Alternative. Zwei Methoden liefern einen Strom von Zeilen:

```
final class java.nio.file.Files
```

- static Stream<String> lines(Path path)

- Stream<String> lines(Path path, Charset cs)

Liefert einen Stream von Zeilen einer Datei. Optional ist die Angabe der Kodierung, die sonst standardmäßig StandardCharsets.UTF\_8 ist.

### Datenströme kopieren

Sollen die Daten nicht direkt aus einer Datei in ein Byte-Array/eine String-Liste gehen bzw. aus einem Byte-Array/einer String-Sammlung in eine Datei, sondern von einer Datei in einen Datenstrom, so bieten sich zwei `copy(...)`-Methoden an:

```
final class java.nio.file.Files
```

- static long copy(InputStream in, Path target, CopyOption... options)

Entleert den Eingabestrom und kopiert die Daten in die Datei. Die Anzahl kopierter Bytes ist die Rückgabe.

- static long copy(Path source, OutputStream out)

Kopiert alle Daten aus der Datei in den Ausgabestrom. Die Anzahl kopierter Bytes ist die Rückgabe.

Im Zusammenhang mit Datenströmen kommen wir noch einmal auf diese beiden Methoden zurück.

## 19.3 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`.

### 19.3.1 Ein RandomAccessFile zum Lesen und Schreiben öffnen

Die Klasse deklariert zwei Konstruktoren, die mit einem Dateinamen oder `File`-Objekt ein `RandomAccessFile`-Objekt anlegen. 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 19.1 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 FileNotFoundException aus, falls die Datei nicht geöffnet werden kann.
- void close()
- Schließt eine geöffnete Datei wieder.

### 19.3.2 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 Array `b` ab der Position `off` im Array.  
Die gelesene Größe wird immer zurückgegeben, auch wenn weniger als `len` Bytes gelesen wurden.
- `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` bzw. 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, »Die UTF-8-Kodierung«, 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 Lesemethoden 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.

### Die UTF-8-Kodierung \*

`writeUTF(String)` und `readUTF()` sind zwei Operationen, die die Schnittstellen `DataOutput` und `DataInput` vorschreiben. Neben `RandomAccessFile` implementiert `DataOutputStream` die Schnittstelle `DataOutput` und `DataInputStream` die Schnittstelle `DataInput`.

Java verwaltet Unicode-Zeichen über den Datentyp `char`, der (immer noch<sup>3</sup>) 16 Bit lang ist. In unseren Breiten stammen die meisten Zeichen aus den herkömmlichen 8 Bit des Latin-1-Zeichensatzes. Würden die Zeichen als Unicode (also 2 Byte) versendet, bestände der 16-Bit-Datenstrom im Wesentlichen zur Hälfte aus Nullen. Aus diesem Grund gibt es eine alternative Kodierung, die jedes 16-Bit-Unicode-Zeichen platzsparend schreibt und in Abhängigkeit von der Belegung 1, 2 oder 3 Byte lang ist. Die Kodierung der Zeichen richtet sich nach der Belegung der Bits wie folgt:

- `'\u0001' bis '\u007F'`: Die Zeichen werden direkt mit einem Byte geschrieben. Die westlichen Texte, die zum Großteil in 7-Bit-ASCII verfasst sind, lassen sich somit kompakt schreiben.
- `'\u0080' bis '\u07FF'`: Die Zeichen werden mit 2 Byte kodiert.
- `'\u0800' bis '\uFFFF'`: Die Zeichen werden mit 3 Byte kodiert.

Die Kodierung, die Java wählt, ist an UTF-8 angelehnt und wird im Folgenden einfach *UTF-8-Kodierung* genannt. Das modifizierte UTF-8-Format<sup>4</sup> von Java kodiert etwa – anders als es der Unicode-Standard im Kapitel »Unicode Encoding Forms« beschreibt – das Zeichen `'\u0000'` in 2 Byte, das laut Unicode-Standard in einem Byte geschrieben würde.

---

<sup>3</sup> Eine Anspielung, da Java seit Version 5 Unicode 4 mit 32-Bit-Zeichen unterstützt und wir für die Umsetzung erheblich tricksen müssen.

<sup>4</sup> <http://tutego.de/go/modifedutf8>



### Hinweis

Würden die Zeichenfolgen lediglich mit der vorgestellten Kodierung geschrieben, wüsste der Zeichenleser nicht, wann das Ende der Zeichenfolge erreicht ist. Daher beginnt writeUTF(String) mit einer Längenkennung. Zum Lesen von Zeilen ist readUTF() von Vorteil, denn readLine() führt keine korrekte Unicode-Konvertierung durch.

### 19.3.3 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 Lese-methoden:

- ▶ 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.

### 19.3.4 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.

### 19.3.5 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.

#### Beispiel

[zB]

Erzeuge eine Datei, und setze an die Stelle 1.000 das Byte 0xFF:

**Listing 19.5** src/main/java/com/tutego/insel/io/CreateBigFile.java, main()

```
try ( RandomAccessFile file = new RandomAccessFile("C:/test.bin", "rw" ) ) {
    file.seek( 999 );
    file.write( 0xFF );
}
```

Da `skipBytes(int)` den Dateizeiger nicht »hinter« die Datei stellen kann, funktioniert die Lösung nur mit `seek(long)`.

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! Es ist schon interessant, dass seek(long) mit long parametrisiert wird, skipBytes(int) aber nur mit int.

Setzt seek(long) den Zeiger weiter, als es möglich ist, wird die Datei dadurch nicht automatisch größer. Sie verändert jedoch ihre Größe, wenn Daten geschrieben werden.

## 19.4 Basisklassen für die Ein-/Ausgabe

Unterschiedliche Klassen zum Lesen und Schreiben von Binär- und Zeichendaten sammeln sich im Paket java.io. Für die byteorientierte Verarbeitung, etwa von PDF- oder MP3-Dateien, gibt es andere Klassen als für Textdokumente, zum Beispiel HTML-Dokumente 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 Stromklassen 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 aus/in eine/r bestimmte Ressource (etwa eine Datei oder ein Speicherbereich) oder sind Filter.

### 19.4.1 Die vier abstrakten Basisklassen

Da Klassen zum Lesen und Schreiben und Unicode-Zeichen von Byte getrennt werden müssen, gibt es folglich Klassen zur Ein-/Ausgabe von Bytes (oder Byte-Arrays) und Klassen zur Ein-/Ausgabe von Unicode-Zeichen (Arrays oder Strings). Die Klassen lauten:

Basisklasse für	Bytes (oder Byte-Arrays)	Zeichen (oder Zeichen-Arrays)
Eingabe	InputStream	Reader
Ausgabe	OutputStream	Writer

Tabelle 19.2 Basisklassen für Ein- und Ausgabe

Die vier Typen wollen wir *Stromklassen* nennen. Hier in den Klassen sind die zu erwartenden Methoden wie `read(...)` und `write(...)` zu finden.

Die Klassen `InputStream` und `OutputStream` bilden die Basisklassen für alle byteorientierten 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 die Methoden, da nur sie wissen, wie etwas tatsächlich gelesen oder geschrieben wird.

#### 19.4.2 Die abstrakte Basisklasse `OutputStream`

Wenn wir uns den `OutputStream` anschauen, dann sehen wir auf den ersten Blick, dass hier alle wesentlichen Operationen rund um das Schreiben versammelt sind. Der Clou bei allen Datenströmen ist, dass spezielle Unterklassen wissen, wie sie genau die vorgeschriebene Funktionalität implementieren. Das heißt, dass ein konkreter Datenstrom, der in Dateien oder in eine Netzwerkverbindung schreibt, weiß, wie Bytes in Dateien oder ins Netzwerk kommen. Und an der Stelle ist Java mit seiner Plattformunabhängigkeit am Ende, denn auf einer so tiefen Ebene können nur native Methoden die Bytes schreiben.

```
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-Arrays, 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`.
- `static OutputStream nullOutputStream()`  
Liefert einen `OutputStream`, der alle Bytes verwirft. Neu in Java 11.

Die `IOException` ist keine `RuntimeException` und muss behandelt werden.



## Beispiel

Die Klasse `ByteArrayOutputStream` ist eine Unterklasse von `OutputStream` und speichert alle Daten in einem internen byte-Array. Schreiben wir ein paar Daten mit den drei gegebenen Methoden hinein:

```
byte[] bytes = { 'O', 'N', 'A', 'L', 'D' };
//          0   1   2   3   4
ByteArrayOutputStream out = new ByteArrayOutputStream();
try {
    out.write( 'D' );           // schreibe D
    out.write( bytes );        // schreibe ONALD
    out.write( bytes, 1, 2 );   // schreibe NA
    System.out.println( out.toString( StandardCharsets.ISO_8859_1.name() ) );
}
catch ( IOException e ) {
    e.printStackTrace();
}
```

## Über konkrete und abstrakte Methoden \*

Zwei Eigenschaften lassen sich an den Methoden vom `OutputStream` ablesen: zum einen, dass nur Bytes geschrieben werden, und zum anderen, dass nicht wirklich alle Methoden `abstract` sind und von Unterklassen für konkrete Ausgabeströme überschrieben werden müssen. Nur `write(int)` ist abstrakt und elementar. Das ist trickreich, denn tatsächlich lassen sich die Methoden, die ein Byte-Array schreiben, auf die Methode abbilden, die ein einzelnes Byte schreibt. Wir werfen einen Blick in den Quellcode der Bibliothek:

**Listing 19.6** java/lang/OutputStream.java, Ausschnitt

```
public abstract void write(int b) throws IOException;

public void write(byte[] b) throws IOException {
    write(b, 0, b.length);
}

public void write(byte b[], int off, int len) throws IOException {
    if (b == null) {
        throw new NullPointerException();
    } else if ((off < 0) || (off > b.length) || (len < 0) ||
               ((off + len) > b.length) || ((off + len) < 0)) {
        throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return;
    }
    int n = len;
    while (n > 0) {
        int m = Math.min(n, 0x100);
        write(b, off, m);
        n -= m;
        off += m;
    }
}
```

```

    }
    for (int i = 0 ; i < len ; i++) {
        write(b[off + i]);
    }
}

```

An beiden Implementierungen ist zu erkennen, dass sie die Arbeit sehr bequem an andere Methoden verschieben. Doch diese Implementierung ist nicht optimal! Stellen wir uns vor, ein Dateiausgabestrom überschreibt nur die eine abstrakte Methode, die nötig ist. Und nehmen wir weiterhin an, dass unser Programm nun immer große Byte-Arrays schreibt, etwa eine 5-MiB-Datei, die im Speicher steht. Dann werden für jedes Byte im Byte-Array in einer Schleife alle Bytes der Reihe nach an eine vermutlich native Methode übergeben. Wenn es so implementiert wäre, könnten wir die Geschwindigkeit des Mediums überhaupt nicht nutzen, zumal jedes Dateisystem Funktionen bereitstellt, mit denen sich ganze Blöcke übertragen lassen. Glücklicherweise sieht die Implementierung nicht so aus, da wir in dem Modell vergessen haben, dass die Unterkasse zwar die abstrakte Methode implementieren muss, aber immer noch andere Methoden überschreiben kann. Ein Blick auf die Unterkasse `FileOutputStream` bestätigt dies.

### Hinweis

Ruft eine Oberklasse eine abstrakte Methode auf, die später die Unterkasse implementiert, ist das ein Entwurfsmuster mit dem Namen *Schablonenmuster* oder auf Englisch *template pattern*.



Gleichzeitig stellt sich die Frage, wie ein `OutputStream`, der die Eigenschaften für alle erdenklichen Ausgabeströme vorschreibt, wissen kann, wie ein spezieller Ausgabestrom etwa mit `close()` geschlossen wird oder seine gepufferten Bytes mit `flush()` schreibt – die Methoden müssten doch auch abstrakt sein! Das weiß `OutputStream` natürlich nicht, aber die Entwickler haben sich dazu entschlossen, eine leere Implementierung anzugeben. Der Vorteil besteht darin, dass Unterklassen nicht verpflichtet werden, die Methoden immer sinnvoll zu überschreiben.

### 19.4.3 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`. Die Klasse bietet mehrere `readXXX(...)`-Methoden und ist auch ein wenig komplexer als `OutputStream`.

```
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.
- `abstract int read() throws IOException`  
Liest ein Byte aus dem Datenstrom und liefert ihn zurück. Die Rückgabe ist »–1«, wenn der Datenstrom keine Daten mehr liefert. Falls Daten grundsätzlich noch verfügbar sind, blockiert die Methode. Der Rückgabetyp ist `int`, weil `–1` (`0xFFFFFFFF`) das Ende des Datenstroms anzeigt und ein `–1` als `byte` (das wäre `0xFF`) nicht von einem normalen Datum unterschieden werden könnte. Schade, dass es für `–1` in der Java-API keine Konstante gibt.<sup>5</sup>
- `int read(byte[] b) throws IOException`  
Liest bis zu `b.length` Bytes aus dem Datenstrom und setzt sie in das Array `b`. Die tatsächliche Länge der gelesenen Bytes wird zurückgegeben und muss nicht `b.length` sein, es können auch weniger Bytes gelesen werden. In der Basisklasse `InputStream` einfach als `return read(b, 0, b.length);` implementiert.
- `int read(byte[] b, int off, int len) throws IOException`  
Liest den Datenstrom aus und setzt die Daten in das Byte-Array `b`, an der Stelle `off` beginnend. Zudem begrenzt `len` die maximale Anzahl der zu lesenden Bytes. Intern ruft die Methode zunächst `read()` auf, und wenn es zu einer Ausnahme kommt, endet auch damit unsere Methode mit einer Ausnahme. Es folgen wiederholte Aufrufe von `read()`, die dann enden, wenn `read()` die Rückgabe »–1« liefert. Falls es zu einer Ausnahme kommt, wird diese aufgefangen und nicht gemeldet.
- `int readNBytes(byte[] b, int off, int len) throws IOException`  
Versucht, `len` viele Bytes aus dem Datenstrom zu lesen und in das Byte-Array zu setzen. Im Gegensatz zu `read(byte[], int, int)` übernimmt `readNBytes(...)` mehrere Anläufe, `len` viele Daten zu beziehen. Dabei greift es auf `read(byte[], int, int)` zurück.
- `byte[] readNBytes(int len) throws IOException`  
Versucht, `len` viele Bytes aus dem Strom zu lesen und in das Rückgabe-Array zu setzen. Das Array kann kleiner als `len` sein, wenn vorher das Ende des Stroms erreicht wurde.
- `byte[] readAllBytes() throws IOException`  
Liest alle verbleibenden Daten aus dem Datenstrom und liefert ein Array mit diesen Bytes als Rückgabe.

<sup>5</sup> Es könnten genutzt werden `TT_EOF` aus der Klasse `java.io.StreamTokenizer`. Oracle selbst wollte keine Konstante spendieren und hat die Anfrage schnell geschlossen: [https://bugs.java.com/bugdatabase/view\\_bug.do?bug\\_id=1204354](https://bugs.java.com/bugdatabase/view_bug.do?bug_id=1204354).

- `long transferTo(OutputStream out) throws IOException`  
Liest alle Bytes aus dem Datenstrom aus und schreibt sie in out. Wenn es zu einer Ausnahme kommt, wird empfohlen, beide Datenströme zu schließen.
- `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.
- `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)`  
Merkt sich die aktuelle Position im Datenstrom. Ist `markSupported()` gleich true, können danach mindestens `readlimit` Bytes gelesen und kann wieder zu der gemerkten Stelle zurückgesprungen werden.
- `void reset() throws IOException`  
Springt wieder zu der Position zurück, die mit `mark()` gesetzt wurde.
- `void close() throws IOException`  
Schließt den Datenstrom. Operation aus der Schnittstelle Closeable.
- `static InputStream nullInputStream()`  
Liefert einen InputStream, der keine Bytes liest.

Auffällig ist, dass bis auf `mark(int)` und `markSupported()` alle Methoden im Fehlerfall eine `IOException` auslösen. Bei der `read(...)`-Methode, die mehrere Bytes liest, wird immer ein Array als Puffer übergeben, in den der `InputStream` – das gleiche Prinzip finden wir aber auch beim Reader – hineinschreibt. Es gibt keine Methode, die als Rückgabe ein Array mit den gelesenen Daten liefert. Das hat zwei Gründe: Zum einen werden Puffer in der Regel wiederverwendet und wir hätten unter Umständen sehr viele Objekte, und zum anderen könnte der Datenstrom sehr groß oder sogar unendlich sein.

### Hinweis

`available()` liefert die Anzahl der 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.



#### 19.4.4 Die abstrakte Basisklasse Writer

Die abstrakte Klasse `Writer` ist der Basistyp für alle zeichenbasierten schreibenden Klassen.

```
abstract class java.io.Writer
implements Appendable, Closeable, Flushable
```

- `void write(int c) throws IOException`  
Schreibt ein einzelnes Unicode-Zeichen. Nur der niedrige Teil (16 Bit des `int`) der 32-Bit-Ganzzahl wird geschrieben.
- `void write(char[] cbuf) throws IOException`  
Schreibt ein Array von Zeichen.
- `abstract void write(char[] cbuf, int off, int len) throws IOException`  
Schreibt `len` Zeichen des Arrays `cbuf` ab der Position `off`.
- `void write(String str) throws IOException`  
Schreibt den String `str`.
- `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`.
- `Writer append(CharSequence csq, int start, int end) throws IOException`  
Hängt Teile einer 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. Das schreibt alle Puffer. Implementierung aus der Schnittstelle `Flushable`.
- `abstract void close() throws IOException`  
Schreibt den gepufferten Strom und schließt ihn. Nach dem Schließen durchgeführte `write(...)`- oder `flush()`-Aufrufe bringen eine `IOException` mit sich. Ein zusätzliches `close()` löst keine Exception aus. Implementierung aus der Schnittstelle `Closeable`.
- `static Writer nullWriter()`  
Ein `Writer`, der alle Zeichen verwirft. Die Rückgabe ist mit dem Unix-Device `/dev/null` vergleichbar. Neu in Java 11.
- Eine übergebene `null`-Referenz führt immer zur Ausnahme.

### Atomares Schreiben \*

Falls zwei Threads gleichzeitig in den Strom schreiben, gibt es Probleme. Um ein atomares Schreiben von mehr als einem Zeichen zu ermöglichen, referenziert der `Writer` intern ein Lock-Objekt zur Synchronisation: Betritt ein Thread den über ein Lock-Objekt geschützten Bereich, wird »abgeschlossen«, und ein anderer Thread, der über das gleiche Lock-Objekt eintreten will, muss warten, bis der erste Thread das Schreiben abgeschlossen hat und den Bereich wieder freigibt.

Die Klasse `Writer` referenziert in einer Objektvariablen `lock` das Synchronisationsobjekt als einfaches `java.lang.Object`. Zwei geschützte Konstruktoren beschreiben die Variable.

```
abstract class java.io.Writer
implements Appendable, Closeable, Flushable
```

- `protected Object lock`  
Objektvariable, mit der Schreiboperationen synchronisiert werden.
- `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.

Die Synchronisation wird also entweder durch ein eigenes `lock`-Objekt durchgeführt, das dann im Konstruktor angegeben werden muss, oder die Klasse verwendet das `this`-Objekt der `Writer`-Klasse als Sperrobject.

### Wie die abstrakten Methoden genutzt und überschrieben werden \*

Von den zehn Methoden in `Writer` sind lediglich drei abstrakt: `flush()`, `close()` und `write(char[], int, int)`. Zum einen bedeutet dies, dass konkrete Unterklassen nur diese Methoden implementieren müssen, und zum anderen, dass die übrigen `write(...)`-Methoden auf die eine überschriebene Implementierung zurückgreifen. Werfen wir einen Blick auf einen nichtabstrakten Nutznießer:

**Listing 19.7** `java.io.Writer, write(int)`

```
public void write(int c) throws IOException {
    synchronized (lock) {
        if (writeBuffer == null){
            writeBuffer = new char[WRITE_BUFFER_SIZE];
        }
    }
}
```

```

        writeBuffer[0] = (char) c;
        write(writeBuffer, 0, 1);
    }
}

```

Wird ein Zeichen geschrieben, so wird zunächst einmal nachgesehen, ob schon früher ein temporärer Puffer eingerichtet wurde (ein schöner Trick, denn Speicherbeschaffung ist nicht ganz billig). Wenn nicht, dann erzeugt die Methode zunächst ein Array mit der Größe von 1.024 Zeichen (dies ist die eingestellte Puffergröße WRITE\_BUFFER\_SIZE). Dann schreibt `write(int)` das Zeichen in den Puffer und ruft die abstrakte `write(char[], int, int)`-Methode auf – die ja in einer Unterklasse implementiert wird. Ist der Parameter ein Array, so muss lediglich die Größe an die abstrakte Methode übergeben werden. Neben `write(int c)` nutzt sonst nur noch `write(String str, int off, int len)` diesen internen `writeBuffer`. Und wenn `write(int c)` zuerst aufgerufen wird und danach `write(String, int, int)`, ist ein Puffer schon vorhanden – ein String kann nur geschrieben werden, indem er in ein char-Array kopiert wird und dann das char-Array geschrieben wird.

#### 19.4.5 Die Schnittstelle Appendable \*

Alle `Writer` und auch die Klassen `PrintStream`, `CharBuffer` sowie `StringBuffer` und `StringBuilder` implementieren die Schnittstelle `Appendable`, die drei Methoden vorschreibt:

`interface java.io.Appendable`

- `Appendable append(char c)`  
Hängt das Zeichen `c` an das aktuelle `Appendable` an und liefert das aktuelle Objekt vom Typ `Appendable` wieder zurück.
- `Appendable append(CharSequence csq)`  
Hängt die Zeichenfolge an dieses `Appendable` an und liefert es wieder zurück.
- `Appendable append(CharSequence csq, int start, int end)`  
Hängt einen Teil der Zeichenfolge an dieses `Appendable` an und liefert es wieder zurück.

#### Kovariante Rückgabe in `Writer` von `Appendable`

Die Klasse `Writer` demonstriert gut einen kovarianten Rückgabetyp, also dass der Rückgabetyp einer überschriebenen oder implementierten Methode ebenfalls ein Untertyp sein kann. So verfährt auch `Writer`, der die Schnittstelle `Appendable` implementiert. Die Methode `append(...)` in `Writer` besitzt nicht einfach den Rückgabetyp `Appendable` aus der Schnittstelle `Appendable`, sondern konkretisiert ihn zu `Writer`, das ein Untertyp von `Appendable` ist.

```
public Writer append( char c ) throws IOException {
    write( c );
    return this;
}
```

#### 19.4.6 Die abstrakte Basisklasse Reader

Die abstrakte Klasse Reader dient dem 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 `write(char[], int, int)` und `close()` implementieren müssen – und `flush()`, das es bei lesenden Strömen nicht gibt. 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 der Datenstrom keine Daten mehr liefert.
- `int read(char[] cbuf) throws IOException`  
Liest Zeichen aus dem Strom und schreibt sie in ein Array. Die Methode wartet, bis Eingaben anliegen. Der Rückgabewert ist die Anzahl der gelesenen Zeichen oder –1, wenn das Ende des Datenstroms erreicht wurde.

- `long transferTo(Writer out) throws IOException`  
Liest alle Zeichen aus diesen Strom und schreibt sie nach `out`. Neu seit Java 10.
- `abstract void close() throws IOException`  
Schließt den Strom. Folgt anschließend noch ein Aufruf von `read(...)`, `mark(int)` oder `reset()`, lösen die Methoden eine `IOException` aus. Ein doppelt geschlossener Stream hat keinen weiteren Effekt.
- `static Reader nullReader()`  
Liefert einen Reader, der keine Zeichen liest. Neu in Java 11.

### Get Ready

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 Standardimplementierung der abstrakten Klasse `Reader` gibt immer `false` zurück.



### Hinweis

Nehmen wir an, der Datenstrom soll komplett leergesaugt werden, bis keine Daten mehr verfügbar sind und der Datenstrom am Ende ist:

```
for ( int c; (c = reader.read()) != -1; )
    System.out.println( (char) c );
```

Wir könnten auf die Idee kommen, die Lösung anders zu realisieren:

```
while ( reader.ready() )
    System.out.println( (char) reader.read() );
```

Allerdings ist die Semantik eine ganz andere: Hier wird lediglich so lange gelesen, bis entweder der Datenstrom leer ist oder – und das ist der Punkt – es zum Blockieren kommt, wenn etwa über das Netzwerk keine Daten verfügbar sind und es etwas dauert, bis Nachschub kommt. Das heißt in diesem Fall: Wir bekommen nur die Daten bis zur Blockade, nicht aber alle Daten.

```
abstract class java.io.Reader
implements Readable, Closeable
```

- `boolean ready() throws IOException`  
Liefert `true`, wenn aus dem Stream direkt gelesen werden kann. Das heißt allerdings nicht, dass `false` immer Blocken bedeutet.



### Tipp

`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

Die Methode `mark(int)` markiert die Position, an der der `Reader` gerade steht. Ein Aufruf der Methode `reset()` setzt den Eingabestrom auf diese Position zurück, das heißt, diese Stelle lässt sich zu einem späteren Zeitpunkt wieder anspringen. `mark(int readAheadLimit)` besitzt einen Ganzahlparameter (`int`, nicht `long`), 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 diesen Rücksprung. 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(int)` und `reset()` ganz einfach und muss von uns im Bedarfsfall überschrieben werden:

**Listing 19.8** `java/io/Reader.java`, Ausschnitt

```
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. Die API-Dokumentation lässt es mit der Bemerkung »könnnte fehlschlagen« offen, wie die Methode reagieren soll, wenn in der Zwischenzeit mehr als `readAheadLimit` Zeichen gelesen wurden.

Reader implementiert die schon bekannte Schnittstelle `Closeable` mit der Methode `close()`. Und so, wie ein Writer die Schnittstelle `Appendable` implementiert, so implementiert ein Reader die Schnittstelle `Readable` und damit die Operation `int read(CharBuffer target) throws IOException`.

#### 19.4.7 Die Schnittstellen `Closeable`, `AutoCloseable` und `Flushable`

Zwei besondere Schnittstellen, `Closeable` und `Flushable`, schreiben Methoden vor, die alle Ressourcen implementieren, die geschlossen werden und/oder Daten aus einem internen Puffer herausschreiben sollen.

##### **Closeable**

Alle lesenden und schreibenden Datenstromklassen, die geschlossen werden können, implementieren `Closeable`. In der Java SE sind das alle Reader/Writer- und InputStream/OutputStream-Klassen und weitere Klassen wie `java.net.Socket`.

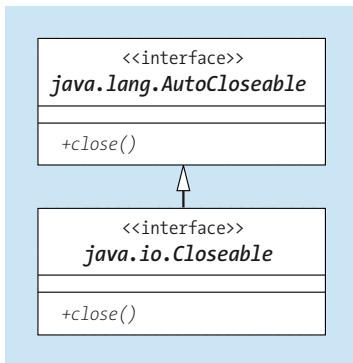
```
interface java.io.Closeable
extends AutoClosable
```

- `void close() throws IOException`  
Schließt den Datenstrom. Einen geschlossenen Strom noch einmal zu schließen, hat keine Konsequenz.

Die Schnittstelle `Closeable` erweitert `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 19.4** Das Klassendiagramm zeigt die Vererbungsbeziehung 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 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 `Writer` und `OutputStream` implementieren diese Schnittstelle, aber auch `Formatter` tut dies.

## 19.5 Lesen aus Dateien und Schreiben in Dateien

Um Daten aus Dateien lesen oder in Dateien schreiben zu können, ist eine Stromklasse nötig, die es schafft, die abstrakten Methoden von Reader, Writer, InputStream und OutputStream auf Dateien abzubilden. Um an solche Implementierungen zu kommen, gibt es drei verschiedene Ansätze:

- ▶ Die Utility-Klasse Files bietet einige newXXX(...)-Methoden, die uns Lese-/Schreib-Datenströme für zeichen- und byteorientierte Dateien liefern.
- ▶ Ein Class-Objekt bietet getResourceAsStream(...) und liefert einen InputStream, der Bytes aus Dateien im Modulpfad liest. Zum Schreiben gibt es nichts Vergleichbares. Falls Unicode-Zeichen gelesen werden sollen, muss der InputStream in einen Reader konvertiert werden.
- ▶ Die speziellen Klassen FileInputStream, FileReader, FileOutputStream, FileWriter sind Stromklassen, die read(...)/write(...)-Methoden auf Dateien abbilden. So gibt es weitere Klassen für spezielle Quellen und Senken, etwa Netzwerkverbindungen.

Jede der Varianten hat Vor- und Nachteile. Wir wollen die einzelnen Möglichkeiten nun kennlernen und voneinander abgrenzen.

### 19.5.1 Byteorientierte Datenströme über Files beziehen

Die Files-Klasse bietet Methoden, die direkt den Eingabe-/Ausgabestrom liefern. Beginnen wir mit den byteorientierten Stream-Klassen:

```
final abstract java.nio.file.Files
```

- static OutputStream newOutputStream(Path path, OpenOption... options)  
throws IOException  
Legt eine Datei an und liefert den Ausgabestrom auf die Datei.
- static InputStream newInputStream(Path path, OpenOption... options)  
throws IOException  
Öffnet die Datei und liefert einen Eingabestrom zum Lesen.

Da die OpenOption ein Vararg ist und somit weggelassen werden kann, 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 folgen die Ausmaße und schließlich die ARGB-Werte für die Pixelfarben.

**Listing 19.9** src/main/java/com/tutego/insel/stream/WriteTinyPPM.java, main()

```
try ( OutputStream out =
    Files.newOutputStream( Paths.get( "littlepic.tmp.ppm" ) ) ) {
    out.write( "P3 1 1 255 255 0 0".getBytes( StandardCharsets.ISO_8859_1 ) );
}
catch ( IOException e ) {
    e.printStackTrace();
}
```

PPM-Dateien können online konvertiert werden, etwa mit <https://convertio.co/de/ppm-jpg/>.

### 19.5.2 Zeichenorientierte Datenströme über Files beziehen

Neben den statischen Files-Methoden newOutputStream(...) und newInputStream(...) gibt es zwei Methoden, die zeichenorientierte Ströme liefern, also Reader/Writer:

`final abstract java.nio.file.Files`

- `static BufferedReader newBufferedReader(Path path, Charset cs)`  
throws IOException
- `static BufferedWriter newBufferedWriter(Path path, Charset cs, OpenOption... options)`  
throws IOException  
Liefert einen Unicode-Zeichen lesenden Ein-/Ausgabestrom. Das Charset-Objekt bestimmt, in welcher Zeichenkodierung sich die Texte befinden, damit sie korrekt in Unicode konvertiert werden.
- `static BufferedReader newBufferedReader(Path path)`  
throws IOException  
Entspricht `newBufferedReader(path, StandardCharsets.UTF_8)`.
- `static BufferedWriter newBufferedWriter(Path path, OpenOption... options)`  
throws IOException  
Entspricht `Files.newBufferedWriter(path, StandardCharsets.UTF_8, options)`.

BufferedReader und BufferedWriter sind Unterklassen von Reader/Writer, die zum Zwecke der Optimierung Dateien im internen Puffer zwischenspeichern.

#### newBufferedWriter(...)

Die Rückgabe von newBufferedWriter(...) ist ein BufferedWriter, eine Unterklasse von Writer. Jeder Writer hat Methoden wie `write(String)`, die Zeichenketten in den Strom schreiben. Die Methode soll das nächste Beispiel nutzen:

**Listing 19.10** src/main/java/com/tutego/insel/stream/NewBufferedWriterDemo.java, main()

```
try ( Writer out = Files.newBufferedWriter( Paths.get( "out.bak.txt" ),
                                             StandardCharsets.ISO_8859_1 ) ) {
    out.write( "Zwei Jäger treffen sich ..." );
    out.write( System.lineSeparator() );
}
catch ( IOException e ) {
    e.printStackTrace();
}
```

**newBufferedReader()**

Der BufferedReader bietet neben den einfachen geerbten Lesemethoden der Oberklasse Reader zwei weitere praktische Methoden:

- ▶ String readLine(): Liest eine Zeile bis zum Zeilenendezeichen (oder Stromende). Die Rückgabe ist null, wenn keine neue Zeile gelesen werden kann, weil das Stromende erreicht wurde. Das Zeilenendezeichen ist nicht Teil des Strings.
- ▶ Stream<String> lines(): Liefert einen Stream von Strings, wobei jeder String eine Zeile (ohne den Trenner) repräsentiert.

So einfach ist ein Programm formuliert, das alle Zeilen einer Datei abläuft:

**Listing 19.11** src/main/java/com/tutego/insel/stream/NewBufferedReaderDemo.java, main()

```
try ( BufferedReader in = Files.newBufferedReader( Paths.get( "lyrics.txt" ),
                                                 StandardCharsets.ISO_8859_1 ) ) {
    for ( String line; (line = in.readLine()) != null; )
        System.out.println( line );
}
catch ( IOException e ) {
    e.printStackTrace();
}
```

**Beispiel**

Mit der Stream-API sieht es ähnlich aus; kurz skizziert:

```
try ( BufferedReader in = Files.newBufferedReader( ... ) ) {
    in.lines().forEach( System.out::println );
}
```

Falls es beim Lesen über den Stream zu einem Fehler kommt, wird eine RuntimeException vom Typ `UncheckedIOException` ausgelöst.

### 19.5.3 Funktion von OpenOption bei den Files.newXXX(...)-Methoden

Sofern eine Datei schon existiert, wird sie beim Öffnen zum Schreiben sozusagen gelöscht und dann neu beschrieben; 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 und StandardOpenOption realisiert wird.

OpenOption	Beschreibung
<b>java.nio.file.StandardOpenOption</b>	
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 die Datei an, falls sie noch nicht existiert.
CREATE_NEW	Legt die Datei nur an, falls sie vorher noch nicht existierte. Andernfalls gibt es einen Fehler.
DELETE_ON_CLOSE	Die Java-Bibliothek versucht, die Datei zu löschen, wenn diese geschlossen wird. <sup>6</sup>
SPARSE	Hinweis für das Dateisystem, die Datei kompakt zu speichern, da sie aus vielen Null-Bytes besteht <sup>7</sup>
SYNC	Jeder Schreibzugriff und jedes Update der Metadaten soll sofort zum Dateisystem.
DSYNC	Jeder Schreibzugriff soll sofort zum Dateisystem.
<b>java.nio.file.LinkOption</b>	
NOFOLLOW_LINKS	Symbolischen Links wird nicht gefolgt.

Tabelle 19.3 Konstanten aus StandardOpenOption und LinkOption

<sup>6</sup> Das ist praktisch etwa, wenn der Hauptspeicher zu klein ist und Dateien zum Lesen/Schreiben als Zwischenspeicher verwendet werden. Am Ende der Operation kann die Datei gelöscht werden.

<sup>7</sup> Das ist für Windows und NTFS interessant, siehe auch <https://stackoverflow.com/questions/17634362/what-is-the-use-of-standardopenoption-sparse>.

Die Option CREATE\_NEW kann nur funktionieren, wenn die Datei noch nicht vorhanden ist.  
Das zeigt anschaulich das folgende Beispiel:

```
Listing 19.12 src/main/java/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 19.13 src/main/java/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(Path, LinkOption...) belegt.

#### 19.5.4 Ressourcen aus dem Modulpfad und aus JAR-Dateien laden

Im Klassen-/Modulpfad können neben den Klassendateien auch Ressourcen wie Grafiken oder Konfigurationsdateien enthalten sein. Der Zugriff auf diese Dateien wird nicht über Path oder File realisiert, denn die Dateien können sich innerhalb eines JAR-Archivs befinden. Daher gibt es am Class-Objekt: getResourceAsStream(String):

```
final class java.lang.Class<T>
implements Serializable, GenericDeclaration, Type, AnnotatedElement
```

- `InputStream getResourceAsStream(String name)`

Gibt einen Eingabestrom auf die Datei mit dem Namen `name` zurück oder `null`, falls es keine Ressource mit dem Namen im Modulpfad gibt oder die Sicherheitsrichtlinien das Lesen verbieten.

### Hinweis

Es geht bei der Methode nicht darum, eine Datei in einer anderen JAR-Datei zu öffnen und auszulesen.



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. Da die Methode keine Ausnahme auslöst, ist ein Test auf die Rückgabe ungleich `null` unabdingbar.

Das folgende Programm liest ein Byte aus der Datei `onebyte.txt` ein und gibt es auf dem Bildschirm aus:

**Listing 19.14** src/main/java/com/tutego/insel/io/stream/GetResourceAsStreamDemo.java

```
package com.tutego.insel.io.stream;
```

```
import java.io.*;
import java.util.Objects;

public class GetResourceAsStreamDemo {

    public static void main( String[] args ) {
        String filename = "onebyte.txt";
        try ( InputStream is = Objects.requireNonNull(
                GetResourceAsStreamDemo.class.getResourceAsStream( filename ),
                "Datei gibt es nicht!" ) ) {
            System.out.println( is.read() ); // 49
        }
        catch ( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

Die Datei `onebyte.txt` befindet sich im gleichen Pfad wie auch die Klasse, sie liegt also in `com/tutego/insel/io/stream/onebyte.txt`. Liegt die Ressource zum Beispiel im Wurzelverzeichnis des Pakets, lautet die Angabe `/onebyte.txt`. Liegen die Ressourcen außerhalb des Modulpfades, können sie nicht gelesen werden. Der große Vorteil ist aber, dass die Methode alle Ressourcen anzapfen kann, die über den Klassenlader zugänglich sind, und das ist insbesondere der Fall, wenn die Dateien aus JAR-Archiven kommen – hier gibt es keinen üblichen Pfad im Dateisystem, der hört in der Regel beim JAR-Archiv selbst auf.

Zum Nutzen der `getResourceAsStream(String)`-Methoden ist ein `Class`-Objekt nötig, das wir in unserem Fall über `Typname.class` besorgen. Das ist nötig, weil unsere `main(String[])`-Methode statisch ist. Andernfalls kann innerhalb von Objektmethoden auch `getClass()` eingesetzt werden, eine Methode, die jede Klasse aus der Basisklasse `java.lang.Object` erbt.

## 19.6 Zum Weiterlesen

Band 2, »Java SE 9 Standard-Bibliothek«, geht in zwei Kapiteln näher auf Dateien und Datenströme ein. Bei Dateien bietet uns die Klasse `Files` vielfache Möglichkeiten, etwa zum Auslesen der Attribute, zum Suchen im Dateisystem oder zum Horchen auf Veränderungen. Ein weiteres Kapitel geht intensiv auf das Filterkonzept ein und stellt alle Typen des Pakets `java.io` vor, etwa dass eine formatierte Textausgabe über eine spezielle Unicode-Kodierung in einen Byte-Strom kommt und dort dann einfach komprimiert und verschlüsselt werden kann.

# Kapitel 20

## Einführung ins Datenbankmanagement mit JDBC

»Was kann gespeichert werden? Alles, was Sie gerne extrahieren möchten.«  
– NSA-Präsentation zum System XKeyscore (2008)

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 system*) 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.

### 20.1 Relationale Datenbanken und Datenbankmanagementsysteme

#### 20.1.1 Das relationale Modell

Die Grundlage für relationale Datenbanken sind Tabellen, die auch *Relationen* genannt werden, mit ihren Spalten und Zeilen, die so genannten *Entitäten* Eigenschaften zuweisen. In der Vertikalen sind die Spalten und in der Horizontalen die Zeilen angegeben. Eine *Zeile* (auch *Tupel* genannt) entspricht einem Eintrag einer Tabelle, eine *Spalte* (auch *Attribut* genannt) einem Element einer Tabelle.

ID	Name	Adresse	Wohnort
004	Hoven G. H.	Sandweg 50	Linz
009	Baumgarten R.	Tankstraße 23	Hannover

Tabelle 20.1 Eine Beispieldatenebene

ID	Name	Adresse	Wohnort
011	Strauch GmbH	Beerweg 34a	Linz
013	Spitzmann	Hintergarten 9	Aalen
...	...	...	...

Tabelle 20.1 Eine Beispieldatenebene (Forts.)

Jede Tabelle entspricht einer logischen Sicht des Benutzers. Die Zeilen einer Relation stellen die *Datenbankausprägung* dar, während das *Datenbankschema* die Struktur der Tabellen – 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.

### 20.1.2 H2-Datenbank

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.

Die Beispiele im Buch basieren auf der H2-Datenbank (<http://www.h2database.com>), da sie so schön einfach ist und auch ohne Administratorrechte auskommt. Sie ist alleinstehend oder eingebettet lauffähig und verfügt über schöne Features. H2 hat eine Weboberfläche zur Konfiguration und für Abfragen und unterstützt alle wichtigen SQL-Eigenschaften wie Trigger, Joins, dazu abgesicherte Verbindungen und Volltextsuche. H2 hält außerdem den Speicherverbrauch klein. Weiterhin lässt sich ein ODBC-Treiber (von PostgreSQL) nutzen, um H2 auch unter Windows-Programmen (etwa Access) als Datenbank einzusetzen.

Da JDBC aber von Datenbanken abstrahiert, ist der Java-Programmcode natürlich auf jeder Datenbank lauffähig.

### 20.1.3 Weitere Datenbank-Management-Systeme \*

Die Anzahl der Datenbanken ist zwar groß, aber es gibt immer wieder Standarddatenbanken und freie Datenbankmanagementsysteme.

#### MySQL

MySQL (<http://www.mysql.de/>) ist ein häufig eingesetzter freier und schneller Open-Source-Datenbankserver. Er wird oft im Internet in Zusammenhang mit dynamischen Webseiten

eingesetzt; das Zusammenspiel zwischen Linux, Apache, MySQL, PHP (LAMP-System) ist hoch gelobt. Herausragende Eigenschaften sind die Geschwindigkeit und die Bedienbarkeit. Die MySQL-Datenbank spricht der unter der LGPL stehende JDBC-Treiber *MySQL Connector/J* (<https://dev.mysql.com/downloads/connector/j/>) an. Nach dem Entpacken muss das JAR-Archiv des Treibers in den Modulpfad aufgenommen werden. Er unterstützt die JDBC 4.2-API. Sun Microsystems hat im Februar 2008 MySQL übernommen, und heute gehört es zu Oracle. Aus der Unzufriedenheit mit der Weiterentwicklung von MySQL bei Oracle heraus ist MariaDB entstanden. Der MariaDB-Treiber ist bei gewissen Szenarios schneller.<sup>1</sup>

### **PostgreSQL**

Die *PostgreSQL*-Datenbank (<http://www.postgresql.org/>) ist ebenfalls quelloffen, läuft auf vielen Architekturen und unterstützt weitgehend den SQL-Standard 92. Gespeicherte Prozeduren, Schnittstellen zu vielen Programmiersprachen, Views und die Unterstützung für Geoinformationssysteme (GIS) haben das unter der BSD-Lizenz stehende PostgreSQL sehr beliebt gemacht. Es gibt moderne JDBC 4.1-Treiber unter <http://jdbc.postgresql.org/>.

### **HSQldb**

*HSQldb* (<http://hsqldb.org/>) ist ein relationales Datenbankmanagementsystem; es ist in purem Java programmiert und unterliegt 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. Sie ist ähnlich der H2-Datenbank und kann als Vorgänger gelten.

### **Oracle Database 11g Express Edition (Oracle Database XE)**

Um die Verbreitung ihrer Produkte weiter zu erhöhen, ist die Firma Oracle dazu übergegangen, eine vollwertige freie Version zum Download oder als CD anzubieten. Wer den Download von 2,5 GiB nicht scheut, der kann unter <https://www.oracle.com/database/technologies/appdev/xe.html> die *Oracle Database XE* für Windows, macOS, Linux und weitere Unix-Systeme herunterladen. Die JDBC-Treiber sind auf dem neuesten Stand.

### **DB2 Universal Database Express/DB2 Express-C**

Von IBM stammt die etwas eingeschränkte, aber freie Version von DB2 mit exzellenter Java-Unterstützung. Unter <http://tutego.de/go/db2express> lässt sich die Datenbank für Windows und Linux herunterladen.

---

<sup>1</sup> <https://mariadb.com/resources/blog/mariadb-java-connector-driver-performance/>

### Microsoft SQL Server und JDBC-Treiber

Mit der *SQL Server 2016 Express Edition* (<https://www.microsoft.com/de-de/sql-server/sql-server-editions-express>) bietet Microsoft eine freie Datenbank. Auch für die kommerzielle Version, den *Microsoft SQL Server*, bietet Microsoft unter <http://www.microsoft.com/de-de/download/details.aspx?id=11774> einen plattformunabhängigen JDBC 4.1- und JDBC 4.2-Treiber (erfordert Java 8).

## 20.2 JDBC und Datenbanktreiber

JDBC ist die inoffizielle Abkürzung für *Java Database Connectivity* und bezeichnet einen Satz von Schnittstellen zum Ansprechen relationaler Datenbanksysteme aus Java heraus. 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 differenzierender 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 (Open Database Connectivity). Dem Programmierer gibt die JDBC-API Methoden, die Verbindungen zu Datenbanken aufbauen, Datensätze lesen oder neue Datensätze verfassen. Zusätzlich können Tabellen aktualisiert und Prozeduren auf der Serverseite ausgeführt werden.



### Tipp

Ein JDBC-Treiber muss nicht unbedingt relationale Datenbanken ansprechen, obwohl das der häufigste Fall ist. Mit der freien Bibliothek *CsvJdbc* (<http://csvjdbc.sourceforge.net/>) ist etwa der Zugriff auf Comma-Separated-Value-(CSV-)Dateien möglich. Es gibt weitere Treiber für Excel-Tabellen.

### 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. Die Hersteller liefern in der Regel selbst einen JDBC-Treiber mit aus, den wir als JAR-Datei in den Modulpfad mit aufnehmen.

## 20.2.1 H2-Datenbank und JDBC-Treiber

Im Fall von H2 gibt es eine Besonderheit: Das Datenbankmanagementsystem und der Treiber sind zusammen in nur einem Java-Archiv gebündelt. Wir können also gut eine Dependency in unsere Maven POM-Datei hinzunehmen und H2 referenzieren:

**Listing 20.1** pom.xml, Ergänzung

```
<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.197</version>
</dependency>
```

## 20.2.2 JDBC-Versionen \*

Mit den Java-Versionen ist auch die Versionsnummer von JDBC gestiegen:

- ▶ JDBC 1.0. Die erste Version, die Sun im Jahr 1997 in Java 1.1 integrierte. JDBC 1.0 basiert auf SQL-92.
- ▶ JDBC 2.0 ist die nachfolgende Spezifikation. Sie berücksichtigt SQL-99 (SQL-3). Die Spezifikation der Version 2 setzt sich aus zwei Teilen zusammen: einer *JDBC 2.0 core API* und einer *JDBC 2.0 Optional Package API*. Die Core-API im Paket `java.sql` erweitert das Ur-JDBC um Batch-Updates und SQL 3-Datentypen. Das JDBC Optional Package liegt im Paket `javax.sql` und bietet unter anderem Data-Source, Connection-Pooling und verteilte Transaktionen. Während das Core-Paket fester Bestandteil von Java 1.2 war, ist das optionale Paket in Java 1.2 noch echt optional und erst in Java 1.3 fest integriert. Für fast alle Datenbanken gibt es JDBC 2.0-Treiber.
- ▶ JDBC 3.0 ist Teil von Java 1.4. Es integriert die JDBC 2.1 core API, das JDBC 2.0 Optional Package und nimmt neu unter anderem hinzu: Savepoints in Transaktionen, Wiederverwendung von Prepared Statements, die JDBC-Datentypen `BOOLEAN` und `DATALINK`, Abrufen automatisch generierter Schlüssel, Änderungen von LOBs und mehrere gleichzeitig geöffnete ResultSets.
- ▶ In Java 5 hat sich nicht viel an JDBC geändert. Es ist immer noch JDBC 3.0, doch sind JDBC-`RowSet`-Implementierungen hinzugekommen.
- ▶ JDBC 4.0 zog in Java 6 ein. Die JDBC-Treiber werden – wenn mit einer speziellen Metadatei vorbereitet – automatisch angemeldet. Weiterhin gibt es XML-Datentypen aus SQL:2003 und Zugriff auf die `SQL-ROWID`.
- ▶ Ein kleines Update auf JDBC 4.1 brachte Java 7 mit sich. Die API unterstützt etwa das neue Sprachfeature `try` mit Ressourcen, in dem JDBC-Typen `AutoCloseable` implementieren.

- ▶ Java 8 aktualisierte auf JDBC 4.2; die Änderungen waren minimal.<sup>2</sup>
- ▶ In Java 9 integriert das Maintenance Release 3 mit JDBC 4.3 vom JSR 221. In Java 10 und Java 11 hat sich bei der JDBC-API nichts verändert.



### Hinweis

Die aktuellen Treiber für die Datenbanken Oracle 12, Java DB (Apache Derby), DB 2 und MySQL implementieren einige JDBC 4-Eigenschaften, wenngleich nicht alle. Entwickler können nicht davon ausgehen, dass jede Datenbank und jeder Treiber alle JDBC-Operationen vollständig realisiert, was die Portabilität einschränken kann.

### Der Grad der SQL-Unterstützung

Einige Möglichkeiten lassen sich über die Metadaten einer Datenbank erfragen. Dazu zählt zum Beispiel, ob ein Treiber bzw. eine Datenbank den vollen ANSI-92-Standard unterstützt. Methoden wie `supportsANSI92XXXSQL()` eines `DatabaseMetaData`-Objekts liefern den Hinweis, ob die Datenbank *ANSI 92 Entry Level* (gilt immer), *Intermediate SQL* oder *Full SQL* unterstützt. Auch für ODBC gibt es unterschiedliche Level: *Minimum SQL Grammar*, *Core SQL Grammar* und *Extended SQL Grammar*.

## 20.3 Eine Beispielabfrage

Um mit der JDBC-API warm zu werden, fangen wir mit einem Beispiel an.

### 20.3.1 Schritte zur Datenbankabfrage

Folgende Schritte sind für einen Zugriff auf eine relationale Datenbank mit JDBC erforderlich:

1. Einbinden der JDBC-Datenbanktreiber in den Modulpfad
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

<sup>2</sup> <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>

Wir beschränken uns im Folgenden auf die Verbindung zum freien Datenbankmanagementsystem H2, wobei der Zugriff für jede relationale Datenbank – bis auf die JDBC-URL – gleich aussieht.

### 20.3.2 Mit Java auf die relationale Datenbank zugreifen

Unser Programm soll eine Datenbank »TutegoDB« nutzen, die wir aufbauen, wenn sie am Anfang leer ist. In JDBC werden wir den Namen in die JDBC-URL einbauen, doch die genaue Syntax ist bei jedem JDBC-Treiber etwas anders. H2 soll die Datenbank als Datei im Benutzerverzeichnis lokalisieren, und daher sieht die JDBC-URL so aus: `jdbc:h2:file:~/TutegoDB`. Wir können ebenso absolute Pfade angeben, etwa `jdbc:h2:file:C:/TutegoDB`. H2 liest 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. Ist die Datenbank als solche noch nicht angelegt, legt H2 sie automatisch an.

**Listing 20.2** src/main/java/com/tutego/insel/jdbc/FirstSqlAccess.java, main()

```
String url = "jdbc:h2:file:~/TutegoDB";
try ( Connection con = DriverManager.getConnection( url, "sa", "" );
      Statement stmt = con.createStatement() ) {

    // Tabelle CUSTOMER fehlt? Dann anlegen
    if ( ! con.getMetaData().getTables( null, null, "CUSTOMER", null ).next() ) {
        String[] sqlStmts = {
            "CREATE TABLE CUSTOMER(ID INTEGER NOT NULL PRIMARY KEY,FIRSTNAME VARCHAR(255),",
            "+ LASTNAME VARCHAR(255),STREET VARCHAR(255),CITY VARCHAR(255))",
            "INSERT INTO CUSTOMER VALUES(0,'Laura','Steel','429 Seventh Av.','Dallas')",
            "INSERT INTO CUSTOMER VALUES(1,'Susanne','King','366 - 20th Ave.','Olten')",
            "INSERT INTO CUSTOMER VALUES(2,'Anne','Miller','20 Upland Pl.','Lyon') ";
        for ( String sql : sqlStmts )
            stmt.executeUpdate( sql );
        System.out.println( "Tabelle und Daten neu angelegt" );
    }

    // Tabelle abfragen
    try ( ResultSet rs = stmt.executeQuery( "SELECT * FROM CUSTOMER" ) ) {
        while ( rs.next() )
            // Zugriff auf FIRSTNAME, LASTNAME, STREET
            System.out.printf( "%s, %s %s%n", rs.getString( 1 ),
                               rs.getString( 2 ), rs.getString( 3 ) );
    }
}
```

```
}

catch ( SQLException e ) {
    e.printStackTrace();
}
```

Bei ersten Start prüft das Programm, ob schon Tabellen in der Datenbank vorhanden sind; wenn nicht, legt es eine neue Tabelle an und fügt drei Datensätze ein.

## 20.4 Zum Weiterlesen

Der zweite Band der Insel (»Java SE 11 Standard-Bibliothek«) widmet sich ausführlicher der JDBC-API und stellt die Typen Connection, Statement, ResultSet detailliert vor. Ganz allgemein sollten sich Entwickler aber die Frage stellen, ob dieser simple Zugriff auf die Datenbank überhaupt nötig und sinnvoll ist, denn üblicherweise sollten Java-Entwickler in Objekten denken, und da liegen Datenbanktabellen auf einer anderen Abstraktionsebene. Gut, dass es objektrelationale Mapper gibt, die zwischen der objektorientierten Welt und den Relationen vermitteln. Das sollte für Entwickler der nächste Schritt nach JDBC sein, das Stichwort ist *JPA*.

# Kapitel 21

## Einführung in <XML>

»Ich bin überall in diesem Land gewesen und habe mit den besten Leuten gesprochen. Ich kann Ihnen versichern, dass Datenverarbeitung ein Tick ist, der sich nächstes Jahr erledigt hat.«

– Editor für Computerbücher bei Prentice Hall, 1957

### 21.1 Auszeichnungssprachen

Auszeichnungssprachen dienen der 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:

```
<Buch>
<Überschrift>Er kommt und geht<Ende Überschrift>
<Absatz>
Alle Kinder, bis auf einen, werden erwachsen. <Ort>Nimmerland<Ende Ort>
<Ende Absatz>
<Ende Buch>
```

Aus dem Kontext können wir Menschen semantische Informationen leicht erkennen. Damit der Computer das auch kann, geben wir zusätzliche Hinweise.

HTML ist die erste populäre Auszeichnungssprache, die Auszeichnungselemente (engl. *tags*) wie `<strong>fett</strong>` 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 fälschlicherweise mit dem Überschriften-Tag markiert, obwohl sie keine Überschrift ist.

### 21.1.1 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 anzugeben. 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 WiX<sup>1</sup>! 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/>.

## 21.2 Eigenschaften von XML-Dokumenten

### 21.2.1 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 zu einem Element ablegen:

**Listing 21.1** party.xml

```
<?xml version="1.0"?>
<party datum="31.12.2018">
  <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

---

<sup>1</sup> Windows Installer XML – Definition von Auslieferungspaketen für Microsoft Windows

Attributwert steht immer in einfachen oder doppelten Anführungszeichen, und das Gleichheitszeichen weist dem Attributnamen den Wert zu.

### Verwendung von Tags

Jedes XML-Element beginnt mit einem öffnenden Tag, *Start-Tag* genannt, zum Beispiel <party datum="31.12.2018>, in spitzen Klammern. Ein XML-Element mit einem Start-Tag muss auch mit einem End-Tag abgeschlossen werden, in unserem Beispiel </party>, andernfalls ist das XML nicht valide. Das Start-Tag gibt den Namen des Tags vor und enthält mögliche Attribute – das End-Tag hat den gleichen Namen wie das Anfangs-Tag und wird durch einen Schrägstrich nach der ersten Klammer gekennzeichnet.

*Element = öffnendes Tag + Inhalt + schließendes Tag*

Zwischen dem Anfangs- und dem End-Tag befindet sich der Inhalt des Elements.

#### Beispiel

Das Element <getraenk> mit dem Wert Wein:

```
<getraenk>Wein</getraenk>
```

[zB]

Hat ein Element keinen Inhalt, nennen wir es *leeres Element*. In diesem Fall kann eine abgekürzte Notation verwendet werden. Dann besteht das Element aus nur einem Tag mit einem Schrägstrich vor der schließenden spitzen Klammer.

#### Beispiel

Zwei Schreibweisen für das Element <zustand> mit den Attributen ledig und nuechtern:

```
<zustand ledig="true" nuechtern="false"></zustand>
<zustand ledig="true" nuechtern="false" />
```

[zB]

### 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 baumartige Struktur eines XML-Dokuments fest. Jedes XML-Dokument muss ein Wurzelement enthalten, das alle anderen Elemente einschließt.



### Beispiel

Das Wurzelement heißt <party> und schließt das Element <gast> ein:

```
<party datum="31.12.11">
  <gast name="Albert Angsthase"></gast>
</party>
```



### Hinweis

Nur weil ein XML-Dokument wohlgeformt ist, heißt es noch nicht, dass eine Anwendung es verarbeiten kann. Ein schärferes Kriterium ist die Validität: Es gibt ein Regelwerk, das ein valides XML-Dokument enthalten muss. Wir sprechen über valide Dokumente im Laufe des Kapitels noch einmal.

### 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 &amp;, &lt;, &gt;, &quot; und &apos;. 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 übergangen. 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="UTF-8"?>
```

Wenn eine XML-Deklaration vorhanden ist, muss sie ganz am Anfang des Dokuments stehen. Dort lässt sich im Prinzip 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 `&#xABCD;` (`A`, `B`, `C`, `D` stehen für Hexadezimalzeichen) dargestellt werden.

### Hinweis

Java und andere XML-Parser nehmen standardmäßig die Zeichenkodierung UTF-8 an. Es ist daher eine gute Idee, grundsätzlich alle XML-Dokumente in UTF-8 abzulegen.



## 21.2.2 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 nun schon ältere *Document Type Definition* (DTD), zu Deutsch *Dokumenttypdefinition*. 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 21.2 party.xml

```
<?xml version="1.0" ?>
<party datum="31.12.2017">
    <gast name="Albert Angsthase">
        <getraenk>Wein</getraenk>
        <getraenk>Bier</getraenk>
        <zustand ledig="true" nuechtern="false"/>
    </gast>
</party>
```

```

</gast>
<gast name="Martina Mutig">
    <getraenkl>Apfelsaft</getraenkl>
    <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	getraenkl und zustand	Die Gäste der Party; Name des Gastes als Attribut
getraenkl			Getränk des Gastes als Text
zustand	ledig und nuechtern		Familienstand und Zustand als Attribute

Tabelle 21.1 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:

- ▶ PCDATA (*Parsed Character Data*) für Text. Ist Standard für einen XML-Parser, wenn es keine DTD geben würde. Der Inhalt vom Text wird vom Parser analysiert. Wenn die Zeichen <, > und & vorkommen, leiten sie Tags ein oder eine Entity. Sollte also ein Getränk »Wodka & Specknote« heißen, dann wäre das ungültig als Text, es müsste »Wodka &#x26; Specknote« heißen.
- ▶ CDATA (*Unparsed Character Data*), bei deren <, > und & vorkommen dürfen. Falls Unterelemente vorkommen, sind sie Teil des Textes und werden vom Parser nicht als Unterelemente erkannt.
- ▶ 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 sie 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 21.2 DTD-Operatoren für Wiederholungen

[zB]

### Beispiel

Das Element `<party>` erlaubt beliebig viele Unterelemente vom Typ `<gast>`, wobei mindestens ein Gast auf einer Party sein muss – und nur mit einer Person ist es eine ziemlich öde »Party«. Nach oben ist die Anzahl nicht beschränkt, es kann eine Riesenparty werden.

```
<!ELEMENT party (gast)+>
```

Die runden Klammern können wir bei einem Unterelement auch weglassen.

### 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 21.3 Attribut-Modifizierer



### 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 Gastes, 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>`; das wird Sequenz genannt. Ein Guest 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 (#CDATA)>
```

Das Element `<zustand>` hat keinen Text und keine Unterelemente (ist `EMPTY`), 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

Eine DTD kann im XML-Dokument eingebettet sein, allerdings ist es praktisch, wenn die Deklaration in eine Datei ausgelagert wird. So können unterschiedliche XML-Dokumente eine gemeinsame DTD referenzieren. Die DTD-Datei kann dabei lokal auf dem Dateisystem stehen (private externe DTD für Entwickler) oder auf einem Webserver hinterlegt werden (öffentliche externe DTD für eine große Gruppe von Anwendern).

Falls die DTD in einer speziellen Datei steht, wird im Kopf der XML-Datei angegeben, wo die DTD für dieses XML-Dokument zu finden ist:

```
<!DOCTYPE party SYSTEM "dtd\partyfiles\party.dtd">
```

Hinter DOCTYPE ist das Wurzelement der zu beschreibenden XML-Datei angegeben, hinter SYSTEM der URI mit der Adresse der DTD-Datei.

Die vollständige DTD zu dem Party-Beispiel sieht folgendermaßen aus:

**Listing 21.3** party.dtd

```
<!ELEMENT party (gast)+>
<!ATTLIST party datum CDATA #REQUIRED>
<!ELEMENT gast (getraenk*, zustand?)>
<!ATTLIST gast name CDATA #REQUIRED>
<!ELEMENT getraenk (#CDATA)>
<!ELEMENT zustand EMPTY>
<!ATTLIST zustand ledig CDATA #IMPLIED nuechtern CDATA #IMPLIED>
```

Diese DTD definiert somit die Struktur aller XML-Dateien, die unsere Party beschreiben.

Bei einer öffentlichen DTD wird das Schlüsselwort PUBLIC statt SYSTEM verwendet.

### Beispiel



Bind für eine XHTML -Datei die DTD ein:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
 "http://www.w3.org/TR/REC-html40/loose.dtd">
```

### 21.2.3 Schema – die moderne Alternative zu DTD

Ein anderes Verfahren, die Struktur von XML-Dateien zu beschreiben, ist das *XML Schema*, abgekürzt XSD (von *XML Schema Definition*). 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 Vergleich 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 21.4** 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="getraenck" type="xsd:string" />
      <xsd:element name="zustand" type="zustandType" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" />
  </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 `<getraenk>`) 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 Zahlenspektrum 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.

### 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.

#### Beispiel

Das Element `<getraenk>` besteht nur aus einer Zeichenkette:

```
<xsd:element name="getraenk" type="xsd:string" />
```

[zB]

Komplexe Typen können neben textbasiertem Inhalt auch Unterelemente und Attribute inkludieren.

#### 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:attribute name="name" type="xsd:string" />
</xsd:complexType>
```

[zB]

Simple und komplexe Typen können andere Typen einschränken. Komplexe Typen können zusätzlich andere Typen erweitern. Beim Erweitern ist es möglich, mehrere Typen miteinan-

der 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.

### XSD einbinden

Eine XML-Datei kann eine XSD-Datei referenzieren, und zwar mit `xsi:schemaLocation` (mit XML-Namensraum) oder `xsi:noNamespaceSchemaLocation` (wenn kein Namensraum verwendet wird).

#### 21.2.4 Namensraum (Namespace)

Als Java-Entwickler kennen wir Pakete, damit die Typen unterschiedlicher Hersteller nicht durcheinanderkommen. Selbst in der Java-Bibliothek haben wir `List` in `java.util` und `java.awt`, und ohne Paket müssen die Typen unterschiedlichen heißen.

Auch XML-Dokumente könnten Fragmente anderer »Hersteller« enthalten, und da sollte es möglich sein, die Elemente eindeutig zuzuschreiben. Die Lösung in XML dafür ist ein *Namensraum*. Das Konzept 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 einem 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. Der 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="31.12.2017">
  <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.

### 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 der 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 der 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. Der URI dient nur der Identifikation des Namensraums für eine XML-Datei.

## 21.2.5 XML-Applikationen \*

Eine *XML-Applikation* ist eine festgelegte Auswahl von XML-Elementen und einem Namensraum – im besten Fall mit einer Schema-Datei. 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.

Bekannte XML-Applikationen sind:

- ▶ SVG (Scalable Vector Graphics): Beschreibung von Vektorgrafiken
- ▶ MathML: Beschreibung von Formeln
- ▶ XHTML: HTML-Seiten, die XML-konform sind.<sup>2</sup>

## 21.3 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, XStream, Castor ...*

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 StAX und SAX sind:

- ▶ sequenzieller Zugriff auf Elemente
- ▶ die Suche nach bestimmten Inhalten
- ▶ das Einlesen von XML-Teilstrukturen, 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

---

<sup>2</sup> XHTML spielt im Webumfeld keine große Rolle mehr, denn HTML5 ist der Standard, und HTML5 ist kein valides XML, denn es gibt viele so genannte Single-Elemente wie `<br>`, die kein End-Tag haben. XHTML gibt es auch, aber das interessiert so gut wie niemanden. Siehe auch <https://www.w3.org/TR/html5/introduction.html#html-vs-xhtml> und <https://www.w3.org/TR/html5/the-xhtml-syntax.html>.

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.

### 21.3.1 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, die den XML-Baum in eine Baumdatenstruktur abbildet. DOM definiert eine Reihe von Schnittstellen, die durch konkrete Programmiersprachen implementiert werden. So lassen sich die Bäume gut im Speicher aufbauen und dynamisch ändern. Mit XPath lassen sich Elemente im DOM-Baum mit einer speziellen Anfragesprache erfragen.

### 21.3.2 Pull-API StAX

Eine Pull-API wie StAX fordert aktiv Elemente von XML-Dokumenten an. Das Prinzip entspricht dem Iterator-Design-Pattern, das auch von der Collection-API oder dem Scanner bekannt ist. Aktiv holt sich ein Programm sequenziell Element für Element aus dem XML-Dokument und kann jederzeit stoppen, wenn die gewünschte Information gefunden wurde. Neben der guten Performance ist ein weiterer Vorteil der geringe Speicherbedarf: Bei StAX muss nicht der XML-Baum im Speicher abgelegt werden, und StAX eignet sich daher auch für sehr große Dokumente.

StAX unterscheidet die beiden Verarbeitungsmodelle *Iterator* und *Cursor*. 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.

### 21.3.3 Simple API for XML Parsing (SAX)

SAX lässt sich als Vorgänger von StAX sehen. Doch statt die XML-Elemente aktiv aus der XML-Struktur zu ziehen, basiert SAX auf einem Ereignismodell. SAX liest die XML-Datei wie einen Datenstrom und ruft per Callback eine Methode auf und meldet das Ereignis. Dies ist aber mit dem Nachteil verbunden, dass wahlfreier Zugriff auf ein einzelnes Element nicht ohne Zwischenspeicherung möglich ist. Wohl ist dieses Verfahren – wie auch StAX – prima beim Extrahieren bestimmter Informationen.

Die Arbeit an der Simple API for XML begann im Dezember 1997 als Gegenentwurf zu DOM. <http://www.saxproject.org/> gibt eine Dokumentation und weitere Verweise. Heutzutage spielt SAX im Java-Umfeld keine Rolle mehr.

### 21.3.4 Java Document Object Model (JDOM)

JDOM ist eine einfache Möglichkeit, XML-Dokumente leicht und effizient mit einer schönen Java-API zu nutzen. Die aktuelle Entwicklung von JDOM geht von Jason Hunter und Rolf Lear aus, die erste Version hat Brett McLaughlin mitgestaltet.

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 das 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* (<https://dom4j.github.io>) oder *XOM* (<http://www.xom.nu/>) gesellen sich dazu. Obwohl es um die Entwicklung von JDOM lange Zeit still war, zählt JDOM immer noch zu den populärsten<sup>3</sup> XML-APIs, wohl auch wegen der üppigen Dokumentation.

### 21.3.5 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 usw. Das JDK integriert Apache Xerces (<http://tutego.de/go/xerces>), das alle möglichen XML-Standards implementiert:

- ▶ XML 1.0 (4th Edition)
- ▶ Namespaces in XML 1.0 (2nd Edition)
- ▶ XML 1.1 (2nd Edition)
- ▶ Namespaces in XML 1.1 (2nd Edition)
- ▶ W3C XML Schema 1.0 (2nd Edition)
- ▶ W3C XML Schema 1.1
- ▶ W3C XML Schema Definition Language (XSD): Component Designators  
(Candidate Recommendation, January 2010)
- ▶ XInclude 1.0 (2nd Edition)
- ▶ OASIS XML Catalogs 1.1
- ▶ SAX 2.0.2
- ▶ DOM Level 3 Core, Load and Save

---

<sup>3</sup> Laut <http://www.servlets.com/polls/results.tea?name=doms>

- ▶ DOM Level 2 Core, Events, Traversal and Range
- ▶ Element Traversal (org.w3c.dom.ElementTraversal)
- ▶ JAXP 1.4
- ▶ StAX 1.0 Event API (javax.xml.stream.events)

XSL-Stylesheet-Transformationen werden standardmäßig über einen *Compiling XSLT Processor* (XSLTC) verarbeitet.

### 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:

- ▶ XML 1.0, XML 1.1
- ▶ DOM Level 3
- ▶ W3C XML Schema 1.0
- ▶ XSLT 1.0
- ▶ XInclude 1.0
- ▶ XPath 1.0
- ▶ SAX 2.0.2
- ▶ StAX 1.2 (JSR 173)

Die Parser validieren mit DTD oder einem W3C-XML-Schema und können mit *XInclude* Dokumente integrieren. Von DOM werden *DOM Level 3 Core* und *DOM Level 3 Load and Save* unterstützt.

JAXP wurde immer wieder aktualisiert. Da von der API unterschiedliche Implementierungen möglich sind, können Entwickler prinzipiell zwischen verschiedenen Parsern und XSLT-Transformern wählen, ohne den eigentlichen Code zu verändern. Das ist das gleiche Prinzip wie bei den Datenbanktreibern.

#### 21.3.6 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 21.5** src/main/java/com/tutego/insel/xml/dom/DOMParty.java, main()

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
try ( InputStream in = Files.newInputStream( Paths.get( "party.xml" ) ) ) {
    Document document = builder.parse( in );
```

```

Element partyRoot = document.getDocumentElement();
NodeList guests = partyRoot.getElementsByTagName( "gast" );
Node firstGuest = guests.item( 0 );
if ( firstGuest.getNodeType() == Node.ELEMENT_NODE ) {
    NodeList drinks = ((Element) firstGuest).getElementsByTagName( "getraenke" );
    System.out.println( drinks.item( 0 ).getTextContent() );
}
}

```

Die Ausgabe ist: »Wein«.

Die Parser sind selbstständig bei DocumentBuilderFactory angemeldet, und newInstance() liefert eine Unterklasse des DocumentBuilder.

## 21.4 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.

### 21.4.1 JAXB raus aus der Java SE, Abhängigkeit rein in die POM

JAXB ist ein Java-Enterprise-Standard, und eine Implementierung war von Java SE 6 bis Java SE 10 auch Teil vom JDK; in Java 11 hat Oracle das Modul `java.xml.bind` mit JAXB wieder aus dem Java SE-Standard entfernt, zusammen mit dem gesamten `java.se.ee`-Modul. Wir wollen daher JAXB über Maven integrieren und setzen in unsere POM-Datei folgende Abhängigkeiten:

**Listing 21.6** pom.xml, Ausschnitt

```

<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>

<dependency>
    <groupId>org.glassfish.jaxb</groupId>

```

```

<artifactId>jaxb-runtime</artifactId>
<version>2.3.0</version>
</dependency>

<dependency>
<groupId>javax.activation</groupId>
<artifactId>activation</artifactId>
<version>1.1.1</version>
</dependency>

```

### 21.4.2 Bean für JAXB aufbauen

Wir wollen einen Player deklarieren, und JAXB soll ihn anschließend in ein XML-Dokument übertragen:

**Listing 21.7** src/main/java/com/tutego/insel/xml/jaxb/Player.java, Player

```

@XmlRootElement
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 und Anpassungen am Namen gewünscht sind. Die Annotation stammt aus dem Paket `javax.xml.bind.annotation`.

### 21.4.3 Utility-Klasse JAXB

Ein kleines Testprogramm baut eine Person auf und bildet sie dann in XML ab – die Ausgabe der Abbildung kommt auf den Bildschirm. Um ein Objekt in seine XML-Präsentation zu bringen, lässt sich auf eine einfache statische `marshal(...)`-Methode der Utility-Klasse JAXB zurückgreifen:

**Listing 21.8** src/main/java/com/tutego/insel/xml/xml/jaxb/SimplePlayerMarshaller.java, main()

```
Player johnPeel = new Player();
johnPeel.setName( "John Peel" );
johnPeel.setBirthday( new GregorianCalendar(1939,Calendar.AUGUST,30).getTime() );
JAXB.marshal( johnPeel, System.out );
```

Die Klasse `Player` hat die JavaBean-Methoden `setBirthday(...)/getBirthday()` sowie `setName(...)/getName()` für die Bean-Properties `birthday` und `name`. JAXB ruft beim Generieren der XML-Struktur automatisch die Getter auf und enthält davon die Daten; die XML-Elemente heißen standardmäßig so wie die Properties:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
    <birthday>1939-08-30T00:00:00+01:00</birthday>
    <name>John Peel</name>
</player>
```

Die `marshal(...)`-Methode schreibt das in XML transformierte `Player`-Objekt in unserem Beispiel in den Konsolenausgabestrom. Die JAXB-Klasse bietet daneben auch `unmarshal(...)`-Methoden, die das Objekt wiederherstellen, das schauen wir uns im nächsten Beispiel an.

Falls das Schreiben oder Lesen misslingt, gibt es eine `RuntimeException`. Diese referenziert die intern geprüften Ausnahmen vom Typ `DataBindingException`.

#### 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.
  - 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-Objektgraphen.

#### 21.4.4 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 @XmlElement bekommen, und bei Player kann sie entfernt werden, wenn nur der Raum selbst, aber keine Player als Wurzelobjekte zum Marshaller kommen:

**Listing 21.9** src/main/java/com/tutego/insel/xml/xml/jaxb/Room.java, Room

```
@XmlRootElement( namespace = "http://tutego.com/" )
public class Room {

    private List<Player> players = new ArrayList<>();

    @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, damit XML-Doku-

mente gemischt werden können. 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:

**Listing 21.10** src/main/java/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 ) );

Path path = Paths.get( "room.xml" );
try ( Writer out = Files.newBufferedWriter( path, StandardCharsets.UTF_8 ) ) {
    JAXB.marshal( room, out );

    Room room2 = JAXB.unmarshal( path.toFile(), Room.class );
    System.out.println( room2.getPlayers().get( 0 ).getName() ); // John Peel
}
catch ( IOException e ) {
    e.printStackTrace();
}
```

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>
```

Verwendet JAXB Namensräume, nutzt es einen *Präfix-Mechanismus*. Er heißt so, weil vor dem Elementnamen ein Präfix steht, das JAXB immer »ns2« nennt; ein Doppelpunkt steht

als Trenner dazwischen.<sup>4</sup> Den eigentlichen Namensraum deklariert das Attribut »xmlns«. Alle Elemente unter room sind automatisch im gleichen Namensraum.

Da beim Spieler das Geburtsdatum nicht gesetzt war (null wird referenziert), wird es auch nicht in XML abgebildet.

#### 21.4.5 JAXBContext und Marshaller/Unmarshaller nutzen

Die Utility-Klasse JAXB verfügt ausschließlich über statische überladene `marshal(...)`- und `unmarshal(...)`-Methoden, ist aber in Wirklichkeit nur eine Fassade. Vielmehr ist `JAXBContext` das tatsächliche Ausgangsobjekt, und davon erfragt werden ein Marshaller zum Schreiben und ein Unmarshaller zum Lesen:

**Listing 21.11** src/main/java/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 );
```

Die Ausgabe ist identisch mit der ersten:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
    <birthday>1939-08-30T00:00:00+01:00</birthday>
    <name>John Peel</name>
</player>
```

#### JAXBContext: Quelle von Marshaller, Unmarshaller und Validator

Alles bei JAXB beginnt mit der zentralen Klasse `JAXBContext`. Die statische Methode `JAXBContext.newInstance(Class...)` erwartet standardmäßig eine Aufzählung der Klassen, die JAXB behandeln soll; die Klassennamen können auch voll qualifiziert über einen String angegeben werden mit `newInstance(String)`. Der `JAXBContext` liefert den Marshaller zum Schreiben und den Unmarshaller zum Lesen. Die Fabrikmethode `createMarshaller()` gibt dabei einen Schreiberling, der mit `marshal(Object, ...)` das Wurzelobjekt in einen Datenstrom schreibt. Das zweite Argument von `marshal(Object, ...)` ist unter anderem ein `OutputStream` (wie `System.out` in unserem Beispiel), `Writer` oder `File`-Objekt.

---

<sup>4</sup> Viele Entwickler stören sich an dem Namen. Das Internet ist voll verzweifelter Seelen, die das Präfix ändern möchten. Das ist dank `package-info` und der Paket-Annotation `@XmlSchema` möglich.

```
abstract 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 Objektgraphen 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 bzw. XMLStreamWriter.
  - void setProperty(String name, Object value)  
Setzt eine Eigenschaft der Marshaller-Implementierung. Eine Einrückung etwa setzt das Paar Marshaller.JAXB\_FORMATTED\_OUTPUT, Boolean.TRUE.

Der Unmarshaller bietet über zehn Varianten von unmarshal(...), unter anderem mit den Parametertypen File, InputSource, InputStream, Node, Reader, URL, XMLEventReader.

### Tipps und Tricks mit JAXBContext

Den JAXBContext aufzubauen kostet Zeit und Speicher. Er sollte daher für wiederholte Operationen gespeichert werden. Noch eine Information: Marshaller und Unmarshaller sind nicht threadsicher; es darf keine zwei Threads geben, die gleichzeitig den Marshaller/Unmarshaller nutzen.

Die unmarshal(...)-Methode ist überladen mit Parametern, die typische Datenquellen repräsentieren, etwa Dateien oder Eingabestrome wie ein Reader. Allerdings sind noch andere Parametertypen interessant, und es lohnt sich, hier einmal in die API-Dokumentation zu schauen. Ein spannender Typ ist javax.xml.transform.Source bzw. die Implementierung der

Schnittstelle durch JAXBSource. JAXBSource ist die Quelle, aus der JAXB seine Informationen bezieht, um ein neues Java-Objekt zu rekonstruieren.

Das nächste Beispiel nimmt sich das aus dem vorangehenden Beispiel aufgebaute Objekt room als Basis und erzeugt eine tiefe Kopie davon:

**Listing 21.12** src/main/java/com/tutego/insel/xml/xml/jaxb/RoomCopy.java, main() Ausschnitt

```
Room room = ...
JAXBContext context = JAXBContext.newInstance( Room.class );
Unmarshaller unmarshaller = context.createUnmarshaller();
JAXBSource source = new JAXBSource( context, room );
Room copiedRoom = Room.class.cast( unmarshaller.unmarshal( source ) );
System.out.println( copiedRoom.getPlayers() ); // [
com.tutego.insel.xml.jaxb.Player@... ]
```

Das Beispiel zeigt somit, wie sich mithilfe von JAXB Objektkopien erzeugen lassen.

## 21.5 Zum Weiterlesen

Als XML sich etablierte, bildete es mit Java ein gutes Gespann. Einer der Gründe lag in Unicode: XML ermöglicht Dokumente mit beliebigen Zeichenkodierungen, die in Java abgebildet werden konnten. Mittlerweile ist diese Abbildung nicht mehr so einfach, da in XML schnell eine Kodierung mit 32 Bit ausgewählt werden kann, die in Java nur Surrogate abbildet – nun macht die Verarbeitung nicht mehr richtig Spaß.

Das online unter <http://tutego.de/go/xmlbook> frei verfügbare Buch »Processing XML with Java« von Elliotte Rusty Harold gibt einen guten Überblick über die Funktionen von JDOM und die Verarbeitung von XML mit Java. Genauere Informationen finden sich auf der Webseite von JDOM (<http://www.jdom.org>). Zur JAXB gibt <http://jaxb.java.net/tutorial> tiefere Einblicke. JAXB ist sehr nützlich, insbesondere für eigene XML-Formate; für Standardformate wie RSS-Feeds, SVG, MathML, OpenDocument oder XUL gibt es in der Regel schon Zugriffsklassen, sodass die Daten nicht aus rohen XML-Dokumenten extrahiert werden müssen – eine objektorientierte Vorgehensweise ist immer besser, als in XML-Zeichenketten direkt zu lesen und diese zu verändern. XPath als Anfragesprache für XML auch für beliebige Java-Objekte zu erweitern, hat sich das Apache-Projekt JXPath (<http://tutego.de/go/jxpath>) zum Ziel gesetzt. Es ist auf jeden Fall einen Blick wert.



# Kapitel 22

## Bits und Bytes, Matematisches und Geld

»Es gibt nur 10 Typen von Menschen auf der Welt.  
Die, die Binärkode verstehen, und die, die ihn nicht verstehen.«

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.

### 22.1 Bits und Bytes

Ein *Bit* ist ein Informationsträger für die Aussage wahr oder falsch, also mit zwei Zuständen. Durch das Zusammensetzen von einzelnen Bits entstehen größere Folgen wie das *Byte*, das aus 8 Bit besteht, zum Beispiel  $00010011_{\text{bin}}$ . Da die genannte Zahl durch die Ziffern 0 und 1 durchaus eine Dezimalzahl sein könnte, stellen wir hinter die Zahl ein kleines  $_{\text{bin}}$ , wenn es zu Missverständnissen kommen kann.

Werden 8 Bit = 1 Byte zugrunde gelegt, lassen sich durch unterschiedliche Belegungen 256 unterschiedliche Zahlen bilden. Ist kein Bit des Bytes gesetzt, so ist die Zahl 0; ist jedes Bit gesetzt, ist die Zahl 255. Jedes Bit an einer Position im Byte kann 0 oder 1 sein, und daraus ergibt sich eine Wertigkeit, an welcher Stelle 0 oder 1 in der Folge steht. Wir sprechen von einem Stellenwertsystem, bei dem die Position einer Ziffer entscheidend ist, denn 12 ist nicht gleich 21, und  $10_{\text{bin}}$  ist nicht gleich  $01_{\text{bin}}$ . Jetzt stellt sich nur noch die Frage, wie die Bitnummerierung ist, denn das Bit mit dem niedrigsten Stellenwert könnte links oder rechts stehen – üblicherweise steht es rechts und das Bit mit dem höchsten Stellenwert links.

Beim Dualsystem haben wir die Ziffern 0 und 1. Der Wert einer Dualzahl ergibt sich durch Addition der Ziffern 0 oder 1, die vorher jeweils mit ihrem Stellenwert  $2^i$  multipliziert werden. Die Werteberechnung für die Zahl 19 berechnet sich aus  $16 + 2 + 1$ , da sie aus einer Anzahl von Summanden der Form  $2^i$  zusammengesetzt ist:  $19_{\text{dez}} = 16 + 2 + 1 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 10011_{\text{bin}}$ .

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 22.1 Werteberechnung

Java bietet zum Modifizieren von Bits

- ▶ bitweise Operatoren und
- ▶ bitweise Verschiebungen (Schiebeoperationen).

Diese schauen wir uns jetzt an.

### 22.1.1 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 das Komplement eines Operanden und Verknüpfungen mit anderen Werten. Durch die bitweisen Operatoren können einzelne Bits abgefragt und manipuliert werden.

Operator	Bezeichnung	Aufgabe
<code>~</code>	Komplement (bitweises Nicht)	Invertiert jedes Bit: Aus 0 wird 1, aus 1 wird 0.
<code> </code>	Bitweises Oder	Bei <code>a   b</code> wird jedes Bit von <code>a</code> und <code>b</code> einzeln Oder-verknüpft.
<code>&amp;</code>	Bitweises Und	Bei <code>a &amp; b</code> wird jedes Bit von <code>a</code> und <code>b</code> einzeln Und-verknüpft.
<code>^</code>	Bitweises exklusives Oder (XOR)	Bei <code>a ^ b</code> wird jedes Bit von <code>a</code> und <code>b</code> einzeln XOR-verknüpft; es ist kein <code>a</code> hoch <code>b</code> .

Tabelle 22.2 Bit-Operatoren in Java

Das Komplement ist ein unärer Operator, die anderen sind binäre Operatoren. 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.<sup>1</sup>

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 22.3 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 22.4 Binäre Verknüpfung zweier Ganzzahlen

## 22.1.2 Repräsentation ganzer Zahlen in Java – das Zweierkomplement

Der Rechner speichert Daten in den Zuständen 0 und 1 und muss positive sowie negative Zahlen darstellen können. Dazu gibt es unterschiedliche Kodierungen, wie das Einer- oder Zweierkomplement, wobei Letzteres für moderne Programme üblich ist, auch weil die internen Schaltungen im Prozessor einfacher sind.

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.

---

<sup>1</sup> Auf XOR können wir im Grunde verzichten, weil es sich durch die anderen Operatoren ausdrücken lässt:  
 $a^b$  ist wie  $(a \& \sim b) | (\sim a \& b)$ .

- Um eine 0 darzustellen, ist kein Bit gesetzt.
- Bei negativen Zahlen: Wir nehmen den Absolutwert der Zahl, invertieren das Bitmuster und addieren 1 zu dem Ergebnis.

[zB]

**Beispiel**

Testen wir das an der Zahl  $-1$ : Der Absolutwert ist 1, das Bitmuster (für 16 Bit) ist 0000 0000 0000 0001. Die Negation ergibt 1111 1111 1111 1110. Addieren wir 1, folgt 1111 1111 1111 1111.

**Java-Ganzzahldatentypen im Zweierkomplement**

Java kodiert die Ganzzahldatentypen `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 Bit-Maske 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 22.5 Darstellungen des Zweierkomplements im Datentyp `short`

Bei allen negativen Ganzzahlen ist also das oberste Bit mit 1 gesetzt.



### Hinweis

Zweierkomplement und Komplement (bitweises Nicht) haben eine interessante Beziehung.  $\sim 0$  invertiert alle Bits, was im Ergebnis -1 ergibt. Allgemein ist  $\sim a$  gleich  $-(a+1)$ . Also ist zum Beispiel  $\sim 5$  gleich  $-(5+1) = -6$ . Auch negiert gilt das:  $\sim -5$  ist gleich  $-(-5+1) = 4$ . Und  $\sim -\sim -\sim -\sim -5$  ist 1; damit lassen sich tolle Späße machen ...

### 22.1.3 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-*, *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 22.6 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 (bzw. A bis F). Das Hexadezimalsystem heißt auch *Sedenzimalsystem*.<sup>2</sup>

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«, das auf Deutsch »acht« heißt. Das Oktalsystem war früher eine verbreitete Darstellung, da nicht mehr einzelne Bits solo betrachtet werden mussten, sondern 3 Bit zu einer Gruppe zusammengefasst wurden. In der Kommunikationselektronik ist das Oktalsystem noch weiterhin beliebt, spielt aber sonst keine Rolle.

2 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 Sedenzimalzahl bzw. sedesimal (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 auf der englischen Wikipedia-Seite <https://en.wikipedia.org/wiki/Hexadecimal#Etymology>.

Für Dualzahlen (also Binärzahlen zur Basis 2) ist das Präfix `0b` oder `0B`. Es sind nur die Ziffern 0 und 1 erlaubt, ein klassischer Fall für 2.



### 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( 0xC0B0L );        // 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
```

### 22.1.4 Auswirkung der Typumwandlung auf die Bit-Muster

Die Typumwandlung 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:

**Listing 22.1** src/main/java/com/tutego/insel/math/TypecastDemo.java, main()

```
int ii = 123456789;      // 0000011101011011_11001101_00010101
int ij = -123456;        // 1111111111111110_00011101_11000000

short si = (short) ii;   // 11001101_00010101
short sj = (short) ij;   // 00011101_11000000

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.

## Typumwandlungen 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 Datenspeicherung, 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 einen `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;           // 1111111111111111168
```

Java erlaubt ohne explizite Anpassung die Konvertierung eines `long` in einen `double` und auch in 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 im Ada-Programm, 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. Das führte zu einer Ausnahme, und 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.

#### 22.1.5 Vorzeichenlos arbeiten

Bis auf `char` sind in Java alle integralen Datentypen vorzeichenbehaftet und kodiert im Zweierkomplement. Bei einem `byte` stehen 8 Bit für die Kodierung eines Werts zur Verfügung, jedoch sind es eigentlich nur 7 Bit, denn über ein Bit erfolgt die Kodierung des Vorzeichens. Der Wertebereich reicht von  $-128$  bis  $+127$ . Über einen Umweg ist es möglich, den vollen Wertebereich auszuschöpfen und so zu tun, als ob Java vorzeichenlose Datentypen hätte.

### byte als vorzeichenlosen Datentyp nutzen

Eine wichtige Eigenschaft der expliziten Typumwandlung bei Ganzzahltypen ist, dass die überschüssigen Bytes einfach abgeschnitten werden. Betrachten wir die Typumwandlung an einem Beispiel:

```
int l = 0xABCD6F;
byte b = (byte) 0xABCD6F;
System.out.println( Integer.toBinaryString( l ) ); // 1010101110011010110111
System.out.println( Integer.toBinaryString( b ) ); // 1101111
```

Liegt eine Zahl im Bereich von 0 bis 255, so kann ein byte sie durch seine 8 Bit grundsätzlich speichern. Java muss jedoch mit der expliziten Typumwandlung gezwungen werden, das Vorzeichen-Bit zu ignorieren. Erst dann entspricht die Zahl 255 acht gesetzten Bit, denn sie mit byte b = 255; zu belegen, funktioniert nicht.

Damit die Weiterverarbeitung gelingt, muss noch eine andere Eigenschaft berücksichtigt werden. Sehen wir uns dazu folgende Ausgabe an:

```
byte b1 = (byte) 255;
byte b2 = -1;
System.out.println( b1 ); // -1
System.out.println( b2 ); // -1
```

Das Bit-Muster ist in beiden Fällen gleich, alle Bits sind gesetzt. Dass die Konsolenausgabe aber negativ ist, hat mit einer anderen Java-Eigenschaft zu tun: Java konvertiert das Byte, das vorzeichenbehaftet ist, bei der Weiterverarbeitung in ein int – der ParameterTyp bei der Methode ist toBinaryString(int) –, und bei dieser Konvertierung wandert das Vorzeichen vom byte zum int weiter. Das folgende Beispiel zeigt das bei der Binärausgabe:

```
byte b = (byte) 255;
int i = 255;
System.out.printf( "%d %s%n", b, Integer.toBinaryString(b) );
// -1 11111111111111111111111111111111
System.out.printf( "%d %s%n", i, Integer.toBinaryString(i) );
// 255 11111111
```

Die Belegung der unteren 8 Bit vom byte b und int i ist identisch. Aber während beim int die oberen 3 Byte wirklich Null sind, füllt Java durch die automatische Anpassung des Vorzeichens bei der Konvertierung von byte nach int im Zweierkomplement auf die oberen 3 Byte mit 255 auf. Soll ohne Vorzeichen weitergerechnet werden, stört das. Diese automatische Anpassung nimmt Java immer vor, wenn mit byte/short gerechnet wird, und nicht nur wie in unserem Beispiel, wenn eine Methode den Datentyp int fordert.

Um bei der Weiterverarbeitung einen Datenwert zwischen 0 und 255 zu bekommen, also das Byte eines `int` vorzeichenlos zu sehen, schneiden wir mit der Und-Verknüpfung die unteren 8 Bit heraus – alle anderen Bits bleiben also ausgenommen:

```
byte b = (byte) 255;
System.out.println( b );           // -1
System.out.println( b & 0xff );    // 255
```

Es bleibt zu bemerken, dass die Und-Verknüpfung zu dem Zieltyp `int` führt.

### Bibliotheksmethoden für vorzeichenlose Behandlung

Immer ein `& 0xff` an einen Ausdruck zu setzen, um die oberen Bytes auszublenden, ist zwar nicht sonderlich aufwändig, aber schön ist das auch nicht. Hübscher sind Methoden wie `toUnsignedInt(byte)`, die mit dem Namen deutlich dokumentieren, was hier eigentlich passt:

Methoden in `Byte`:

- ▶ `static int toUnsignedInt(byte x)`
- ▶ `static long toUnsignedLong(byte x)`

In `Integer`:

- ▶ `static long toUnsignedLong(int x)`
- ▶ `static String toUnsignedString(int i, int radix)`
- ▶ `static String toUnsignedString(int i)`
- ▶ `static int parseUnsignedInt(String s, int radix)`
- ▶ `static int compareUnsigned(int x, int y)`
- ▶ `static int divideUnsigned(int dividend, int divisor)`
- ▶ `static int remainderUnsigned(int dividend, int divisor)`

In `Long`:

- ▶ `String toUnsignedString(long i, int radix)`
- ▶ `static String toUnsignedString(long i)`
- ▶ `static long parseUnsignedLong(String s, int radix)`
- ▶ `static int compareUnsigned(long x, long y)`
- ▶ `static long divideUnsigned(long dividend, long divisor)`
- ▶ `static long remainderUnsigned(long dividend, long divisor)`

In `Short`:

- ▶ `static int toUnsignedInt(short x)`
- ▶ `static long toUnsignedLong(short x)`

Neben den einfachen `toUnsignedXXX(...)`-Methoden in den Wrapper-Klassen gesellen sich Methoden hinzu, die auch die Konvertierung in einen String und das Parse eines Strings ermöglichen. Bei `Integer` und `Long` lassen sich ebenfalls neue Methoden ablesen, die Vergleich, Division und Restwertbildung vorzeichenlos durchführen.

### Konvertierungen von byte in ein char

Mit einer ähnlichen Arbeitsweise können wir auch die Frage lösen, wie sich ein Byte, dessen `int`-Wert 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
```

In der Regel wird so eine explizite Anpassung selten im Code stehen, denn zur Konvertierung von Zeichenkodierungen bietet Java extra Klassen.

#### 22.1.6 Die Verschiebeoperatoren

Unter Java gibt es drei *Verschiebeoperatoren* (engl. *shift-operators*), die die Bits eines Werts um eine gewisse Anzahl an 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 Datenwortes (das Bit-Muster) 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 bleibt erhalten) 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 (bei jedem Schritt mit 2 multipliziert), was 2 hoch s ergibt. Der rechts frei werden-de 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 Operanden erst einmal ein `int` bzw. `long` mit vollem Wertebereich, doch als Verschiebepositionen (Weite) sind bei `int` nur Werte bis 31 sinnvoll und für ein `long` Werte bis 63 Bit, da nur die letzten 5 bzw. 6 Bit berücksichtigt werden. Sonst wird immer um den Wert verschoben, der sich durch das Teilen durch 32 bzw. 64 als Rest ergibt, so-dass `x << 32` und `x << 0` auch gleich sind.

```
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 also unberührt.

**Beispiel**

```
Consumer<Integer> printBinary = value -> {
    String s = String.format( "%32s", Integer.toBinaryString( value ) );
    System.out.println( s.replace( ' ', '0' ) );
};

printBinary.accept( 0b10000000_00000000__00000000_00000000 >> 0 );
printBinary.accept( 0b10000000_00000000__00000000_00000000 >> 1 );
printBinary.accept( 0b10000000_00000000__00000000_00000000 >> 2 );

printBinary.accept( 0b10000000_00000000__00000000_00000000 >>> 0 );
printBinary.accept( 0b10000000_00000000__00000000_00000000 >>> 1 );
printBinary.accept( 0b10000000_00000000__00000000_00000000 >>> 2 );
```

Die Ausgabe ist:

```
10000000000000000000000000000000
11000000000000000000000000000000
11100000000000000000000000000000
10000000000000000000000000000000
```

```
01000000000000000000000000000000
00100000000000000000000000000000
```

### Hinweis

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 Ganzahldatentypen folgt unter Berücksichtigung des immer vorhandenen Vorzeichens bei normalen Rechtsverschiebungen eine vorzeichenrichtige Ganzahldivision 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 mit geschoben.



### Beispiel

Mit den Verschiebeoperatoren lassen sich die einzelnen Bytes eines größeren Datentyps, etwa eines 4 Byte großen `int`, einfach extrahieren:

```
byte b1 = (byte)(v >>> 24),
byte b2 = (byte)(v >>> 16),
byte b3 = (byte)(v >>> 8),
byte b4 = (byte)(v );
```

Bei einer positiven Zahl hat dies keinerlei Auswirkungen, und das Verhalten ist wie beim `>>`-Operator.



### 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.

### 22.1.7 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;
}
```

### 22.1.8 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)
 

Das Ergebnis ist i, wobei alle Bits gelöscht sind mit Ausnahme des höchst-/niedrigstwertigen Bits. Anders gesagt: Liefert einen Wert, wobei nur das höchste (links stehende) bzw. 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 0.
- 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 bzw. nach dem niedrigsten gesetzten Bit.

Als Beispiel Anwendungen der statischen Bit-Methoden der Klasse Long:

Statische Methode der Klasse Long	Methodenergebnis
highestOneBit(0b00011000 )	16
lowestOneBit(0b00011000 )	8
numberOfLeadingZeros( Long.MAX_VALUE )	1
numberOfLeadingZeros( Long.MIN_VALUE )	0
numberOfTrailingZeros( 16 )	4
numberOfTrailingZeros( 3 )	0
bitCount( 8 + 4 + 1 )	3

Tabelle 22.7 Statische Methoden der Klasse Long

Statische Methode der Klasse Long	Methodenergebnis
rotateLeft( 12, 1 )	24
rotateRight( 12, 1 )	6
reverse( 0x00FF00FF00FF000FL )	f000ff0fff00ff00 <sub>16</sub>
reverseBytes( 0xFEDCBA9876543210L )	1032547698badcfe <sub>16</sub>
reverse( Long.MAX_VALUE )	-2

Tabelle 22.7 Statische Methoden der Klasse Long (Forts.)

## 22.2 Fließkomma-Arithmetik 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 Dezimalpunktes als Produkt aus einer Zahl und einer Potenz der Zahl 10 dargestellt wird. Nehmen wir als Beispiel die Lichtgeschwindigkeit mit 299.792,458 Kilometer in der Sekunde:  $299.792,458 = 29.979,2458 \times 10^1 = 2.997,92458 \times 10^2 = 299,792458 \times 10^3 = 29,9792458 \times 10^4 = 2,99792458 \times 10^5$ .

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 eine Länge von 32 Bit und ein `double` eine Länge von 64 Bit. Die Rechenoperationen sind im *IEEE Standard for Floating-Point Arithmetic* definiert.



### 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
```

### 22.2.1 Spezialwerte für Unendlich, Null, NaN

Die Datentypen `double` und `float` können nicht nur »Standardzahlen« speichern, sondern

- ▶ auch eine positive oder negative Null annehmen,
- und zudem definiert Java Sonderwerte für
- ▶ positives und negatives Unendlich (engl. *infinity*) und
- ▶ *NaN*, die Abkürzung für *Not a Number*.

## Positive, negative Null

Es gibt eine positive Null (+0,0) und eine negative Null (-0,0), die etwa beim Unterlauf auftauchen.

### 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 Bit-Maske 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.

## Unendlich

Der Überlauf einer mathematischen Operation führt zu einem positiven oder negativen Unendlich.

### Beispiel

Multiplikation zweier wirklich großer Werte und Division durch null.

```
System.out.println( 1E300 * 1E20 ); // Infinity
System.out.println( -1E300 * 1E20 ); // -Infinity
System.out.println( 1. / 0. ); // Infinity
System.out.println( 1. / -0. ); // -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

Tabelle 22.8 Spezialwerte und ihre Konstanten

Wert für	Float	Double
kleinster Wert	Float.MIN_VALUE	Double.MIN_VALUE
größter Wert	Float.MAX_VALUE	Double.MAX_VALUE

Tabelle 22.8 Spezialwerte und ihre Konstanten (Forts.)

Das Minimum für double-Werte liegt bei etwa  $10^{-324}$  und das Maximum bei etwa  $10^{308}$ . Außerdem deklarieren Double und Float Konstanten für MAX\_EXPONENT/MIN\_EXPONENT.



### Hinweis

Die Anzeige des Über-/Unterlaufs und des undefinierten Ergebnisses gibt es nur bei Fließkomazahlen, nicht aber bei Ganzzahlen.

```
public final class java.lang.Float/Double
    extends Number
    implements Comparable<Float/Double>
```

- static boolean isInfinite(float/double v)  
Liefert true, wenn v entweder POSITIVE\_INFINITY oder NEGATIVE\_INFINITY ist.
- static boolean isFinite(float/double d)  
Liefert true, wenn d eine endliche Zahl ist.

### NaN ist eine Zahl, die keine ist

NaN wird als Fehlerindikator für das Ergebnis von undefinierten Rechenoperationen benutzt, etwa 0/0.



### Beispiel

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 Float/Double deklariert. Die statische Methode isNaN(...) testet, ob eine Zahl NaN ist.

```
public final class java.lang.Float/Double
extends Number
implements Comparable<Float/Double>
```

- static final float/double NaN = 0.0 / 0.0;  
Deklaration von NaN bei Double und Float
- static boolean isNaN(float/double v)  
Liefert true, wenn die übergebene Zahl v NaN ist.
- boolean isNaN()  
Liefert true, wenn das aktuelle Float/Double-Objekt NaN ist.

Die Implementierung von isNaN(float/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 <, <=, >, >= mit 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.

### Beispiel

[zB]

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 Endlosschleife darstellt, mit d als Double.NaN einfach übersprungen:

```
while ( d == d ) { /* Nie ausgeführt */ }
```

Ein NaN-Wert auf eine Ganzzahl angepasst, also etwa (int) Double.NaN, ergibt 0 und keine Ausnahme.<sup>3</sup>

### 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 müs-

---

<sup>3</sup> <https://stackoverflow.com/questions/5876369/why-does-casting-double-nan-to-int-not-throw-an-exception-in-jav>

sen 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*).

### 22.2.2 Standardnotation 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 Vor-kommateil darf durch die Vorzeichen + oder - eingeleitet werden. Auch der Exponent kann positiv oder negativ<sup>4</sup> sein, muss aber eine Ganzzahl sein. Tabelle 22.9 stellt drei Beispiele zusammen:

Standard	Wissenschaftlich
123450.0	1.2345E5
123450.0	1.2345E+5
0.000012345	1.2345E-5

Tabelle 22.9 Notationen der Fließkommazahlen



#### Beispiel

Nutzen der wissenschaftlichen Notation:

```
double x = 3.00e+8;
float y = 3.00E+8F;
```

### 22.2.3 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*  $\times$   $2^{\text{Exponent}}$ , wobei *Vorzeichen* -1 oder +1 sein kann. Die Mantisse  $m$  ist keine Zahl mit beliebigem Wertebereich, sondern normiert mit dem Wertebereich  $1 \leq m < 10$ .

<sup>4</sup> LOGO verwendet für negative Exponenten den Buchstaben N an Stelle des E. In Java steht das E mit einem folgenden unären Plus- oder Minuszeichen.

< 2, also eine Fließkommazahl, die mit 1 beginnt und daher auch *1-plus-Form* heißt.<sup>5</sup> 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 +1.023 (bei double) addiert; nach der Berechnung steht in der Darstellung immer eine ganze Zahl. 127 und 1.023 nennen sich *Bias*.

Das Vorzeichen kostet immer 1 Bit, und die Anzahl der Bits für Exponent und Mantisse richtet sich nach dem Datentyp, siehe Tabelle 22.10.

Datentyp	Anzahl Bits für den Exponenten	Anzahl Bits für die Mantisse
float	8	23
double	11	52

Tabelle 22.10 Anzahl der Bits für Exponent und Mantisse

[zB]

### Beispiel

Das Folgende sind Kodierungen für die Zahl 123.456,789 als float und double. Das # trennt Vorzeichen, Exponent und Mantisse:

```
0#10001111#11100010010000001100101
0#10000001111#111000100100000011001001111101111100111011011001001
```

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.

Zugang zum Bit-Muster liefern die Methoden `long doubleToLongBits(double)` und `int Float.floatToIntBits(float)`. Die Umkehrung ist `double Double.longBitsToDouble(long)` bzw. `float Float.intBitsToFloat(int)`.

<sup>5</sup> Es gibt eine Ausnahme durch denormalisierte Zahlen, aber das spielt für das Verständnis keine Rolle.



### Beispiel

```
double x = 123456.789;
float y = 123456.789f;
out.printf( "%016x%n", Double.doubleToLongBits( x ) ); // 40fe240c9fbe76c9
out.printf( "%08x%n", Float.floatToIntBits( y ) ); // 47f12065
out.println( Long.toBinaryString( Double.doubleToLongBits( x ) ) );
out.println( Integer.toBinaryString( Float.floatToIntBits( y ) ) );
```

## 22.3 Die Eigenschaften der Klasse Math

Die Klasse `java.lang.Math` ist eine typische Utility-Klasse, die nur statische Methoden (bzw. Attribute als Konstanten) deklariert. Mit dem privaten Konstruktor lassen sich keine Exemplare von `Math` erzeugen.

### 22.3.1 Attribute

Die `Math`-Klasse besitzt zwei statische Attribute:

```
class java.lang.Math
```

- static final double E  
die eulersche Zahl<sup>6</sup>  $e = 2,7182818284590452354$
- static final double PI  
die Kreiszahl Pi<sup>7</sup>  $= 3,14159265358979323846$

### 22.3.2 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(double, double)`. Sie ermittelt das Vorzeichen einer Fließkommazahl und überträgt es auf eine andere Zahl.

---

<sup>6</sup> Die irrationale Zahl  $e$  ist nach dem schweizerischen Mathematiker Leonhard Euler (1707–1783) benannt.

<sup>7</sup> Wer noch auf der Suche nach einer völlig unsinnigen Information ist: Die einmilliardste Stelle hinter dem Komma von Pi ist eine Neun.

```
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.

### Hinweis



Es gibt genau einen Wert, auf den Math.abs(int) keine positive Rückgabe liefern kann: -2.147.483.648. Dies ist die kleinste darstellbare int-Zahl (`Integer.MIN_VALUE`), während +2.147.483.648 gar nicht in ein int passt! Die größte darstellbare int-Zahl ist 2.147.483.647 (`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)`

### 22.3.3 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)

Als Vararg sind diese Methoden nicht deklariert, auch gibt es keine Min-Max-Methoden für drei Argumente. Wenn Arrays vorliegen, muss eine andere Lösung gewählt werden, etwa durch `Arrays.stream(value).min().getAsInt()`.

### 22.3.4 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(double)` und `floor(double)`

Die statische Methode `ceil(double)` dient dem 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(double)` rundet auf die nächstniedrigere Ganzzahl ab:

**Listing 22.2** src/main/java/com/tutego/insel/math/Rounding1Demo.java, main()

```
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` bzw. `int`. Ganze Zahlen werden nicht aufgerundet. Wir können `round(...)` als Gegenstück zur Typumwandlung (`long`) `doubleValue` einsetzen:

**Listing 22.3** src/main/java/com/tutego/insel/math/Rounding2Demo.java, main()

```
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
```

### Interna

Die Methoden `Math.round(int)` und `Math.round(long)` sind in Java ausprogrammiert. Sie addieren zum aktuellen Parameter 0,5 und übergeben das Ergebnis der statischen `floor(double)`-Methode.

**Listing 22.4** java.lang.Math, round(double), etwas gekürzt

```
public static long round( double a ) {
    return (long) floor( a + 0.5 );
}
```

### Gerechtes Runden `rint(double)`

`rint(double)` ist mit `round(...)` vergleichbar, nur ist es im Gegensatz zu `round(...)` gerecht, was bedeutet, dass `rint(double)` bei 0,5 in Abhängigkeit davon, ob die benachbarte Zahl ungerade oder gerade ist, auf- oder abrundet:

**Listing 22.5** src/main/java/com/tutego/insel/math/Rounding3Demo.java, main()

```
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.



### Beispiel

Die statische `rint(double)`-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 22.6** 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(double)` 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 Format-String `.2f` bei `printf(...)` rundet.

### 22.3.5 Rest der ganzzahligen Division \*

Neben dem Restwert-Operator `%`, der den Rest der Division berechnet, gibt es auch eine statische Methode `Math.IEEEremainder(double, double)`.

**Listing 22.7** src/main/java/com/tutego/insel/math/IEEEremainder.java, main()

```
double a = 44.0;
double b = 2.2;
System.out.println( a / b ); // 20.0
System.out.println( a % b ); // 2.1999999999999966
System.out.println( Math.IEEEremainder( a, b ) ); // -3.552713678800501E-15
```

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(double, double) 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)));
}
```

### 22.3.6 Division mit Rundung Richtung negativ unendlich, alternativer Restwert \*

Die Ganzzahldivision in Java ist simpel gestrickt. Vereinfacht ausgedrückt: Konvertiere die Ganzzahlen in Fließkommazahlen, führe die Division durch und schneide alles hinter dem Komma ab. So ergeben zum Beispiel  $3 / 2 = 1$  und  $9 / 2 = 4$ . Bei negativem Ergebnis, durch entweder negativen Dividenden oder Divisor, das gleiche Spiel:  $-9 / 2 = -4$  und  $9 / -2 = -4$ . Schauen wir uns einmal die Rundungen an.

Ist das Ergebnis einer Division positiv und mit Nachkommaanteil, so wird das Ergebnis durch das Abschneiden der Nachkommastellen ein wenig kleiner, also abgerundet. Wäre  $3/2$  bei Fließkommazahlen 1,5, ist es bei einer Ganzzahldivision abgerundet 1. Bei negativen Ergebnissen einer Division ist das genau andersherum, denn durch das Abschneiden der Nachkommastellen wird die Zahl etwas größer.  $-3/2$  ist genau genommen  $-1,5$ , aber bei der Ganzzahldivision  $-1$ . Doch  $-1$  ist größer als  $-1,5$ . Java wendet ein Verfahren an, das gegen null rundet.

### Gegen minus unendlich rundernde Methode floorDiv(...)

Zwei Methoden runden bei negativem Ergebnis einer Division nicht gegen null, sondern gegen negativ unendlich, also auch in Richtung der kleineren Zahl, wie es bei den positiven Ergebnissen ist.

```
class java.lang.Math
class java.lang.StrictMath
```

- static int floorDiv(int x, int y)
- static long floorDiv(long x, long y)
- static long floorDiv(long x, int y)

Ganz praktisch heißt das:  $4/2 = \text{Math.floorDiv}(4, 3) = 1$ , aber wo  $-4 / 3 = -1$  ergibt, liefert  $\text{Math.floorDiv}(-4, 3) = -2$ .

### Gegen minus unendlich rundernde Methode floorMod(...)

Die Division taucht indirekt auch bei der Berechnung des Restwerts auf. Zur Erinnerung: Der Zusammenhang zwischen Division  $a/b$  und Restwert  $a \% b$  ( $a$  heißt *Dividend*,  $b$  *Divisor*) ist  $(\text{int})(a/b) * b + (a \% b) = a$ . In der Gleichung gibt es eine Division, doch da es mit  $a / b$  und  $\text{floorDiv}(a, b)$  zwei Arten von Divisionen gibt, muss es folglich auch zwei Arten von Restwertbildung geben, die sich dann unterscheiden, wenn die Vorzeichen unterschiedlich sind. Neben  $a \% b$  gibt es daher die Bibliotheksmethode  $\text{floorMod}(a, b)$ , und der Zusammenhang zwischen  $\text{floorMod}(\dots)$  und  $\text{floorDiv}(\dots)$  ist:  $\text{floorDiv}(a, b) * b + \text{floorMod}(a, b) == b$ . Nach einer Umformung der Gleichung folgt  $\text{floorMod}(a, b) = a - (\text{floorDiv}(a, b) * b)$ . Das Ergebnis ist im Bereich  $-\text{abs}(b)$  und  $\text{abs}(b)$ , und das Vorzeichen des Ergebnisses bestimmt der Divisor  $b$  (beim  $\%$ -Operator ist es der Dividend  $a$ ).

Die Javadoc zeigt ein Beispiel mit den Werten 4 und 3 bei unterschiedlichen Vorzeichen:

floorMod(..)-Methode	%-Operator
$\text{floorMod}(+4, +3) == +1$	$+4 \% +3 == +1$
$\text{floorMod}(+4, -3) == -2$	$+4 \% -3 == +1$
$\text{floorMod}(-4, +3) == +2$	$-4 \% +3 == -1$
$\text{floorMod}(-4, -3) == -1$	$-4 \% -3 == -1$

Tabelle 22.11 Ergebnis unterschiedlicher Restwertbildung

Sind die Vorzeichen für  $a$  und  $b$  identisch, so ist auch das Ergebnis von  $\text{floorMod}(a, b)$  und  $a \% b$  gleich. Die Beispiele in der Tabelle machen auch den Unterschied im Ergebnisvorzeichen deutlich, das einmal vom Dividenden (%) und einmal vom Divisor ( $\text{floorMod}(\dots)$ ) kommt. Die

komplett anderen Ereignisse (vom Vorzeichen einmal abgesehen) bei den Paaren (+4, -3) und (-4, +3) ergeben sich ganz einfach aus unterschiedlichen Ergebnissen der Division:

$$\begin{aligned}\text{floorMod}(+4, -3) &= +4 - (\text{floorDiv}(+4, -3) * -3) = +4 - (-2 * -3) = +4 - +6 = \\ -2 + 4 \% -3 &= +4 - (+4/-3 * -3) = +4 - (-1 * -3) = +4 - 3 = +1\end{aligned}$$

```
class java.lang.Math
class java.lang.StrictMath
```

- static int floorMod(int x, int y)
- static long floorMod(long x, long y)
- static long floorMod(long x, int y)

Die Implementierung ist relativ einfach mit  $x - \text{floorDiv}(x, y) * y$  angegeben.

### 22.3.7 Multiply-Accumulate

Die mathematische Operation  $a \times b + c$  kommt in Berechnungen oft vor, sodass Prozessoren heute diese Berechnung optimiert – das heißt schneller und mit besserer Genauigkeit – durchführen können. Bei `Math` und `StrictMath` gibt es die Methode `fma(...)` für die »fused multiply-accumulate«-Funktion aus dem IEEE-754-Standard.

```
class java.lang.Math
class java.lang.StrictMath
```

- static double fma(double a, double b, double c)
  - static float fma(float a, float b, float c)
- Führt  $a \times b + c$  mit dem `RoundingMode.HALF_EVEN` durch. Gegenüber einem einfachen  $a * b + c$  berücksichtigt `fma(...)` die Besonderheiten unendlich und NaN.

### 22.3.8 Wurzel- und Exponentialmethoden

Die `Math`-Klasse bietet außerdem 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 Kubikwurzel 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  $e = 2,71828\dots$ ), 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 Ganzahlwerten 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 in ein `long` 0 ergibt.

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  $\frac{1}{2}$  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.<sup>8</sup>

### 22.3.9 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(double)` berechnet, ist der natürliche Logarithmus

---

<sup>8</sup> Hier schreiben die Autoren R. Graham, D. Knuth, O. Patashnik des Buchs *Concrete Mathematics*, Addison-Wesley, 1994, ISBN 0-201-55802-5: »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.«

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(double)` für den briggschen Logarithmus *lg*, nicht aber für den binären Logarithmus *lb*, der weiterhin nachgebildet werden muss. Allgemein gilt folgende Umrechnung:  $\log_b(x) = \log_a(x) \div \log_a(b)$ .

### Beispiel

[zB]

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 );
}
```

Da `Math.log(2)` konstant ist, sollte dieser Wert aus Performance-Gründen in einer Konstanten gehalten werden: `static final double log2 = Math.log(2.0)`. Das gilt übrigens für andere konstante Ergebnisse auch, denn ist es nicht davon auszugehen, dass die JVM das Ergebnis des Methodenaufrufs einsetzt, statt die Methode aufzurufen. Für die JVM handelt es sich erst einmal um irgendeine Methode, deren Semantik – wenn ein konstanter Wert reinkommt, kommt auch ein konstanter Wert heraus – erst lernen muss. C-Compiler sind hier deutlich weiter.<sup>9</sup>

```
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  $\log(x) + 1$ .

### 22.3.10 Winkelmethoden \*

Die Math-Klasse stellt einige winkelbezogene Methoden und deren Umkehrungen zur Verfügung. Im Gegensatz zur bekannten Schulmathematik werden die Winkel für `sin(double)`, `cos(double)`, `tan(double)` im Bogenmaß ( $2\pi$  entspricht einem Vollkreis) und nicht im Gradmaß (360 Grad entspricht einem Vollkreis) übergeben.

<sup>9</sup> Wer Lust zu spielen hat, besucht <https://godbolt.org/> und setzt im Editor Folgendes ein: `#include <math.h>#include <stdio.h>int main() { printf("%f", log(2.0)); }`

**class java.lang.Math**

- static double sin(double x)
  - static double cos(double x)
  - static double tan(double x)
- Liefert den Sinus/Kosinus/Tangens von x.

**Arcus-Methoden**

Die Arcus-Methoden realisieren die Umkehrfunktionen zu den trigonometrischen Methoden. Das Argument ist kein Winkel, sondern zum Beispiel bei asin(double) der Sinuswert zwischen -1 und 1. Das Ergebnis ist ein Winkel im Bogenmaß, etwa zwischen  $-\pi/2$  und  $\pi/2$ .

**class java.lang.Math**

- static double asin(double x)
  - static double acos(double x)
  - static double atan(double x)
- Liefert den Arcus-Sinus, Arcus-Kosinus bzw. Arcus-Tangens von x.
- static double atan2(double x, double y)
- Liefert bei der Konvertierung von Rechteckkoordinaten in Polarkoordinaten den Winkel *theta*, 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(double), tanh(double) und cosh(double).

**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.

### 22.3.11 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 Typumwandlung in `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 ganz-zahlige Zufallszahlen zwischen `min` (inklusiv) und `max` (inklusiv) zu erhalten, schreiben wir:

**Listing 22.8** src/main/java/com/tutego/insel/math/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)`.

Die Implementierung steckt nicht in der Mathe-Klasse selbst, sondern in `Random`, was wir uns in [Abschnitt 22.5, »Zufallszahlen: Random, SecureRandom, SplittableRandom«](#), genauer ansehen.

## 22.4 Genauigkeit, Wertebereich eines Typs und Überlaufkontrolle \*

### 22.4.1 Der größte und der kleinste Wert

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  $-2.147.483.648$  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 22.9** src/main/java/com/tutego/insel/math/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
```

### 22.4.2 Überlauf und alles ganz exakt

Bei einigen mathematischen Fragestellungen muss sich feststellen lassen, 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 standardmäßig nicht von Java angezeigt; weder der Compiler noch die Laufzeitumgebung melden dieses Problem. Es gibt auch keine Ausnahme, Java hat keine eingebaute Überlaufkontrolle in der Sprache.



#### Beispiel

Mathematisch gilt  $a \times a \div a = a$ , also zum Beispiel  $100.000 \times 100.000 \div 100.000 = 100.000$ . In Java ist das anders, da wir bei  $100.000 \times 100.000$  einen Überlauf im int haben.

```
System.out.println( 100_000 * 100_000 / 100_000 ); // 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( 100_000L * 100_000 / 100_000 ); // 100000
```

### Multiplizieren von int-Ganzzahlen

Der \*-Operator führt bei int keine Anpassung an den Datentypen durch, sodass die Multiplikation von zwei ints wiederum int liefert. Doch das Produkt kann schnell aus dem Werte-

bereich laufen, sodass es zum Überlauf kommt. Selbst wenn das Produkt in eine long-Variablen geschrieben wird, erfolgt die Konvertierung von int in long erst nach der Multiplikation:

```
int i = Integer.MAX_VALUE * Integer.MAX_VALUE;
long l = Integer.MAX_VALUE * Integer.MAX_VALUE;
System.out.println( i );      // 1
System.out.println( l );      // 1
```

Sollen zwei ints ohne Überlauf multipliziert werden, ist einer der beiden Faktoren auf long anzupassen, damit es zum korrekten Ergebnis 4611686014132420609 führt.

```
System.out.println( Integer.MAX_VALUE * (long) Integer.MAX_VALUE );
System.out.println( (long) Integer.MAX_VALUE * Integer.MAX_VALUE );
```

Da diese Typumwandlung schnell vergessen werden kann und nicht besonders explizit ist, bieten die Klassen `Math` und `StrictMath` die statische Methode `long multiplyFull(int x, int y)`, die für uns über `(long)x * (long)y` die Typumwandlung vornimmt.

### Multiplizieren von long-Ganzzahlen

Es gibt keinen primitiven Datentyp, der mehr als 64 Bit (8 Byte) hat, sodass das Ergebnis von `long * long` mit seinen 128 Bit nur in ein `BigInteger` komplett passt. Allerdings erlaubt eine Methode, die oberen 64 Bit einer `long`-Multiplikation getrennt zu erfragen, und zwar mit der Methode `multiplyHigh(long x, long y)` in `Math` und `StrictMath` – wobei `StrictMath` nur auf `Math` leitet.

#### Beispiel

[zB]

```
BigInteger v = BigInteger.valueOf( Long.MAX_VALUE )
                .multiply( BigInteger.valueOf( Long.MAX_VALUE ) );
System.out.println( v ); // 85070591730234615847396907784232501249

long lowLong = Long.MAX_VALUE * Long.MAX_VALUE;
long highLong = Math.multiplyHigh( Long.MAX_VALUE, Long.MAX_VALUE );
BigInteger w = BigInteger.valueOf( highLong )
                .shiftLeft( 64 )
                .add( BigInteger.valueOf( lowLong ) );
System.out.println( w ); // 85070591730234615847396907784232501249
```

## Überlauf erkennen

Bestimmte Methoden in Math und StrictMath ermöglichen eine Überlauferkennung:

```
class java.lang.Math
class java.lang.StrictMath
```

- static int addExact(int x, int y)
- static long addExact(long x, long y)
- static int subtractExact(int x, int y)
- static long subtractExact(long x, long y)
- static int multiplyExact(int x, int y)
- static int multiplyExact(long x, int y)
- static long multiplyExact(long x, long y)
- static int toIntExact(long value)

Alle Methoden werfen eine `ArithmaticException`, falls die Operation nicht durchführbar ist, die letzte zum Beispiel, wenn `(int)value != value` ist. Leider deklariert Java keine Unterklassen wie `UnderflowException` oder `OverflowException`, und Java meldet nur alles vom Typ `ArithmaticException` mit der Fehlermeldung »xxxx overflow«, auch wenn es eigentlich ein Unterlauf ist.

[zB]

### Beispiel

Von der kleinsten Ganzzahl mit `subtractExact(...)` eins abzuziehen, führt zur Ausnahme:

```
subtractExact( Integer.MIN_VALUE, 1 ); // ☠ ArithmaticException
```

In Math, aber nicht in StrictMath, gibt es außerdem:

```
class java.lang.Math
```

- static int incrementExact(int a)
- static long incrementExact(long a)
- static int decrementExact(int a)
- static long decrementExact(long a)
- static int negateExact(int a)
- static long negateExact(long a)

Kann wegen des Wertebereichs die Operation nicht durchgeführt werden, folgt wieder eine `ArithmaticException`.

### Vergleich mit C#

C# verhält sich genauso wie Java und reagiert standardmäßig nicht auf einen Überlauf. Es gibt jedoch spezielle checked-Blöcke, die eine OverflowException melden, wenn es bei arithmetischen Grundoperationen zu einem Überlauf kommt. Folgendes löst diese Ausnahme aus: `checked { int val = int.MaxValue; val++; }`. Solche checked-Blöcke gibt es in Java nicht – wer diese besondere Überlaufkontrolle braucht, muss die Methoden nutzen und ein `val++` dann auch umschreiben zu `Math.addExact(val, 1)` bzw. `Math.incrementExact(val)`.

### 22.4.3 Was bitte macht eine 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)` bzw. `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(float|double)/nextDown(float|double)`, die auf `nextAfter(...)` zurückgreifen.

### Beispiel

Was kommt nach und vor 1?

```
System.out.printf( "%.16f%n", Math.nextUp( 1 ) );
System.out.printf( "%.16f%n", Math.nextDown( 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:

```
1,000000119209289
0,999999403953552
1,0000001192092896
0,999999403953552
```

[zB]

`nextUp(d)` ist eine Abkürzung für `nextAfter(d, Double.POSITIVE_INFINITY)` und `nextDown(d)` eine Abkürzung für `nextAfter(d, Double.NEGATIVE_INFINITY)`. 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.

Methode	Rückgabe der ulp
Math.ulp( 0.00001 )	0,00000000000000000000000000000001694065895
Math.ulp( -1 )	0,00000011920928955078125
Math.ulp( 1 )	0,00000011920928955078125
Math.ulp( 2 )	0,0000002384185791015625
Math.ulp( 10E30 )	1125899906842624

**Tabelle 22.12** Beispiel: Je größer die Zahlen werden, desto größer werden die Sprünge.

## Ein Quantum Ungenauigkeit

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(double)` eine mögliche Ungenauigkeit von 1 ulp haben, `atan2(double, double)` von maximal 2 ulp und `sinh(double), cosh(double), tanh(double)` 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)`

## 22.5 Zufallszahlen: Random, SecureRandom, SplittableRandom

Die Math-Klasse bietet mit `random()` eine einfache Methode für Zufallszahlen. Intern basiert sie jedoch auf einer anderen Klasse, die wir auch direkt nutzen können, womit wir die Möglichkeit haben, nicht nur Zufallszahlen zwischen 0 und 1 zu erfragen.

Die üblichen Zufallszahlen von Java sind so genannte *Pseudozufallszahlen*, weil sie von einem mathematischen Algorithmus erzeugt werden. Gute »Zufallswerte« wiederholen sich erst nach sehr langen Sequenzen, haben keinen offensichtlichen Zusammenhang und sind

schnell generiert. Perfekte Zufallszahlen wären nie vorhersehbar, die Wahrscheinlichkeit für jede Zahl wäre immer gleich – unabhängig von den vorangehenden Werten –, und die Sequenzen würden sich nie wiederholen.

### 22.5.1 Die Klasse Random

Die Klasse Random im java.util-Paket implementiert einen Generator für Pseudozufallszahlen. Im Gegensatz zu Math.random() bietet Random keine statischen Funktionen, sondern eine Reihe von nextXXX(...)-Methoden. Die statische Funktion Math.random() nutzt intern ein Random-Objekt.

### 22.5.2 Random-Objekte mit dem Samen aufbauen

Jedes Random-Objekt benötigt für die Berechnung einen Startwert. 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.)

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, sprich, ist der Samen der gleiche, ist natürlich auch die Folge der Zufallszahlen immer gleich; für Tests ist das gar nicht so schlecht. Der parameterlose 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.

### 22.5.3 Einzelne 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()`  
Liefert die nächste Pseudo-Zufallszahl aus dem gesamten Wertebereich, also zwischen `Integer.MIN_VALUE` und `Integer.MAX_VALUE` bzw. `Long.MIN_VALUE` und `Long.MAX_VALUE`.
- ▶ `float nextFloat()`  
Liefert die nächste Pseudo-Zufallszahl zwischen 0,0 und 1,0.
- ▶ `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 gleich eine Reihe von Zufallszahlen erzeugen lässt. Dies ist die Methode `nextBytes(byte[])`. Der Parameter ist ein Byte-Array, und dieses wird komplett mit Zufallszahlen gefüllt.

Hinter allen Methoden zur Erzeugung von Zufallszahlen steckt die Methode `next(int bits)`. Sie ist in `Random` implementiert, aber durch die Sichtbarkeit `protected` nur von einer erbbaren Klasse sichtbar. Unterklassen sind möglich, denn `Random` ist eine ganz normale, öffentliche, nichtfinale Klasse.

### 22.5.4 Pseudo-Zufallszahlen in der Normalverteilung \*

Über eine spezielle Methode können wir Zufallszahlen erhalten, die einer Normalverteilung genügen: `nextGaussian()`. Diese Methode arbeitet intern nach der so genannten Polar-Methode und erzeugt aus zwei unabhängigen Pseudo-Zufallszahlen zwei normalverteilte 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.

### 22.5.5 Strom von Zufallszahlen generieren \*

Sind mehrere Zufallszahlen nötig, ist eine Schleife mit wiederholten Aufrufen von `nextXXX()` nicht nötig; stattdessen gibt es in `Random` zwei Sorten von Methoden, die ein Bündel von Zufallszahlen liefern.

Als Erstes:

- ▶ `void nextBytes(byte[] bytes)`  
Füllt das Array `bytes` mit Zufallsbytes auf.

Weitere Methoden liefern einen Stream von Zufallszahlen:

- ▶ `IntStream ints(...)`
- ▶ `LongStream longs(...)`
- ▶ `DoubleStream doubles(...)`

#### Beispiel

[zB]

Liefere zehn zufällige Zahlen, die vermutlich Primzahlen sind:

```
LongStream stream = new Random().longs()
    .filter( v -> BigInteger.valueOf( v ).isProbablePrime( 5 ) );
stream.limit( 10 ).forEach( System.out::println );
```

Die Methoden `ints(...)`, `longs(...)` und `doubles(...)` gibt es in vier Spielarten, siehe [Tabelle 22.13](#).

Parametrisierung	Erklärung
<code>IntStream ints()</code>	Liefert einen unendlichen Strom von Zufallszahlen im kompletten Wertebereich der Primitiven.
<code>LongStream longs()</code>	
<code>DoubleStream doubles()</code>	
<code>ints(long streamSize)</code>	Liefert einen Strom mit <code>streamSize</code> Zufallszahlen.
<code>longs(long streamSize)</code>	
<code>doubles(long streamSize)</code>	

**Tabelle 22.13** Stream-Methoden der Random-Klasse

Parametrisierung	Erklärung
<code>ints(int randomNumberOrigin, int randomNumberBound)</code>	Liefert einen unendlichen Strom von Zufallszahlen mit Werten im Bereich von <code>randomNumberOrigin</code> (inklusiv) bis <code>randomNumberBound</code> (exklusiv).
<code>longs(long randomNumberOrigin, long randomNumberBound)</code>	
<code>doubles(double randomNumberOrigin, double randomNumberBound)</code>	
<code>ints(long streamSize, int randomNumberOrigin, int randomNumberBound)</code>	Liefert einen Strom mit <code>streamSize</code> Zufallszahlen mit Werten im Bereich von <code>randomNumberOrigin</code> (inklusiv) bis <code>randomNumberBound</code> (exklusiv).
<code>longs(long streamSize, long randomNumberOrigin, long randomNumberBound)</code>	
<code>doubles(long streamSize, double randomNumberOrigin, double randomNumberBound)</code>	

Tabelle 22.13 Stream-Methoden der Random-Klasse (Forts.)



### Beispiel

Gib fünf Fließkomma-Zufallszahlen im Bereich von 10 bis 20 aus:

```
new Random().doubles(5, 10, 20).forEach( System.out::println );
```

### 22.5.6 Die Klasse SecureRandom \*

Die Random-Klasse steht in der Zwickmühle, auf der einen Seite gute und zufällige Zahlen zu erzeugen, auf der anderen Seite aber auch schnell zu sein. Kryptografisch ordentliche Zahlen erzeugt Random nicht, dafür müsste die Implementierung mehr Aufwand treiben – was mehr Zeit kostet –, und so gute Zufallszahlen sind im Alltag auch gar nicht nötig.

Kryptografisch bessere Zufallszahlen liefert `java.security.SecureRandom`, eine Unterklasse von `Random`. Als Unterklasse bietet sie natürlich den gleichen Satz an `nextXXX(...)`-Methoden, nur ist eben die Qualität der Zufallszahlen besser. Das liegt nicht an `SecureRandom` selbst, denn die Klasse realisiert keine Algorithmen, sondern an den referenzierten austauschbaren Zufallszahlen-Providern. Eine Implementierung liefert `SecureRandom.getInstanceStrong()` oder

auch der parameterlose Konstruktor, dann muss das `SecureRandom`-Objekt aber nicht zwingend »stark« sein, also höchstens kryptografischen Standards entsprechen. Den Unterschied gibt es, denn `new SecureRandom().getAlgorithm().toString()` liefert `SHA1PRNG`, und `SecureRandom.getInstanceStrong().getAlgorithm().toString()` gibt `Windows-PRNG`.

### 22.5.7 `SplittableRandom` \*

Die Klasse `SplittableRandom` aus dem `java.util`-Paket hat die Aufgabe, Folgen guter Zufallszahlen zu liefern. (Auch wenn die Klasse `SplittableRandom` heißt, hat sie mit einem `java.util.Splitter` nichts zu tun.) Während bei `Random` eher die einzelne Zufallszahl im Mittelpunkt steht, rückt `SplittableRandom` Folgen von Zufallszahlen in den Mittelpunkt, die insbesondere den Dieharder-Test<sup>10</sup> bestehen.

Die Methoden von `SplittableRandom` drehen sich daher auch um Ströme von Zufallszahlen, die als `IntStream`, `LongStream` und `DoubleStream` geliefert werden. Zudem gibt es auch die auf `Random` bekannten `nextXXX()`-Methoden und eine Methode `split()`, die ein neues `SplittableRandom` liefert, sodass zwei parallele Threads weiterhin unabhängig gute Zufallszahlen bekommen.

## 22.6 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.

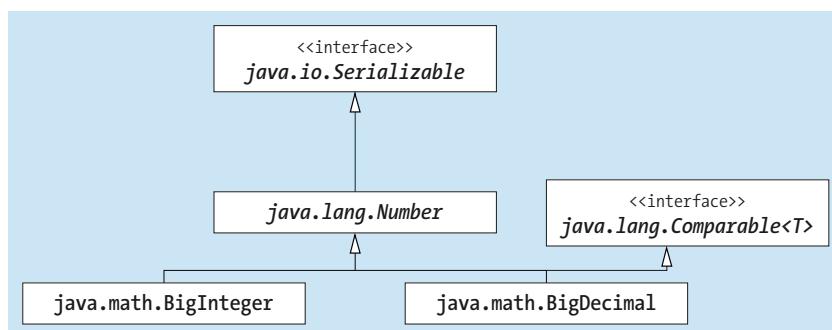


Abbildung 22.1 Vererbungsbeziehung von `BigInteger` und `BigDecimal`

<sup>10</sup> <http://www.phy.duke.edu/~rgb/General/dieharder.php>

### 22.6.1 Die Klasse BigInteger

Mit der Klasse `BigInteger` ist es uns möglich, beliebig genaue Ganzzahlen anzulegen, zu verwalten und damit zu rechnen. Die `BigInteger`-Objekte werden dabei intern 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, Bit-Manipulation und Weiteres.



#### Beispiel

Das Quadrat einer nur aus Einsen bestehenden Zahl gibt ein interessantes Ergebnis, nämlich eine Zahl, die erstens ein Palindrom<sup>11</sup> ist und zweitens (bis zur Mitte) hochzählt in der Art 1234...

**Listing 22.10** src/main/java/com/tutego/insel/math/BigIntegerDemo.java, main()

```
BigInteger ones = BigInteger.ONE;
for ( int i = 0; i < 20; i++ ) {
    System.out.println( ones + "² = " + ones.pow( 2 ) );
    ones = ones.multiply( BigInteger.TEN ).add( BigInteger.ONE );
}
```

Das Ergebnis der Schleife ist:

```
1² = 1
11² = 121
111² = 12321
1111² = 1234321
11111² = 123454321
...
11111111111111111111² = 123456790123456790120987654320987654321
```

`BigInteger`-Objekte sind immer `immutable` wie `Strings`, das Gleiche gilt im Übrigen auch für `BigDecimal`. Die mathematischen Operationen auf den Objekten liefern also immer neue Objekte.

#### Konstanten für einige BigInteger-Objekte

In unserem kleinen Beispiel haben wir auf vorhandene `BigInteger`-Objekte zurückgegriffen und mit ihrer Hilfe neue Objekte gebaut. Insgesamt bietet die Klasse drei Konstanten für die Werte 0, 1 und 10:

<sup>11</sup> Wort, das sich von vorne wie von hinten gleich liest

```
class java.math.BigInteger
extends Number
implements Comparable<BigInteger>
```

- static final BigInteger ZERO
- static final BigInteger ONE
- static final BigInteger TEN

Im Prinzip lässt sich jedes BigInteger durch mathematische Operationen von diesen Konstanten ableiten, praktisch ist das aber nicht, daher bietet die Klasse auch Konstruktoren und Fabrikmethoden an.

### BigInteger-Objekte erzeugen

Zur Erzeugung stehen uns verschiedene Konstruktoren und eine Fabrikmethode zur Verfügung. Einen parameterlosen Konstruktor gibt es nicht, denn ein leeres immutable BigInteger-Objekt ist wenig sinnvoll.

```
class java.math.BigInteger
extends Number
implements Comparable<BigInteger>
```

- static BigInteger valueOf(long val)  
Statische Fabrikmethode, die aus einem long ein BigInteger konstruiert.
- 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 in ein 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 bzw. a–z als zusätzliche »Ziffern« verwendet.
- BigInteger(byte[] val)  
Ein Byte-Array mit einer Zweierkomplement-Repräsentation einer BigInteger-Zahl im Big-Endian-Format (Array-Element mit Index 0, enthält die höchstwertigen Bits) initialisiert das neue BigInteger-Objekt.
- BigInteger(byte[] val, int off, int len)  
Wie BigInteger(byte[] val), nur ein Teilstück wird betrachtet.
- BigInteger(int signum, byte[] magnitude)  
Erzeugt aus einem Big-Endian-Betrag bzw. 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 signum, byte[] magnitude, int off, int len)`  
Wie vorangehender Konstruktor, nur mit Teilstück.

Neben Konstruktoren, die das Objekt mit Werten aus einem Byte-Array 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.

- `BigInteger(int numbits, Random rnd)`  
Liefert eine Zufallszahl aus dem Wertebereich 0 bis  $2^{\text{numBits}-1}$ . Alle Werte sind gleich wahrscheinlich.
- `BigInteger(int bitLength, int certainty, Random rnd)`  
Erzeugt eine `BigInteger`-Zahl mit der Bit-Länge `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  $1/(2^{\text{certainty}})$  handelt es sich bei der erzeugten Zahl doch um keine Primzahl. Je größer `certainty` (und je unwahrscheinlicher ein Fehlurteil) ist, desto mehr Zeit nimmt sich der Konstruktor.

Bei falschen Zeichenfolgen löst der Konstruktor mit String-Parameter eine `NumberFormatException` aus.



### 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. In einem zweiten Schritt soll eine Ausgabe erfolgen:

```
String s = "11011101 10101010 0010101 00010101".replace( " ", "" );
byte[] bytes = new BigInteger( s, 2 ).toByteArray(); // [158,261,69,69]
System.out.println( Long.toHexString( ByteBuffer.wrap( bytes ).getInt() ) );
// 1101110110101010001010100010101
System.out.printf( "%04X", new BigInteger( 1, bytes ) ); // 6ED51515
```

Die erste Ausgabe benutzt mit `ByteBuffer.wrap(bytes).getInt()` einen Trick, um die Bytes eines (kleinen) Arrays zu einem `int` zusammenzubauen. Der Aufruf funktioniert aber nur dann, wenn das Array für ein `int` genau 4 Byte groß ist – neben `getInt()` gibt es noch `getShort()` und `getLong()` für Arrays der Größe 2 und 8. Für unser Beispiel ist das in Ordnung, weil wir wirklich 4 Bytes haben. Die zweite Ausgabe zeigt noch einen anderen Anwendungsfall für `BigInteger`, nämlich eine Hex-Ausgabe zu produzieren; das Byte-Array kann eine beliebige Länge haben. Dass es `new BigInteger(1, bytes)` statt `new BigInteger(bytes)` heißt, ist wichtig, denn wir wollen für die Hexadezimalausgabe kein Vorzeichen, falls das erste Bit im Byte-Array gesetzt ist und üblicherweise eine negative Zahl anzeigt. Der Format-String `%4X` füllt die Ausgabe links mit Nullen auf, falls es nötig ist.

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(...)`.

### Interne Repräsentation \*

Die Implementierung stellt ein BigInteger-Objekt intern wie auch die primitiven Datentypen byte, short, int, long im Zweierkomplement dar. Auch die weiteren Operationen entsprechen den Ganzzahl-Operationen der primitiven Datentypen, wie etwa die Division durch null, die eine `ArithmaticException` auslöst.

Intern vergrößert ein BigInteger, wenn nötig, den Wertebereich, sodass einige Operationen nicht übertragbar sind. So kann der Verschiebeoperator `>>>` 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 Bit-Länge wird der kleinere Wert dem größeren durch Replikation (Wiederholung) des Vorzeichen-Bits angepasst. Über spezielle Bit-Operatoren 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 Bit-Operationen lässt sich das Vorzeichen einer Zahl nicht verändern; gegebenenfalls wird vor der Zahl ein neues Vorzeichen-Bit mit dem ursprünglichen Wert ergänzt.

### 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 Array mit zwei BigInteger-Objekten. Im Array, 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 das aktuelle BigInteger invertiert (entspricht  $1/\text{this}$ ) 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, dessen Bits negiert sind.
- `BigInteger pow(int exponent)`  
Bildet `this` hoch exponent.
- `int signum()`  
Liefert das Vorzeichen des eigenen BigInteger-Objekts.



### Beispiel

Die Wrapper-Klassen Integer oder Long bieten keine Methoden zur Bestimmung eines größten gemeinsamen Teilers oder zum Test auf eine Primzahl. Hier lohnt es, BigInteger einzusetzen. Beispiel: Was ist der ggT von 855 und 99? Antworten wird `BigInteger.valueOf( 855 ).gcd( BigInteger.valueOf( 99 ) ).longValueExact()` mit 9.

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 gelöschttem/gekipptem/gesetztem *n*-ten Bit.
- `BigInteger shiftLeft(int n)`
- `BigInteger shiftRight(int n)`  
Schiebt die Bits um *n* Stellen nach links/rechts.
- `int getLowestSetBit()`  
Liefert die Position des 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*.

Des Weiteren gibt es Extraktions- und Konvertierungsmethoden:

- `double doubleValue()`
- `float floatValue()`
- `int intValue()`
- `long longValue()`  
Konvertiert das BigInteger in einen double/float/int/long. Es handelt sich um implementierte Methoden der abstrakten Oberklasse Number. Falls die Wertebereiche nicht passen,

werden sie passend gemacht, Konvertierungsfehler aufgrund von fehlender Genauigkeit fallen fälschlicherweise nicht auf.<sup>12</sup>

- long longValueExact()
- int intValueExact()
- short shortValueExact()
- byte byteValueExact()

Konvertiert den Wert des BigInteger in den gewünschten primitiven Datentyp. Reicht die Genauigkeit nicht aus, folgt eine ArithmeticException. Die xxxValueExact()-Methoden folgen der Bauart der xxxExact()-Methoden aus Math/StrictMath, die bei Überschreitung des Wertebereichs bei Operationen (Addition, Multiplikation ...) eine ArithmeticException auslösen.

- byte[] toByteArray()  
Liefert ein Byte-Array mit dem BigInteger als Zweierkomplement.
- String toString()
- String toString(int radix)  
Liefert die String-Präsentation von diesem BigInteger zur Basis 10 bzw. zu einer beliebigen Basis.



### Beispiel

byteValue() und byteValueExact() im Vergleich:

```
System.out.println( BigInteger.TEN.pow( 3 ).byteValue() );
BigInteger.TEN.pow( 3 ).byteValueExact();
```

Während Ersteres die Ausgabe »–24« ergibt, folgt bei Letzterem eine »java.lang.Arithmeti-Exception: BigInteger out of int range«, denn natürlich passt 1.000 nicht in ein Byte.

Die letzte Gruppe bilden Vergleichsmethoden:

- BigInteger max(BigInteger val)
- BigInteger min(BigInteger val)  
Liefert das größere/kleinere der BigInteger-Objekte als Rückgabe.
- boolean equals(Object x)  
Vergleicht, ob x und das eigene BigInteger-Objekt den gleichen Wert annehmen.  
Überschreibt die Methode aus Object.

---

<sup>12</sup> Bei der Konvertierung gibt bei machen Werten Probleme mit der Performance. So steht in der Implementierung bei doubleValue() an return Double.parseDouble(this.toString()); »Somewhat inefficient, but guaranteed to work.«

- int compareTo(BigInteger o)

Da die Klasse BigInteger die Schnittstelle java.lang.Comparable implementiert, lässt sich jedes BigInteger-Objekt mit einem anderen BigInteger vergleichen. Das ist praktisch, denn BigInteger-Objekte lassen sich so einfach in sortierbare Datenstrukturen legen.

## 22.6.2 Beispiel: ganz lange Fakultäten mit BigInteger

Unser Beispielprogramm soll die Fakultät einer natürlichen Zahl berechnen; die Zahl muss positiv sein:

**Listing 22.11** src/main/java/com/tutego/insel/math/Factorial.java

```
import java.math.*;

public class Factorial {
    static BigInteger factorial( int n ) {
        BigInteger result = BigInteger.ONE;

        if ( n > 1 )
            for ( int i = 1; i <= n; i++ )
                result = result.multiply( BigInteger.valueOf(i) );

        return result;
    }

    public static 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 22.12** src/main/java/com/tutego/insel/math/Factorial.java, factorial2()

```
public static BigInteger factorial2( int i ) {
    if ( i <= 1 )
        return BigInteger.ONE;
```

```
    return BigInteger.valueOf( i ).multiply( factorial2( i - 1 ) );
}
```

### 22.6.3 Große Fließkommazahlen mit BigDecimal

Während sich BigInteger um die beliebig genauen Ganzzahlen kümmert, übernimmt BigDecimal die Fließkommazahlen. Wieder sind die Objekte immutable, und es gibt drei Konstanten für die Zahlen BigDecimal.ZERO, BigDecimal.ONE und BigDecimal.TEN.

## BigDecimal aufbauen

`BigDecimal` bietet parametrisierte Konstruktoren für unterschiedliche Typen an, unter anderem `long`, `double` und `String`, und initialisiert damit ein `BigDecimal`-Objekt.



## Hinweis

Bei double ist Obacht geboten, denn während der Compiler bei

das Literal auf den für double gültigen Bereich bringt (1), ist nur

präzise. Das gleiche Phänomen ist bei System.out.println(new BigDecimal(Math.PI)); zu beobachten; die Ausgabe suggeriert mit

3.141592653589793**115997963468544185161590576171875**

eine hohe Genauigkeit, richtig ist jedoch

3.141592653589793238462643383279502884197169399375

## Methoden statt Operatoren

Mit den `BigDecimal`-Objekten lässt sich nun rechnen, wie von `BigInteger` her 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)

### Rundungsmodus

Eine Besonderheit stellt die Methode `divide(...)` dar, die zusätzlich einen Rundungsmodus und optional auch eine Anzahl gültiger Nachkommastellen bekommen kann. Zunächst ohne Rundungsmodus:

```
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( BigDecimal.ONE.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 Laufzeitfehler: »java.lang.ArithmetiException: Non-terminating decimal expansion; no exact representable decimal result.«

An dieser Stelle kommen diverse Rundungsmodi ins Spiel, die bestimmen, wie die letzte Ziffer einer Rundung bestimmt werden soll. Sie lassen sich über eine Aufzählung `java.math.RoundingMode` übermitteln. Folgende Konstanten gibt es:

Konstante in RoundingMode	Bedeutung
DOWN	Runden nach 0
UP	Runden weg von 0
FLOOR	Runden nach negativ unendlich
CEILING	Runden nach positiv unendlich
HALF_UP	Runden zum nächsten Nachbarn und weg von der 0, wenn beide Nachbarn gleich weit weg sind

Tabelle 22.14 Bedeutung der RoundingMode-Konstanten

Konstante in RoundingMode	Bedeutung
HALF_DOWN	Runden zum nächsten Nachbarn und hin zur 0, wenn beide Nachbarn gleich weit weg sind
HALF_EVEN	Runden zum nächsten Nachbarn und zum geraden Nachbarn, wenn beide Nachbarn gleich weit weg sind
UNNECESSARY	Kein Runden, Operation muss exakt sein.

Tabelle 22.14 Bedeutung der RoundingMode-Konstanten (Forts.)

ROUND\_UNNECESSARY darf nur dann verwendet werden, wenn die Division exakt ist, sonst gibt es eine ArithmeticException.

Listing 22.13 src/main/java/com/tutego/insel/math/DivideRoundingMode.java, main()

```
BigDecimal one    = BigDecimal.ONE;
BigDecimal three = new BigDecimal( "3" );

System.out.println( one.divide( three, RoundingMode.UP ) );      // 1
System.out.println( one.divide( three, RoundingMode.DOWN ) );     // 0
```

Jetzt kann noch die Anzahl der Nachkommastellen bestimmt werden:

```
System.out.println( one.divide( three, 6, RoundingMode.UP ) );   // 0.333334
System.out.println( one.divide( three, 6, RoundingMode.DOWN ) ); // 0.333333
```



### 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 22.14 src/main/java/com/tutego/insel/math/RoundWithsetScale.java, main()

```
BigDecimal petrol = new BigDecimal( "1.399" ).multiply( new BigDecimal(45) );
System.out.println( petrol.setScale( 3, RoundingMode.HALF_UP ) );
System.out.println( petrol.setScale( 2, RoundingMode.HALF_UP ) );
```

Die Ausgaben sind 62.955 und 62.96.

```
class java.math.BigDecimal
extends Number
implements Comparable<BigDecimal>
```

- `BigDecimal divide(BigDecimal divisor, RoundingMode roundingMode)`
- `BigDecimal divide(BigDecimal divisor, int scale, RoundingMode roundingMode)`
- `BigDecimal setScale(int newScale, RoundingMode roundingMode)`

#### 22.6.4 Mit `MathContext` komfortabel die Rechengenauigkeit setzen

Ein Objekt vom Typ `java.math.MathContext` gibt für `BigDecimal` komfortabel die Rechengenauigkeit (nicht die Nachkommastellen) und den Rundungsmodus an. 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`-Objekt 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`-Objekt aus einem String auf. Der Aufbau des Strings entspricht der Formatierung von `MathContext.toString()`.
- `toString()`  
Liefert eine String-Repräsentation, etwa `precision=34 roundingMode=HALF_EVEN` bei `MathContext.DECIMAL128`.

Für die üblichen Fälle stehen vier vorgefertigte `MathContext`-Objekte als Konstanten der Klasse zur Verfügung: `DECIMAL128`, `DECIMAL32`, `DECIMAL64` und `UNLIMITED`.

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)`

## 22.6.5 Noch schneller rechnen durch mutable Implementierungen

Die beiden Datentypen `BigInteger` und `BigDecimal` sind immutable, was bei vielen Berechnungen zu Laufzeiteinbußen führt. Es gibt in der Open-Source-Welt durchaus Alternativen, etwa <https://github.com/tbuktu/bigint> oder <https://github.com/bwakell/Huldra>, das neben der Veränderbarkeit auch andere Algorithmen implementiert. `BigInteger` wird von Oracle allerdings auch laufend optimiert.

## 22.7 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 bzw. 64 Bit rechnen. Auf der PC-Seite kommen Intel und AMD mit internen Rechengenauigkeiten von 80 Bit, also 10 Byte, zum Zuge.<sup>13</sup> Die 80-Bit-Lösung bringt in Java zwei Nachteile mit sich:

---

<sup>13</sup> Dieses Dilemma betrifft aber nur 80x86- und andere CISC-Prozessoren. Bei RISC sind 32 Bit und 64 Bit das Übliche.

- ▶ 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 sehen aber die mathematischen Ergebnisse auf unterschiedlichen Maschinen anders aus.

### 22.7.1 Strikte Fließkommaberechnungen mit strictfp

Damit zum einen die Vorgaben der IEEE-Norm erfüllt werden und zum anderen die Geschwindigkeit gewährleistet werden kann, lässt sich der Modifizierer `strictfp` einsetzen – so geht die JVM bei Operationen strikt nach der IEEE-Norm vor. Wir nennen das *FP-strikt*. Ohne `strictfp` (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.

Der Modifizierer `strictfp` lässt sich an unterschiedlichen Stellen einsetzen: an Klassen, Schnittstellen oder Methoden, aber nicht an einzelnen Variablen. Ist ein Typ oder eine Methode `strictfp`, dann ist alles innerhalb des Typs oder der Methode FP-strikt.

### 22.7.2 Die Klassen Math und StrictMath

Für strikte mathematische Operationen gibt es eine eigene Klasse: `StrictMath`. An der Klassendeklaration für `StrictMath` lässt sich ablesen, dass sich alle Methoden an die IEEE-Norm halten.

**Listing 22.15** `java.lang.StrictMath.java`, `StrictMath`

```
public final strictfp class StrictMath {
    // ...
}
```

Allerdings gibt es nicht zwei Implementierungen der mathematischen Methoden – einmal strikt und genau bzw. einmal nicht strikt, dafür potenziell schneller. Bisher delegiert die Implementierung für `Math` direkt an `StrictMath`:

**Listing 22.16** `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(double, double)` strikt nach IEEE-Norm rechnen. Das ist zwar aus Sicht der Präzision und Übertragbarkeit der Ergebnisse wünschenswert, aber die Performance ist nicht optimal.

## 22.8 Geld und Währung

### 22.8.1 Geldbeträge repräsentieren

Für Geldbeträge gibt es in Java keinen eigenen Datentyp, und so kann eine Speicherung je nach Programm immer anders aussehen. Es bieten sich an:

- ▶ `BigDecimal`: Vorteil sind die präzisen Berechnungen und die wählbaren Rundungen.
- ▶ Paar von `int` bzw. `long`: getrenntes Speichern der Vor-/Nachkommastellen



#### Hinweis

Die primitiven Datentypen `double` und `float` sind wegen ihrer Unfähigkeit, Vielfaches von 0,01 korrekt dazustellen, nicht empfohlen; Rundungsfehler treten schnell auf.

#### Money and Currency API

Im »JSR 354: Money and Currency API« wird ein eigener Datentyp für Geldbeträge definiert, und die Typen sollten eigentlich in Java 9 aufgenommen werden, doch dazu kam es nicht. Dennoch sind die Typen interessant, und die Referenzimplementierung *Moneta* ist einen Blick wert: <http://javamoney.github.io/ri.html>. Neben Geldbeträgen erlaubt die kleine Bibliothek Umrechnungen, Formatierungen und eigene Währungen.

### 22.8.2 ISO 4217

Währungen werden durch Währungscodes beschrieben, und die Definition findet sich in der Norm ISO 4217. Einige ISO-Codes sind:

ISO-4217-Code	Währung/Einheit	Land
EUR	Euro	Länder der europäischen Währungsunion
CNY	Renminbi	China

Tabelle 22.15 Einige ISO-4217-Codes

ISO-4217-Code	Währung/Einheit	Land
DKK	Krone	Dänemark
GBP	Pfund	Vereinigtes Königreich
INR	Rupie	Indien
USD	Dollar	USA, Ecuador ...
XAU	Feinunze Gold	

Tabelle 22.15 Einige ISO-4217-Codes (Forts.)

Die Tabelle lässt am letzten Eintrag erkennen, dass es auch ISO-Codes für Edelmetalle und sogar Fonds gibt. Für jedes Kürzel gibt es ebenfalls einen numerischen Code.

### 22.8.3 Währungen in Java repräsentieren

Java repräsentiert Währungen durch die Klasse `java.util.Currency`. Exemplare der Klasse werden durch eine Fabrikmethode `getInstance(String currencyCode)` erfragt bzw. aus einer Aufzählung ausgewählt.

#### Beispiel

[zB]

Gib alle im System angemeldeten Währungen mit ein paar Informationen aus:

```
Currency.getAvailableCurrencies().stream()
    .sorted( Comparator.comparing( Currency::getCurrencyCode ) )
    .forEach( c -> System.out.printf( "%s, %s, %s, %s%n",
        c.getCurrencyCode(), c.getSymbol(),
        c.getDisplayName(), c.getNumericCode() ) );
```

Die Ausgabe beginnt so:

```
ADP, ADP, Andorranische Pesete, 20
AED, AED, VAE-Dirham, 784
AFA, AFA, Afghanische Afghani (1927-2002), 4
AFN, AFN, Afghanischer Afghani, 971
ALL, ALL, Albanischer Lek, 8
...
```

## 22.9 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>14</sup>

Auf dem (freien) Markt gibt es aber eine große Zahl an Erweiterungen, etwa für Brüche, Polynome, und Matrizen. Eine kleine Auswahl:

- ▶ *Eclipse January* (<https://projects.eclipse.org/proposals/january>). Junges Projekt der Eclipse Foundation für große n-dimensionale Felder und Matrizen. Bildet die unter Python bekannte NumPy-Bibliothek nach. Ziel ist die Definition von Standardschnittstellen.
- ▶ *Commons Math: The Apache Commons Mathematics Library* (<http://commons.apache.org/proper/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. Seit 2012 nicht mehr aktualisiert
- ▶ *JAMA: A Java Matrix Package* (<http://math.nist.gov/javanumerics/jama>). Bietet Berechnung von Eigenwerten, Lösen nichtsingulärer Systeme, Determinante ... Seit 2012 nicht mehr aktualisiert

Ein weiteres Problemfeld sind die Rechenun genauigkeiten, 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 Schlüsselwortes `strictfp` veröffentlicht, sodass nicht jeder Punkt gültig ist.

---

<sup>14</sup> [http://bugs.java.com/bugdatabase/view\\_bug.do?bug\\_id=4645692](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4645692)

# Kapitel 23

## Testen mit JUnit

»Es gibt keinen Grund, warum man einen Computer zu Hause haben sollte.«

– Ken Olson, Präsident der Digital Equipment Corp, 1977

### 23.1 Softwaretests

Um möglichst viel Vertrauen in die eigene Codebasis zu bekommen, bieten sich Softwaretests an. Tests sind kleine Programme, die ohne Benutzerkontrolle automatisch über die Quellcodebasis laufen und anhand von Regeln zeigen, dass vom Kunden gewünschte Teile sich so verhalten wie spezifiziert. (Die Abwesenheit von Fehlern kann eine Software natürlich nicht zeigen, aber immer zeigt ein Testfall, dass das Programm die Vorgaben aus der Spezifikation erfüllt.)

Obwohl Softwaretests extrem wichtig sind, sind sie unter Softwareentwicklern nicht unbedingt populär. Das liegt unter anderem daran, dass sie natürlich etwas Zeit kosten, die neben der tatsächlichen Entwicklung aufgewendet werden muss. Wenn dann die eigentliche Software geändert wird, müssen auch die Testfälle oftmals mit angepasst werden, sodass es gleich zwei Baustellen gibt. Und da Entwickler ein Feature eigentlich immer gestern fertigstellen sollten, fallen die Tests gerne unter den Tisch. Ein weiterer Grund ist, dass einige Entwickler sich für unfehlbare Kodierungsgötter halten, die jeden Programmcode (nach ein paar Stunden Debuggen) für absolut korrekt, performant und wohlduftend halten.

Wie lässt sich diese skeptische Gruppe nun überzeugen, doch Tests zu schreiben? Ein großer Vorteil von automatisierten Tests ist die Eigenschaft, dass bei großen Änderungen der Quellcodebasis (Refactoring) die Testfälle automatisch sagen, ob alles korrekt verändert wurde. Denn wenn nach dem Refactoring, etwa einer Performance-Optimierung, die Tests einen Fehler melden, ist wohl etwas kaputtoptimiert worden. Da Software einer permanenten Änderung unterliegt und nie fertig ist, sollte das Argument eigentlich schon ausreichen, denn wenn eine Software eine gewisse Größe erreicht hat, ist nicht absehbar, welche Auswirkungen Änderungen an der Quellcodebasis nach sich ziehen. Dazu kommt ein weiterer Grund, sich mit Tests zu beschäftigen: Es ist der positive Nebeneffekt, dass die erzeugte Software vom Design deutlich besser ist, denn testbare Software zu schreiben ist knifflig, führt aber fast zwangsläufig zu besserem Design. Und ein besseres Design ist immer erstrebenswert, denn es erhöht die Verständlichkeit und erleichtert die spätere Anpassung der Software.

### 23.1.1 Vorgehen beim Schreiben von Testfällen

Die Fokussierung bei Softwaretests liegt auf zwei Attributen: automatisch, wiederholbar und nachvollziehbar. Dazu ist eine Bibliothek nötig, die zwei Dinge unterstützen muss:

- ▶ Testfälle sehen immer gleich aus und bestehen aus drei Schritten. Zunächst wird ein Szenario aufgebaut, dann wird die zu testende Methode oder Methodenkombination aufgerufen, und zum Schluss wird mit der spezifischen API vom Test-Framework geschaut, ob das ausgeführte Programm genau das gewünschte Verhalten gebracht hat. Das übernehmen eine Art »Stimmt-es-dass«-Methoden, die den gewünschten Zustand mit dem tatsächlichen abgleichen und bei einem Konflikt eine Ausnahme auslösen. Im Fehlerfall sollen die Tests genau dokumentieren, was schiefgelaufen ist.
- ▶ Das Test-Framework muss die Tests laufen lassen und im Fehlerfall eine Meldung geben; dieser Teil nennt sich *Test-Runner*.

Wir werden uns im Folgenden auf so genannte *Unit-Tests* beschränken. Das sind Tests, die einzelne Bausteine (engl. *units*) testen. Daneben gibt es andere Tests, wie Lasttests, Performance-Tests oder Integrationstests, die aber im Folgenden keine große Rolle spielen.

## 23.2 Das Test-Framework JUnit

Oracle definiert kein allgemeines Standard-Framework zur Definition von Unit-Testfällen, und es bietet auch keine Ablaufumgebung für Testfälle. Diese Lücke füllen Test-Frameworks, wobei das populärste im Java-Bereich das freie quelloffene *JUnit* (<https://junit.org/junit5/>) ist; mehr als 60 % aller quelloffenen Projekte unter GitHub referenzieren diese Bibliothek.

### JUnit-Versionen

Das JUnit-Framework wurde im Jahr 2000 von Kent Beck und Erich Gamma entwickelt, die aktuellen Änderungen kommen von diversen Entwicklern. Der alte JUnit 3-Zweig nutzte noch keine Annotationen, das änderte sich 2006 mit der Version JUnit 4. Aktuell ist die Version JUnit 5, die eine neue Modularisierung hat: JUnit 5 besteht aus den Teilen JUnit Platform, JUnit Jupiter und JUnit Vintage. JUnit 5 benötigt mindestens Java 8 und benennt auch Annotationen um, sodass es gewissen Migrationsaufwand bei der Umstellung von JUnit 4 gibt.

### JUnit aufnehmen

Wir wollen mit der aktuellen Version JUnit 5 arbeiten. Wir können natürlich JUnit von Hand mit in den Klassenpfad aufnehmen, doch über Maven und die IDE geht es einfacher. Daher fügen wir folgende Abhängigkeit in der POM-Datei ein:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
```

Die Standard-IDEs *Eclipse* und *IntelliJ* bringen JUnit gleich mit und bieten Wizards zum einfachen Erstellen von Testfällen aus vorhandenen Klassen an. Über Tastendruck lassen sich Testfälle abarbeiten, und ein farbiger Balken zeigt direkt an, ob wir unsere Arbeit gut gemacht haben.

### 23.2.1 Test-Driven Development und Test-First

Unser JUnit-Beispiel wollen wir nach einem ganz speziellen Ansatz entwickeln, der sich *Test-First* nennt. Dabei wird der Testfall noch vor der eigentlichen Implementierung geschrieben. Die Reihenfolge mit dem Test-First-Ansatz sieht (etwas erweitert) so aus:

1. Überlege, welche Klasse und Methode geschrieben werden soll. Lege Quellcode für die Klasse und für die Variablen/Methoden/Konstruktoren an, sodass sich die Kompilationseinheit übersetzen lässt. Die Codeblöcke sind leer, enthalten aber mitunter eine return-Anweisung mit Rückgabe, sodass die Typen und Methoden/Konstruktoren »da« sind, aber keine Funktionalität besitzen.
2. Schreibe die API-Dokumentation für die Methoden/Konstruktoren, und überlege, welche Parameter, Rückgaben und Ausnahmen nötig sind.
3. Teste die API an einem Beispiel, das zeigt, ob sich die Klasse mit Eigenschaften »natürlich« anfühlt. Falls nötig, wechsle zu Punkt 1, und passe die Eigenschaften an.
4. Implementiere eine Testklasse.
5. Implementiere die Logik des eigentlichen Programms.
6. Gibt es durch die Implementierung neue Dinge, die ein Testfall testen sollte?  
Wenn ja, erweitere den Testfall.
7. Führe die Tests aus, und wiederhole Schritt 5, bis alles fehlerfrei läuft.

Der Test-First-Ansatz hat den großen Vorteil, dass er überschnelle Entwickler, die, ohne groß zu denken, zur Tastatur greifen, dann implementieren und nach 20 Minuten wieder alles ändern, zum Nachdenken zwingt. Große Änderungen kosten Zeit und somit Geld, und Test-First verringert die Notwendigkeit späterer Änderungen. Denn wenn Entwickler Zeit in die API-Dokumentation investieren und Testfälle schreiben, dann haben sie eine sehr gute Vorstellung von der Arbeitsweise der Klasse, und große Änderungen sind seltener.

Der Test-First-Ansatz ist eine Anwendung von *Test-Driven Development* (TDD). Hier geht es darum, die Testbarkeit gleich als Ziel bei der Softwareentwicklung zu definieren. Hieran

krankten frühere Entwicklungsmodelle, etwa das wohlbekannte Wasserfallmodell, das das Testen an das Ende – nach Analyse, Design und Implementierung – stellte. Die Konsequenz dieser Reihenfolge war oft ein großer Klumpen Programmcode, der unmöglich zu testen war. Mit TDD soll das nicht mehr passieren. Heutzutage sollten sich Entwickler bei jeder Architektur, jedem Design und jeder Klasse gleich zu Beginn überlegen, wie das Ergebnis zu testen ist. Untersuchungen zeigen: Mit TDD ist das Design signifikant besser.

Zu der Frage, wann Tests durchgeführt werden sollen, lässt sich nur eines festhalten: so oft wie möglich. Denn je eher ein Test durch eine falsche Programmänderung fehlschlägt, desto eher kann der Fehler behoben werden. Gute Zeitpunkte sind daher vor und hinter größeren Designänderungen und auf jeden Fall vor dem Einpflegen in die Versionsverwaltung. Im modernen Entwicklungsprozess gibt es einen Rechner, auf dem eine Software zur kontinuierlichen Integration läuft (engl. *continuous integration*). Diese Systeme integrieren einen Build-Server, der die Quellen automatisch aus einer Versionsverwaltung auscheckt, compiliert und dann Testfälle und weitere Metriken laufen lässt. Diese Software übernimmt dann einen *Integrationstest*, da hier alle Module der Software zu einer Gesamtheit zusammengebaut werden und so Probleme aufgezeigt werden, die sich vielleicht bei isolierten Tests auf den Entwicklermaschinen nicht zeigen.

### 23.2.2 Testen, implementieren, testen, implementieren, testen, freuen

Bisher bietet Java keine einfache Funktion, die Strings umdreht. Unser erstes JUnit-Beispiel soll daher um eine Klasse `StringUtils` gestrickt werden, die eine statische Methode `reverse()` anbietet. Nach dem TDD-Ansatz implementieren wir eine Klasse und die Methode, sodass korrekt übersetzt werden kann, aber alles zunächst ohne Funktionalität ist. (Auf die API-Dokumentation verzichtet das Beispiel.)

**Listing 23.1** src/main/java/com/tutego/insel/junit/util/StringUtils.java, `StringUtils`

```
public class StringUtils {

    public static String reverse( String string ) {
        return null;
    }
}
```

Gegen diese eigene API lässt sich nun der Testfall schreiben. Spontan fällt uns ein, dass ein Leer-String umgedreht natürlich auch einen Leer-String ergibt und die Zeichenkette »abc« daher umgedreht »cba« ergibt. Unser Ziel ist es, eine möglichst gute *Abdeckung* aller Fälle zu bekommen. Wenn wir Fallunterscheidungen im Programmcode vermuten, sollten wir versuchen, so viele Testfälle zu finden, dass alle diese Fallunterscheidungen abgelaufen werden.

Interessant sind beim Eingeben immer Sonderfälle bzw. Grenzen von Wertebereichen. (Unsere Methode gibt da nicht viel her, aber wenn wir etwa eine Substring-Funktion haben, lassen sich schnell viele Methodenübergaben finden, die interessant sind.)

**Listing 23.2** src/test/java/com/tutego/insel/junit/util/StringUtilsTest.java

```
package com.tutego.insel.junit.util;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class StringUtilsTest {

    @Test
    public void testReverse() {
        assertEquals( "", StringUtils.reverse( "" ) );
        assertEquals( "cba", StringUtils.reverse( "abc" ) );
    }
}
```

Die Klasse zeigt vier Besonderheiten:

1. Die Methoden, die sich einzelne Szenarien vornehmen und die Klassen/Methoden testen, tragen die Annotation `@Test`.
2. Eine übliche Namenskonvention (obwohl sie nicht zwingend nötig ist) ist, dass die Methode, die den Test enthält, mit dem Präfix »test« beginnt und mit dem Namen der Methode endet, die sie testet. Da in unserem Fall die Methode `reverse(...)` getestet wird, heißt die Testmethode dementsprechend `testReverse`. Eine `testXXX()`-Methode liefert nie eine Rückgabe. Testmethoden können auch ganze Szenarien testen, die nicht unbedingt an einer Methode festzumachen sind, aber hier testet `testReverse()` nur die `reverse(...)`-Methode.
3. JUnit bietet eine Reihe von `assertXXX(...)`-Methoden, die den erwarteten Zustand mit dem Ist-Zustand vergleichen; gibt es Abweichungen, folgt eine Ausnahme. `assertEquals(...)` nimmt einen `equals(...)`-Vergleich der beiden Objekte vor. Wenn demnach `StringUtils.reverse("")` die leere Zeichenkette `" "` liefert, ist alles in Ordnung, und der Test wird fortgesetzt.
4. Der statische Import aller statischen Eigenschaften der Klasse `org.junit.jupiter.api.Assertions` kürzt die Schreibweise ab, sodass im Programm statt `Assertions.assertEquals(...)` nur `assertEquals(...)` geschrieben werden kann.

### 23.2.3 JUnit-Tests ausführen

In einer Entwicklungsumgebung lässt sich die Testausführung leicht ausführen. Eclipse zeigt zum Beispiel die Ergebnisse in der JUnit-View an und bietet mit einem grünen oder roten Balken direktes visuelles Feedback.

Natürlich lassen sich die Tests auch von der Kommandozeile ausführen, obwohl das selten ist, denn in der Regel werden die Tests im Zuge eines Build-Prozesses – den etwa Maven steuert – angestoßen. Wer das dennoch über die Kommandozeile machen möchte, nutzt den Console-Launcher. In jedem Fall benötigen wir in Maven eine Abhängigkeit und im Klassenpfad *junit-platform-launcher-x.y.z.jar*:

**Listing 23.3** pom.xml, Ausschnitt

```
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-launcher</artifactId>
  <version>1.3.1</version>
  <scope>test</scope>
</dependency>
```

### 23.2.4 assertXXX(..)-Methoden der Klasse Assert

Assert ist die Klasse mit den assertXXX(..)-Methoden, die immer dann einen AssertionError auslösen, wenn ein aktueller Wert nicht so wie der gewünschte war. Der JUnit-Runner registriert alle AssertionErrors und speichert sie für die Statistik. Bis auf drei Ausnahmen beginnen alle Methoden der Klasse Assert mit dem Präfix assert – zwei andere heißen fail(..), und eine heißt isArray(..). Die assertXXX(..)-Methoden gibt es einmal mit einer Testmeldung, die dann erscheint, wenn JUnit extra eine Meldung angeben soll, und einmal ohne, wenn keine Extramedlung gefragt ist. Die Meldungen können auch über einen Supplier<String> geliefert werden, das ist in der folgenden Dokumentation nicht mit aufgeführt.

#### Ist etwas wahr oder falsch?

Eigentlich reicht zum Testen die Methode `assertTrue(boolean condition)` aus. Ist die Bedingung wahr, so ist alles in Ordnung. Wenn nicht, gibt es den AssertionError:

```
class org.junit.jupiter.api.Assertions
```

- `static void assertTrue(boolean condition)`
- `static void assertTrue(String message, boolean condition)`
- `static void assertFalse(boolean condition)`

- static void assertFalse(String message, boolean condition)
- Weiterhin gibt es assert[True|False](...) mit einem BooleanSupplier.

### Ist etwas null?

Um es Entwicklern etwas komfortabler zu machen, bietet JUnit sechs Kategorien an Hilfsmethoden. Zunächst sind es assertNull(...) und assertNotNull(...), die testen, ob das Argument null bzw. nicht null ist. Ein Aufruf von assertNull(Object object) ist dann nichts anderes als assertTrue(object == null):

- static void assertNull(Object object)
- static void assertNull(String message, Object object)
- static void assertNotNull(Object object)
- static void assertNotNull(String message, Object object)

### Sind Objekte identisch?

Die nächste Kategorie testet, ob das Objekt identisch (nicht equals(...)-gleich) mit einem anderen Objekt ist:

- static void assertNotSame(Object unexpected, Object actual)
- static void assertNotSame(String message, Object unexpected, Object actual)
- static void assertEquals(Object expected, Object actual)
- static void assertEquals(String message, Object expected, Object actual)

### Sind Objekte gleichwertig?

Statt eines Referenztests führen die folgenden Methoden einen equals(...)-Vergleich durch:

- static void assertEquals(Object expected, Object actual)
- static void assertEquals(String message, Object expected, Object actual)

### Sind primitive Werte gleich?

Zum Testen von primitiven Datentypen gibt es im Grunde nur drei Methoden. (Das reicht, denn zum einen werden andere primitive Typen automatisch typangepasst, zum anderen kommt dann Boxing ins Spiel, sodass assertEquals(Object, Object) wieder passt.)

- static void assertEquals(long expected, long actual)
- static void assertEquals(String message, long expected, long actual)
- static void assertEquals(float|double expected, float|double actual, float|double delta)
- static void assertEquals(String message, float|double expected, float|double actual, float|double delta)

Fließkommazahlen bekommen bei `assertEquals(...)` einen Delta-Wert, in dem sich das Ergebnis bewegen muss. Das trägt der Tatsache Rechnung, dass vielleicht in der Bildschirmausgabe zwei Zahlen gleich sind, jedoch nicht bitweise gleich sind, da sich kleine Rechenfehler akkumuliert haben. Sind jedoch die Fließkommazahlen in einem Wrapper, also etwa `Double`, verpackt, leitet ja `assertEquals(...)` den Test nur an die `equals(...)`-Methode der Wrapper-Klasse weiter, die natürlich kein Delta berücksichtigt.

### Sind Arrays gleich?

Weitere Methoden vergleichen Array-Inhalte:

- `static void assertEquals(byte[] expecteds, byte[] actuals)`
- `static void assertEquals(String message, byte[] expecteds, byte[] actuals)`
- `static void assertEquals(char[] expecteds, char[] actuals)`
- `static void assertEquals(String message, char[] expecteds, char[] actuals)`
- `static void assertEquals(int[] expecteds, int[] actuals)`
- `static void assertEquals(String message, int[] expecteds, int[] actuals)`
- `static void assertEquals(long[] expecteds, long[] actuals)`
- `static void assertEquals(Object[] expecteds, Object[] actuals)`
- `static void assertEquals(String message, long[] expecteds, long[] actuals)`
- `static void assertEquals(String message, Object[] expecteds, Object[] actuals)`
- `static void assertEquals(short[] expecteds, short[] actuals)`
- `static void assertEquals(String message, short[] expecteds, short[] actuals)`

Neben den `assertEquals(...)`-Methoden gibt es für einige Varianten Negationen:

- `static void assertNotEquals(long unexpected, long actual)`
- `static void assertNotEquals(float unexpected, float actual, float delta)`
- `static void assertNotEquals(double unexpected, double actual, double delta)`
- `static void assertNotEquals(Object unexpected, Object actual)`
- `static void assertNotEquals(String message, long unexpected, long actual)`
- `static void assertNotEquals(String message, float unexpected, float actual, float delta)`
- `static void assertNotEquals(String message, double unexpected, double actual, double delta)`
- `static void assertNotEquals(String message, Object unexpected, Object actual)`

### Ist das alles wahr oder falsch?

In JUnit 5 ist ein neuer Typ `org.junit.jupiter.api.function.Executable` eingezogen, mit dem sich ein beliebiger Block Code ausdrücken lässt. Assertions nimmt diesen Typ an über:

- static void assertAll(Executable... executables)
- static void assertAll(Stream<Executable> executables)
- static void assertAll(String heading, Executable... executables)
- static void assertAll(String heading, Stream<Executable> executables)

Ein Executable ist eine funktionale Schnittstelle mit einer Methode void execute() throws Throwable. Eine Ausnahme soll execute() beenden, und Assertions.assertAll(...) fängt diese Ausnahme auf und meldet den Fehler bei der Abarbeitung des Testcodes.

### 23.2.5 Hamcrest

Mit Drittbibliotheken wie *Hamcrest* (<http://hamcrest.org/JavaHamcrest/>), *AssertJ* (<https://joel-costigliola.github.io/assertj/>) oder *Truth* (<https://google.github.io/truth/>) lassen sich Tests deklarativ schreiben, sodass sie sich schon fast wie englische Sätze lesen.

Wir wollen die API für Hamcrest der von JUnit kurz gegenüberstellen – Hamcrest verwendet `assertThat(...)`- statt `assertXXX(..)`-Methoden:

<code>assertXXX(..)</code> von JUnit	<code>assertThat(..)</code> von Hamcrest
<code>assertNotNull(new Object());</code>	<code>assertThat(new Object(), is(notNullValue()));</code>
<code>assertEquals("", StringUtils.reverse(""));</code>	<code>assertThat("", is(equalTo(StringUtils.reverse(" "))));</code>
<code>assertSame("", "");</code>	<code>assertThat("", is(sameInstance(" ")));</code>
<code>assertNotSame("", "a");</code>	<code>assertThat("a", is(not(sameInstance(" "))));</code>

Tabelle 23.1 Vergleich der `assertXXX(..)`- und `assertThat(..)`-Methoden

Zunächst fällt auf, dass das erste Argument bei `assertThat(..)` den Wert beschreibt, den wir haben, der aber bei den sonstigen `assertXXX(..)`-Methoden immer erst hinter dem erwarteten Wert folgt. Das zweite Argument ist ein besonderes Matcher-Objekt, das die Bedingung kodiert.

JUnit hängt nicht von Hamcrest ab, sodass wir als Erstes das Java-Archiv *java-hamcrest-2.0.0.0.jar* in den Klassenpfad einbinden müssen. (Das Release ist schon älter, vom 18. Januar 2015, und bisher noch nicht aktualisiert.) Maven-Nutzer schreiben:

#### Listing 23.4 pom.xml, Ausschnitt

```
<dependency>
  <groupId>org.hamcrest</groupId>
```

```
<artifactId>java-hamcrest</artifactId>
<version>2.0.0.0</version>
<scope>test</scope>
</dependency>
```

## Matcher-Objekte

Die allgemeine Syntax von `assertThat(...)` ist folgende:

```
class org.hamcrest.MatcherAssert
```

- `static <T> void assertThat(T actual, Matcher<? super T> matcher)`
- `static <T> void assertThat(String reason, T actual, Matcher<? super T> matcher)`

Ob der Test korrekt ist oder nicht, entscheidet ein `org.hamcrest.Matcher`-Objekt.

Sehen wir uns ein Beispiel an:

**Listing 23.5** com/tutego/insel//hamcrest/HamcrestText.java, Ausschnitt

```
@Test
public void testHamcrestMatcher() {
    assertThat( new Object(), is( notNullValue() ) );
    assertThat( "", is( equalTo( "" ) ) );
    assertThat( 1L, is( sameInstance( Long.valueOf( 1 ) ) ) );
    assertThat( 1000L, is( not( sameInstance( Long.valueOf( 1000 ) ) ) ) );
}
```

Da die `Matcher` über eine clevere Art verschachtelt werden, lesen sich die `assertThat(...)`-Aufrufe wie Sätze. `is(...)` hat funktional keine Bedeutung, lässt die Aussage aber noch »englischer« werden. Aber wo sind bei einem Aufruf wie `assertThat("", not(sameInstance("a")))` die Objekte? Zunächst gilt, dass »Wörter« wie »not« und »equalTo« statische Methoden der Klasse `org.hamcrest.CoreMatchers` sind. Werden diese Methoden statisch eingebunden, ergibt sich die kurze Schreibweise, sonst ist sie lang und lautet:

```
CoreMatchers.not(CoreMatchers.sameInstance("a"))
```

Die statischen Methoden geben `Matcher`-Objekte zurück. Wir hätten auch `new IsNot<String>(new IsSame<String>("a"))` schreiben können, und dann hätten wir auch wieder das `Matcher`-Objekt, aber die statischen Methoden sind kürzer als der Konstruktorauftrag. Alle `Matcher`-Objekte besitzen eine Methode `boolean matches(Object item)`, die letztendlich den Test durchführt, und `assertThat(...)` sagt, ob eine Ausnahme ausgelöst werden muss, weil ein Fehler auftrat.

## CoreMatchers-Methoden

Die Methoden `is(...)`, `isInstanceOf(...)`, `not(...)`, `notNullValue(...)`, `equalTo(...)` sind nicht die einzigen aus CoreMatchers. Die folgende Übersicht zählt die aktuellen statischen Methoden auf:

```
class org.hamcrest.CoreMatchers
```

- `static <T> Matcher<T> allOf(Iterable<Matcher<? extends T>> matchers)`
- `static <T> Matcher<T> allOf(Matcher<? extends T>... matchers)`
- `static <T> Matcher<T> any(Class<T> type)`
- `static <T> Matcher<T> anyOf(Iterable<Matcher<? extends T>> matchers)`
- `static <T> Matcher<T> anyOf(Matcher<? extends T>... matchers)`
- `static <T> Matcher<T> anything()`
- `static <T> Matcher<T> anything(String description)`
- `static <T> Matcher<T> describedAs(String description, Matcher<T> matcher, Object... values)`
- `static <T> Matcher<T> equalTo(T operand)`
- `static Matcher<Object> instanceOf(Class<?> type)`
- `static Matcher<Object> is(Class<?> type)`
- `static <T> Matcher<T> is(Matcher<T> matcher)`
- `static <T> Matcher<T> is(T value)`
- `static <T> Matcher<T> not(Matcher<T> matcher)`
- `static <T> Matcher<T> not(T value)`
- `static <T> Matcher<T> notNullValue()`
- `static <T> Matcher<T> notNullValue(Class<T> type)`
- `static <T> Matcher<T> nullValue()`
- `static <T> Matcher<T> nullValue(Class<T> type)`
- `static <T> Matcher<T> sameInstance(T object)`

## Vorteile von Matcher-Objekten

Stellen wir noch einmal

- ▶ `assertEquals("", StringUtils.reverse(""));` und
- ▶ `assertThat("", is(equalTo(StringUtils.reverse(""))));`

gegenüber. Ist `assertThat(...)` die bessere Alternative? Nicht wirklich, denn es gibt keinen Nutzen, wenn `assertThat(...)` exakt das übernimmt, was die `assertXXX(...)`-Methode macht, also in unserem Fall einen Gleichheitstest. `assertThat(...)` ist dann sogar länger.

Interessant wird `assertThat(...)` aus zwei Gründen:

1. Es gibt einige Sammlungen an Matcher-Objekten, die einem Programmierer viel Arbeit abnehmen. So etwas bietet zum Beispiel Spotify unter <https://github.com/spotify/java-hamcrest> Matcher für JSON-Objektstrukturen oder unter <https://github.com/seinesoftware/hamcrest-path> für Files-Operationen.<sup>1</sup> Ein zweites Beispiel betrifft Mengenabfragen. Die praktische Methode `hasItems(...)` zum Beispiel testet, ob Elemente in einer Collection sind; ohne Matcher wäre der Test in Java mehr Schreibarbeit.
2. Die `assertEquals(...)`-Methode läuft entweder durch oder bricht mit einer Exception ab, was den Test dann beendet. Wir bekommen beim Abbruch dann den Hinweis, dass der gewünschte Wert nicht mit dem berechneten übereinstimmt, aber wo genau der Fehler ist, das fällt weniger auf. `assertThat(...)` liefert ausgezeichnete Fehlermeldungen.

Folgendes Beispiel fasst die beiden Vorteile zusammen: Stellen wir uns vor, wir haben eine Datenstruktur (in diesem Beispiel eine `ArrayList`). Sie kann Elemente auch entfernen, und das ist genau die Methode `removeAll(...)`, deren Funktionalität wir testen wollen:

```
List<String> list = new ArrayList<>();
Collections.addAll( list, "a", "b", "c", "d", "e" );
list.removeAll( Arrays.asList( "b", "d" ) );
```

Wie kann ein Test aussehen? Ein Test könnte schauen, ob die Größe der Liste von fünf Elementen auf 3 kommt, wenn die beiden Elemente »b« und »d« gelöscht werden. Und der Test kann prüfen, ob »b« und »d« wirklich entfernt wurden, aber »a«, »c« und »e« weiterhin in der Liste sind:

**Listing 23.6** com/tutego/insel//hamcrest/HamcrestText.java, Ausschnitt

```
public void testHamcrestCollection() {
    List<String> list = new ArrayList<>();
    Collections.addAll( list, "a", "b", "c", "d", "e" );
    list.removeAll( Arrays.asList( "b", "d" ) );
    assertThat( list, hasSize(3) );
    assertThat( list,
        both( hasItems( "a", "c", "e" ) ).and( not( hasItems( "b", "d" ) ) ) );
}
```

Die Methode `hasSize(...)` prüft die Größe der Liste, und `hasItems(...)` testet, ob Elemente in der Datenstruktur sind. Die Kombination `both(...).and(...)` prüft zwei Bedingungen, die beide erfüllt sein müssen. Alternativ wäre auch `allOf(...)` möglich.

---

<sup>1</sup> Wobei `allOf(exists(), sized(0))` noch etwas besser ist – der Autor ist informiert.

Im Fehlerfall gibt es präzisere Fehlermeldungen. Ändern wir zum Test das erste `hasItems(...)` in `hasItems("_", "c", "e")`. Der Testlauf wird dann natürlich einen Fehler geben. Die Meldung ist (etwas eingerückt):

```
Expected: (
  (a collection containing "_" and a collection containing "c" and a collection
  containing "e")
  and not
  (a collection containing "b" and a collection containing "d")
)

but: (
  a collection containing "_" and
  a collection containing "c" and
  a collection containing "e"
)
a collection containing "_"

mismatches were:
[was "a", was "c", was "e"]
```

Der Hinweis `[was "a", was "c", was "e"]` zeigt den Inhalt der Liste und hilft bei der Zuordnung, welcher Matcher fehlschlägt. Das ist lokaler, als sonst bei `assertTrue(...)` nur für den gesamten Ausdruck festzustellen, ob dieser wahr oder falsch war.

### 23.2.6 Exceptions testen

Während der Implementierung fallen oft Dinge auf, die die eigentliche Implementierung noch nicht berücksichtigt. Dann sollte sofort diese neu gewonnene Erkenntnis in den Testfall einfließen. In unserem Beispiel soll das bedeuten, dass bisher nicht wirklich geklärt ist, was bei einem `null`-Argument passieren soll. Bisher gibt es eine `NullPointerException`, und das ist auch völlig in Ordnung, aber in einem Testfall steht das bisher nicht, dass auch wirklich eine `NullPointerException` folgt. Diese Fragestellung legt den Fokus auf eine gern vergessene Seite des Testens, denn Testautoren dürfen sich nicht nur darauf konzentrieren, was die Implementierung denn so alles richtig machen soll – der Test muss auch kontrollieren, ob im Fehlerfall auch dieser korrekt gemeldet wird. Wenn es nicht in der Spezifikation steht, dürfen auf keinen Fall falsche Werte geradegebügelt werden: Falsche Werte müssen immer zu einer Ausnahme oder zu einem wohldefinierten Verhalten führen.

### Versuch und fail(...)

Wir wollen unser Beispiel so erweitern, dass `reverse(null)` eine `IllegalArgumentException` auslöst. Auf zwei Arten lässt sich testen, ob die erwartete `IllegalArgumentException` auch wirklich kommt. Die erste Variante:

**Listing 23.7** com/tutego/insel/junit/utils/StringUtilsTest.java, Ausschnitt

```
try {
    StringUtils.reverse( null );
    fail( "reverse(null) should throw IllegalArgumentException" );
}
catch ( IllegalArgumentException e ) { /* Ignore */ }
```

Führt `reverse(null)` zur Ausnahme, was ja gewollt ist, dann wird der `catch`-Block die `IllegalArgumentException` einfach auffangen und ignorieren, und dann geht es in der Testfunktion mit anderen Dingen weiter. Sollte keine Ausnahme folgen, so wird die Anweisung nach dem `reverse(...)`-Aufruf ausgeführt, und die ist `fail(...)`. Diese Methode löst eine JUnit-Ausnahme mit einer Meldung aus und signalisiert dadurch, dass im Test etwas nicht stimmte.

### @Test(expected = XXXException.class)

Allerdings bleibt ein Problem: Was ist, wenn zwar eine Ausnahme ausgelöst wird, aber eine falsche? Eine Nicht-`RuntimeException` kann es nicht sein, denn dann würde der Compiler uns zum `catch`-Block zwingen. Aber was wäre mit einer anderen `RuntimeException`, etwa der `NullPointerException`? Diese würde zwar von JUnit abgefangen werden, und JUnit würde einen Fehler melden, aber dann ist nicht abzulesen, was das für ein Fehler ist und welche Rolle er spielt. Eine Lösung wäre, noch einen `catch`-Block anzuhängen und `fail(...)` aufzurufen, doch das würde erst einmal Quellcodeduplikierung bedeuten. Daher bietet JUnit eine andere, eine elegante Variante – die Annotation `@Test` wird parametrisiert:

**Listing 23.8** com/tutego/insel/junit/util/StringUtilsTest.java, `testReverseException()`

```
@Test( expected = IllegalArgumentException.class )
public void testReverseException() {
    StringUtils.reverse( null );
}
```

JUnit erwartete (engl. *expected*) hier eine `IllegalArgumentException`. Folgt sie nicht, meldet JUnit das als Fehler. Der Vorteil gegenüber der `fail(...)`-Variante ist die Kürze; ein Nachteil ist, dass dann unter Umständen mehrere `testXXX()`-Methoden für eine zu testende Methode nötig sind. Wir haben hier eine zweite Methode, `testReverseException()`, hinzugenommen. (Der `reverse(null)`-Test hätte auch am Ende der ersten Methode stehen können, dann müssen wir aber sichergehen, dass dieser Exception-Test am Ende durchgeführt wird.)

### assertThrows(...)

Eine neue Möglichkeit von JUnit 5 bietet Assertions mit folgenden Methoden:

- static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)
- static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable, String message)
- static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable, Supplier<String> messageSupplier)

### 23.2.7 Tests ignorieren

Durch Umstrukturierung von Quellcode kann es sein, dass Testcode nicht länger gültig ist und entfernt oder umgebaut werden muss. Damit der Testfall nicht ausgeführt wird, muss er nicht auskommentiert werden (das bringt den Nachteil mit sich, dass sich das Refactoring etwa im Zuge einer Umbenennung von Bezeichnern nicht auf auskommentierte Bereiche auswirkt). Stattdessen wird einfach eine weitere Annotation @Ignore an die Methode gesetzt:

```
@Ignore
@Test
public void testReverse()
```

### 23.2.8 Grenzen für Ausführungszeiten festlegen

Nach großen Refactorings kann die Software funktional noch laufen, aber vielleicht viel langsamer geworden sein. Dann stellt sich die Frage, ob es im Sinne des Anforderungskatalogs noch korrekt ist, wenn ein performantes Programm nach einer Änderung wie eine Schnecke läuft. Um Laufzeitveränderungen als Gültigkeitskriterium einzuführen, kann die Annotation @Test ein timeout in Millisekunden angeben:

```
@Test( timeout = 500 )
public void test()
```

Wird die Testmethode dann nicht innerhalb der Schranke ausgeführt, gilt das als fehlgeschlagener Test, und JUnit meldet einen Fehler.

Neu in JUnit 5 sind die Methoden assertTimeout(...) und assertTimeoutPreemptively(...), die nach einer gegebenen Duration ein Ergebnis von einem Executable oder ThrowingSupplier erwarten.

### 23.2.9 Mit Methoden der Assumptions-Klasse Tests abbrechen

Während die `assertXXX(...)`-Methoden zu einer Ausnahme führen und so anzeigen, dass der Test etwas gefunden hat, was nicht korrekt ist, bietet JUnit mit `Assumptions.assumeXXX(...)`-Methoden die Möglichkeit, die Tests nicht fortzuführen. Das ist zum Beispiel dann sinnvoll, wenn die Testausführung nicht möglich ist, etwa weil der Testrechner keine Grafikkarte hat, das Netzwerk nicht da oder das Datensystem voll ist. Dabei geht es nicht darum, zu testen, wie sich die Routine bei einem nicht vorhandenen Netzwerk verhält – das gilt es natürlich auch zu testen. Aber steht das Netzwerk nicht, dann können logischerweise auch keine Tests laufen, die das Netzwerk zwingend benötigen. Zwei der über 10 Methoden:

```
class org.junit.jupiter.api.Assumptions
```

- `static void assumeTrue(boolean assumption)`
- `static void assumeFalse(boolean assumption)`

Die `assumeXXX(...)`-Methoden führen zu keiner Ausnahme, brechen die Testausführung aber ab.

## 23.3 Wie gutes Design das Testen ermöglicht

Statische Methoden nach dem Muster »Parameter liefert die Eingabe, der Rückgabewert das Ergebnis« sind einfach zu testen. Sie verändern keine Umgebung, und Zustände gibt es keine. Der Testfall muss lediglich die Rückgabe untersuchen, und das ist einfach. Aufwändiger wird es dann schon, wenn Dinge getestet werden sollen, die aufwändige Systemänderungen nach sich ziehen: Wurde eine Datei angelegt? Stehen Dinge in der Datenbank wie gewünscht? Hat der Cluster die Daten auf andere Server gespiegelt? Liefert ein externes Programm die Rückgabe wie erwartet? Gibt eine native Methode tatsächlich das zurück, was sie verspricht, und zieht sie nicht die JVM ins Grab?

Sind Dinge plötzlich nicht mehr testbar, offenbart das im Allgemeinen ein schwaches Design. Häufig liegt es daran, dass eine Klasse zu viele Verantwortlichkeiten hat. Als Beispiel wollen wir uns eine Klasse ansehen, die Visitenkarten im vCard-Format (Dateiendung `.vcf`) schreibt.<sup>2</sup> Um den Quellcode schlank zu halten, verzichtet die Klasse `VCard` auf Setter/Getter.

**Listing 23.9** com/tutego/insel/junit/util/vdf/v1/VCard.java, VCard

```
public class VCard {

    public String formattedName;
    public String email;
```

---

<sup>2</sup> Mehr Informationen zum Dateiformat gibt <https://de.wikipedia.org/wiki/VCard>.

```

public void export( String filename ) throws IOException {
    StringBuilder result = new StringBuilder( "BEGIN:VCARD\n" );
    if ( formattedName != null && ! formattedName.isEmpty() )
        result.append( "FN:" ).append( formattedName ).append( "\n" );
    if ( email != null && ! email.isEmpty() )
        result.append( "EMAIL:" ).append( email ).append( "\n" );
    Files.write( Paths.get( filename ),
                Collections.singleton( result.append( "END:VCARD" ) ) );
}
}

```

Wenn die Anwendung etwa die Variable `formattedName` auf »Powerpuff Girls« setzt und `email` auf »powerpuff@townsville.com«, dann würde die Methode `export(...)` eine Datei mit dem folgenden Inhalt erstellen:

```

BEGIN:VCARD
FN:Powerpuff Girls
EMAIL:powerpuff@townsville.com
END:VCARD

```

Die Hauptaufgabe der Klasse ist die korrekte Erstellung des Ausgabeformats nach dem vCard-Standard. Die Klasse lässt sich grundsätzlich testen, aber der Test wird nicht besonders schön. Zunächst müssten unterschiedliche vCard-Eigenschaften gesetzt werden, dann wird die vCard in eine Datei geschrieben, anschließend wird die Datei geöffnet, der Inhalt ausgelesen und zum Schluss auf Korrektheit untersucht. Das ist kein sympathischer Weg! Die Klasse `VCard` ist nicht testorientiert entworfen worden. Warum? Neben der Tatsache, dass so ein Test wegen der Dateizugriffe recht lange dauern könnte, lässt sich prinzipiell festhalten, dass die Methode `export(...)` zwei Verantwortlichkeiten verbindet, nämlich die Ausgabe in dem speziellen vCard-Format und die Ausgabe in eine Datei. Stünde das Prinzip TDD hinter dem Entwurf, so hätte der Autor die Anteile *Format* und *Ausgabe* getrennt. Denn gäbe es eine eigene Methode zur Aufbereitung der Dateien, etwa in einem String, so müsste der Test nur diese Methode aufrufen und bräuchte nicht in einen String zu schreiben. Verbessern wir die Klasse:

**Listing 23.10** com/tutego/insel/junit/util/vdf/v2/VCard.java, VCard

```

public class VCard {

    public String formattedName;
    public String email;

    public void export( Writer out ) throws IOException {

```

```

        out.write( toString() );
    }

public void export( String filename ) throws IOException {
    try ( Writer writer = Files.newBufferedWriter( Paths.get( filename ) ) ) {
        export( writer );
    }
}

@Override public String toString() {
    StringBuilder result = new StringBuilder( "BEGIN:VCARD\n" );
    if ( formattedName != null && ! formattedName.isEmpty() )
        result.append( "FN:" ).append( formattedName ).append( \n );
    if ( email != null && ! email.isEmpty() )
        result.append( "EMAIL:" ).append( email ).append( \n );
    return result.append( "END:VCARD" ).toString();
}
}

```

Diese Variante bringt gleich zwei Verbesserungen mit sich:

1. Die Methode `toString()` liefert nun den nach dem vCard-Standard aufbereiteten String. Der Test muss nun lediglich ein `vCard`-Objekt aufbauen, die Variablen setzen, `toString()` aufrufen und ohne Dateioperationen den String auf Korrektheit testen. Für den Client ändert sich die API aber nicht; er schreibt weiterhin `export(...)`.
2. Direkt in Dateien zu schreiben ist nicht mehr so richtig zeitgemäß. Das berücksichtigt die Klasse und bietet eine überladene Version von `export(...)` mit einem allgemeinen `Writer`. Sollte dann etwa eine vCard über das Netzwerk verschickt werden, ist das kein Problem, und es muss lediglich ein passender `Writer` für das Netzwerkziel übergeben werden. Vorher wäre das sehr umständlich gewesen: Datei erzeugen, Datei auslesen, String versenden.

Im Endeffekt ist der Gewinn groß. Der Test ist performanter, und das Design führt zu bessrem Quellcode: eine Win-win-Situation.

Der gewählte Ansatz zeigt, wie bei Implementierungen zu verfahren ist, die insbesondere mit externen Ressourcen sprechen. Diese gilt es, so weit wie möglich herauszuziehen, wenn nötig auch in einem neuen Typ, der dann als Testimplementierung injiziert werden kann.

## 23.4 Aufbau größerer Testfälle

Bisher haben wir uns mit abgeschlossenen Testfällen beschäftigt und weniger darüber nachgedacht, wie eine größere Anzahl Tests organisiert werden.

### 23.4.1 Fixtures

Eine wichtige Eigenschaft von Tests ist, dass sie voneinander unabhängig sind. Die Annahme, dass ein erster Test zum Beispiel ein paar Testdaten anlegt, auf die dann der zweite Test zurückgreifen kann, ist falsch. Aus dieser Tatsache muss die Konsequenz gezogen werden, dass jede einzelne Testmethode davon ausgehen muss, die erste Testmethode zu sein, und somit ihren Initialzustand selbst herstellen muss. Es wäre aber unnötige Quellcodeduplizierung, wenn jede Testmethode nun diesen Startzustand selbst aufbauen würde. Dieser Anfangszustand heißt *Fixture* (zu Deutsch etwa »festes Inventar«), und JUnit bietet hier vier Annotationen (die sich von Version 4 auf Version 5 geändert haben). Wie sie wirken, zeigt folgendes Beispiel:

**Listing 23.11** com/tutego/insel/junit/util/FixtureDemoTest.java, FixtureDemoTest

```
package com.tutego.insel.junit.util;

import java.util.logging.Logger;
import org.junit.jupiter.api.*;

public class FixtureDemoTest {

    static final Logger log = Logger.getLogger( FixtureDemoTest.class.getName() );

    @BeforeAll
    public static void beforeClass() { log.info( "@BeforeAll" ); }

    @AfterAll
    public static void afterClass() { log.info( "@AfterAll" ); }

    @BeforeEach
    public void setUp() { log.info( "@Before" ); }

    @AfterEach
    public void tearDown() { log.info( "@After" ); }

    @Test
    public void test1() { log.info( "test 1" ); }

    @Test
    public void test2() { log.info( "test 2" ); }
}
```

Die Annotationen beziehen sich auf zwei Anwendungsfälle:

- ▶ @BeforeAll, @AfterAll: Annotiert statische Methoden, die einmal aufgerufen werden, wenn die Klasse für den Test initialisiert wird bzw. wenn alle Tests für die Klasse abgeschlossen sind.
- ▶ @BeforeEach, @AfterEach: Annotiert Objektmethoden, die immer vor bzw. nach einer Testmethode aufgerufen werden.

Läuft unser Beispielprogramm, ist die (verkürzte) Ausgabe daher wie folgt:

```
INFO: @BeforeAll
INFO: @Before
INFO: test 1
INFO: @After
INFO: @Before
INFO: test 2
INFO: @After
INFO: @AfterAll
```

In die @BeforeAll-Methoden wird üblicherweise das reingesetzt, was teuer im Aufbau ist, etwa eine Datenbankverbindung. Die Ressourcen werden dann in der symmetrischen Methode @AfterAll wieder freigegeben, also zum Beispiel werden Datenbankverbindungen wieder geschlossen. Da nach einem Test keine Artefakte vom Testfall bleiben sollen, führen gute @AfterAll/@AfterEach-Methoden sozusagen ein Undo durch.



### Beispiel

Setzt ein System.setProperty(..) »globale« Zustände oder überschreibt es vordefinierte Properties, so ist @BeforeAll ein guter Zeitpunkt, um einen Snapshot zu nehmen und diesen später bei @AfterAll wiederherzustellen:

```
private static String oldValue;
@BeforeAll public static void beforeClass() {
    oldValue = System.getProperty( "property" );
    System.setProperty( "property", "newValue" );
}
@AfterAll public static void afterClass() {
    if ( oldValue != null ) {
        System.setProperty( "property", oldValue );
        oldValue = null;
    }
}
```

### 23.4.2 Sammlungen von Testklassen und Klassenorganisation

Werden die Tests zahlreicher, stellt sich die Frage nach der optimalen Organisation. Als praktikabel hat sich erwiesen, die Testfälle in das gleiche Paket wie die zu testenden Klassen zu setzen, aber den Quellcode physikalisch zu trennen. Entwicklungsumgebungen bieten hierzu Konzepte: So kann Eclipse unterschiedliche Quellcodeordner verwenden, die physikalisch und visuell getrennt sind, aber letztendlich zu Klassendateien im gleichen Paket führen. Nach dem Maven-Standard-Verzeichnis-Layout sind das `src/main/java` und `src/test/java`. Der Vorteil von Typen im gleichen Paket ist, dass oftmals die Paketsichtbarkeit ausreicht und nicht vorher private Eigenschaften nur für Tests öffentlich gemacht werden müssen.

In JUnit lässt sich eine Test-Suite deklarieren und dann über die IDE ausführen. Wir nehmen in unsere POM zwei Abhängigkeiten hinzu:

**Listing 23.12** pom.xml, Ausschnitt

```
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite-api</artifactId>
    <version>1.3.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.3.1</version>
    <scope>test</scope>
</dependency>
```

In einem Paket legen wir eine neue Klasse an, die dann

- ▶ Unterpakete mit Testklassen aufzählt – in diesen nutzen wir die Annotation `@SelectPackages` oder
- ▶ einzelne Testklassen aufzählt – dann mit der Annotation `@SelectClasses`.

Nutzen wir `@SelectPackages` an einem Beispiel. Wenn wir in `com.tutego.insel.junit.util` diverse Testklassen haben, so setzen wir in das übergeordnete Paket `com.tutego.insel.junit` (ohne `util` also) eine Suite `PackageTest`, die das Unterpaket benennt.

**Listing 23.13** com/tutego/insel/junit/util/PackageTest.java

```
package com.tutego.insel.junit;

import org.junit.platform.runner.JUnitPlatform;
```

```
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith( JUnitPlatform.class )
@SelectPackages( "com.tutego.insel.junit.util" )
public class PackageTest { }
```

Die Testklasse ist im Rumpf leer und zählt lediglich über @SelectPackages die Klassen auf.

Üblicherweise besteht ein Projekt aus mehreren hierarchischen Paketen. Hier hilft uns die Tatsache, dass eine Suite selbst eine Art Testfall ist und eine Suite andere Suites benennen kann. Der eigentliche Test wird demnach oben in einer Haupthierarchie gestartet und läuft dann rekursiv alle Unterpakete ab.



In Eclipse reicht es, auf einen Zweig zu gehen und dann im Kontextmenü RUN AS • JUNIT TEST auszuwählen; das läuft dann alle Tests auch in den Unterpaketen ab. Test-Suites ersetzt dies noch nicht, denn Suites werden außerhalb der IDE ausgeführt.

## 23.5 Dummy, Fake, Stub und Mock

Gute objektorientiert entworfene Systeme zeichnen sich dadurch aus, dass es eine hohe Interaktion mit anderen Objekten gibt. Idealerweise zerlegt eine Klasse ein Problem nur bis zu dem Punkt, an dem es sich einer anderen Klasse bedienen kann, die dieses einfachere Problem löst. Schwierig wird es, wenn eine eigene Klasse auf eine andere komplexe Klasse zurückgreift und das Objekt nur dann sinnvoll arbeitet, wenn das referenzierte Objekt da ist und irgendwie sinnvoll antwortet. Diese Abhängigkeit ist ungünstig, denn das Ziel eines guten Tests besteht ja darin, lokal zu sein, also die eigentliche Klasse zu testen und nicht alle referenzierten Klassen um sie herum gleich mit.

In der Praxis begegnen uns drei Hilfskonstrukte, die die Lokalität von Tests ermöglichen:

- ▶ *Fake-Objekte*: Sie sind eine gültige Implementierung einer Schnittstelle, haben aber kein Verhalten, sondern der Rumpf der Methoden ist quasi leer. Sie gibt es nur für die Testfälle. Wenn ein Service etwa auf einen anderen Service zurückgreift, um eine E-Mail mit der einzigen angebotenen Methode void send(String msg, String receiver) zu versenden, kann ein Fake-Objekt diesen E-Mail-Service »implementieren«, aber es muss dazu überhaupt kein Verhalten nachbilden.
- ▶ *Stub-Objekte*: Stub-Objekte implementieren ein bestimmtes Protokoll, sodass sie für den Testfall immer die gleichen Antworten geben können. Wenn etwa der E-Mail-Service eine Methode isTransmitted() anbietet, so kann der Stub immer true liefern. Oder ein Stub-Repository liefert statt Kunden aus der Datenbank immer die gleichen zehn vorgefertigten Kunden. Oder man wartet nicht, bis ein langsamer Webservice-Aufruf die aktuellen Wetterdaten liefert, sondern der Stub gibt vorgefertigte ab. Stubs sind auch praktisch, wenn

zum Beispiel eine GUI-Anwendung programmiert wird, die statt echter Datenbankdaten erst einmal mit den Stubs entwickelt wird und so die Demodaten anzeigt. Wenn ein Team die GUI baut und ein anderes Team den Service, so können beide Gruppen unabhängig arbeiten, und das GUI-Team muss nicht erst auf die Implementierung warten.

- ▶ **Mock-Objekte:** Mock-Objekte sind noch funktionsreichhaltiger als Stubs und bilden auch komplexe Interaktionen ab. In der Regel werden Mock-Objekte durch eine Bibliothek wie *mockito* (<http://mockito.org>) oder *EasyMock* (<http://easymock.org>) »aufgeladen« – sie liefern also nicht wie Stubs immer das gleiche Ergebnis – und zeigen dann das gewünschte Verhalten.

Diese drei Typen können wir unter dem Oberbegriff *Dummy-Objekt* zusammenfassen. Grundsätzlich gilt bei den vier Begriffen aber, dass sie von Autoren nicht einheitlich verwendet werden.<sup>3</sup>

### Mockito-Beispiel

[zB]

Nehmen wir an, alles aus `org.mockito.Mockito.*` ist statisch importiert und wir wollen eine `java.util.List` aufbauen. Dazu muss Mockito erst etwas aufbauen, was sich wie `List` verhält:

```
List mockedList = mock( List.class );
```

Im nächsten Schritt muss das Verhalten der speziellen Liste bestimmt werden:

```
when( mockedList.get(0) ).thenReturn( "tutego" );
```

Anschließend ist die Liste bereit zur Nutzung:

```
System.out.println( mockedList.get(0) ); // tutego
```

## 23.6 JUnit-Erweiterungen, Testzusätze

Das Framework *JUnit* selbst ist recht kompakt, doch wie an den Hamcrest-Matchern abzulesen ist, gibt es die Notwendigkeit für komfortable Testmethoden, die häufig wiederkehrende typische Testaufgaben vereinfachen. Dazu zählen nicht nur Methoden, die testen, ob eine Datei vorhanden ist, sondern auch Unterstützungen für Tests mit Datenbankzugriffen, Webtests oder GUI-Tests.

### Webtests

Beim Testen von Webanwendungen kommen zwei Verfahren zum Einsatz. Das eine ist eine werkzeugunterstützte Aufzeichnung von Webinteraktionen und das automatische Abspielen

---

<sup>3</sup> Die Seite <http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html> stellt einige Autoren mit ihrer Begriffsnutzung vor.

der Folgen für den Test, und das andere ist die programmierte Lösung. Für die Aufzeichnung bietet sich das freie *Selenium* (<http://seleniumhq.org>) bzw. die Integration in Firefox mit der *Selenium IDE* (<http://seleniumhq.org/projects/ide>) an. Wer Tests programmieren möchte, der findet mit dem *Apache HttpUnit* (<http://httpunit.sourceforge.net>) eine gute Basis.

### Tests der Datenbankschnittstelle

Der Zugriff auf die Datenbank geschieht in der Regel über *Repository-Klassen* (auch *DAO-Klassen* genannt). Greift ein Service auf eine Datenbank zu, so geht er immer über das Repository. Der Test des Services wird dadurch vereinfacht, dass statt einer Datenbank-Repository-Implementierung ein Repository-Dummy untergeschoben wird. Bleibt die Frage, wie die Repository-Klassen zu testen sind.

Tests können sehr lange dauern, denn die Interaktion mit der Datenbank ist häufig der langsamste Schritt in einer ganzen Geschäftsanwendung. Eine Herangehensweise ist, die Tests lokal im Speicher laufen zu lassen. Dazu werden In-Memory-Datenbanken wie *Derby*, *H2* oder *HSQLDB* verwendet. Die Datenbank ist also rein im Speicher, und so läuft ein Test sehr schnell. Der größte Nachteil dabei ist aber, dass es SQL-Dialekte gibt, und eine In-Memory-Oracle-Datenbank gibt es bisher nicht. Wenn die Repository-Implementierung für Massenoperationen auf eine gespeicherte Oracle-Prozedur zurückgreift, so kann das das einfache H2 nicht testen.

Eine weitere Aufgabe ist das Füllen der Datenbank mit Testdaten. Die Open-Source-Software *DbUnit* (<http://dbunit.sourceforge.net>) ist hier eine große Hilfe. Externe Daten sind in XML verfasst und können leicht in die Datenbank importiert werden, bevor dann der Test auf diesen Probedaten arbeitet. Die Probedaten werden dann, wenn möglich, in der In-Memory-Datenbank eingefügt oder in einer lokalen Entwicklungsdatenbank. Für fortgeschrittene Tests (und insbesondere zum Abschätzen der Laufzeit) müssen Tests aber auch mit einer Kopie der echten Geschäftsdaten durchgeführt werden.

## 23.7 Zum Weiterlesen

Testen von Anwendungen und testgetriebene Entwicklung ist ein heißes Thema, und der Buchmarkt bietet viel Lektüre. Es lohnt sich auch eine kritische Auseinandersetzung, zum Beispiel angeregt durch TDD Harms Architecture, <http://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html>.

Nicht angesprochen wurden in diesem Kapitel zum Beispiel parametrisierte Tests und Datenpunkte (erwartete Werte kommen automatisch über eine Datenstruktur). JUnit 5 hat einiges über Bord geworfen, sodass Leser genau auf die Versionen schauen sollten, wenn Beispiele aus dem Internet studiert werden.

# Kapitel 24

## 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.

### 24.1 Übersicht

Bei den JDK-Programmen handelt es sich unter anderem um folgende Tools:

Werkzeug	Beschreibung
javac	Java-Compiler zum Übersetzen von <code>.java</code> - in <code>.class</code> -Dateien
java	Java-Interpreter zum Ausführen der Java-Applikationen
javaw	wie java, nur öffnet sich kein Konsolenfenster
jlink	Erzeugen von Laufzeit-Images
javap	Anzeige des Bytecodes einer Klassendatei
jdb	Debugger zum Durchlaufen eines Programms
javadoc	Dienstprogramm zum Erzeugen von Dokumentationen
jar	Archivierungswerkzeug, das Dateien in einem Archiv zusammenfasst
jconsole	Java-Monitoring- und Management-Konsole
serialver	Generiert <code>serialVersionUID</code> .
keytool, jarsigner	Programme zum Einstellen der Sicherheitseigenschaften
jcmd	Diagnose-Kommandos an die JVM schicken
jdeps	Modulabhängigkeiten auflisten

Tabelle 24.1 Werkzeuge vom JDK

Obwohl es versionsabhängig noch weitere Aufrufparameter gibt, sind nur diejenigen aufgeführt, die offiziell in der aktuellen Dokumentation genannt sind.

### 24.1.1 Aufbau und gemeinsame Schalter

Die Kommandozeilenprogramme haben Optionen (auch Schalter genannt) und definieren dadurch unterschiedliche Funktionen und Verhalten. Viele Schalter gibt es in einer Kurz- und Langfassung. Die Kurzfassung beginnt mit genau einem Minuszeichen, wie zum Beispiel `-cp`, und die Langversion mit zwei Minuszeichen, etwa `--class-path`.

Obwohl die Werkzeuge unterschiedliche Funktionen haben, gibt es Gemeinsamkeiten, und die Schalter wurden unter Java 9 vereinheitlicht.<sup>1</sup> Häufig kommen vor:

- ▶ `-help`, `--help` oder `-?` gibt die Hilfe aus.
- ▶ `--module-path` oder `-p` zur Angabe des Modulpfads
- ▶ `--version` oder `-version` zur Ausgabe der Version
- ▶ `-v` oder `--verbose` gibt mehr Meldungen.
- ▶ `-Xmn` und `-Xms` für die Angabe der Heap-Größe

## 24.2 Java-Quellen übersetzen

Ein Java-Compiler übersetzt Java-Quellcode-Dateien in Bytecode. Das JDK liefert standardmäßig den Java-Compiler `javac` mit aus.

### 24.2.1 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
```

---

<sup>1</sup> Siehe <http://openjdk.java.net/jeps/293>

Option	Bedeutung
-classpath <i>Klassenpfad</i> oder -cp <i>Klassenpfad</i>	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) bzw. Doppelpunkt (Unix) trennt mehrere Verzeichnisse.
-d <i>Verzeichnis</i>	Gibt an, wo die übersetzten .class-Dateien gespeichert werden. Ohne Angabe legt der Compiler sie in das gleiche Verzeichnis wie das mit den Quelldateien.
-deprecation	Zeigt die Nutzung von veraltetem Code an.
-g	Erzeugt Debug-Informationen. Die Option muss gesetzt sein, damit der Debugger alle Informationen hat. -g:none erzeugt keine Debug-Informationen, was die Klassendatei etwas kleiner macht.
-nowarn	Deaktiviert die Ausgabe von Warnungen. Fehler (errors) werden noch angezeigt.
-source <i>Version</i>	Erzeugt Bytecode für eine bestimmte Java-Version.
-sourcepath <i>Quellpfad</i>	Ähnlich wie -classpath, nur sucht der Compiler im Quellpfad nach Quelldateien.
-verbose	Ausgabe von Meldungen über geladene Quell- und Klassendateien während der Übersetzung

Tabelle 24.2 Optionen des Compilers javac

### 24.2.2 Alternative Compiler

Neben dem Java-Compiler vom JDK gibt es weitere Compiler, die Java-Quellcode in Bytecode abbilden, doch sie haben keine große Bedeutung, da *javac* frei ist und einfach die »Referenz« bildet. Die Entwicklungsumgebung Eclipse nutzt einen eigenen Java-Compiler (EJC), um flexibel zu sein und einfach während des Speichers und Tippens schon Teile übersetzen zu können.

Wie auch *javac* ist der Eclipse-Compiler selbst in Java implementiert und generiert Bytecode<sup>2</sup>. Es gibt aber auch Java-Compiler in C(++), wie den GNU Compiler for Java (gcj)<sup>3</sup> oder den Jikes-Compiler<sup>4</sup>, doch die beiden letzten Projekte wurden lange nicht aktualisiert und spielen praktisch keine Rolle mehr.

### 24.2.3 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* (kurz AOT). 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>5</sup> angeht.

Existierende Laufzeitumgebungen erreichen mittlerweile eine ausreichende Geschwindigkeit, einen vertretbaren Speicherbrauch und annehmbare Startzeiten. Dennoch ist seit Java 9 ein AOT-Compiler jaotc mit an Bord; er befindet sich wie alle anderen Werkzeuge im bin-Verzeichnis vom JDK. Allerdings haftet dem AOT-Compiler immer noch etwas Experimentelles an; vom Produktiveinsatz rät Oracle ab.

### 24.2.4 Java-Programme in ein natives ausführbares Programm einpacken

Wer Java-Programme vertreibt, weiß um das Problem der JVM-Versionen, Pfade, Start-Icons, Splash-Screens usw. 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 übergibt ihr die Klassen. Es ist also immer noch eine Laufzeitumgebung nötig, doch lassen sich den Java-Programmen Icons mitgeben und Startparameter setzen.

---

<sup>2</sup> Natürlich gibt es da ein Henne-Ei-Problem: Wie sollte ein neuer in Java geschriebener Compiler durch Java übersetzt werden? Daher entsteht der erste Compiler immer in einer anderen Sprache, und die übersetzt eine kleine Teilmenge der Zielsprache, und dann wird ein neuer Compiler in der Minisprache entwickelt. Im nächsten Schritt wachsen und vergrößern sich Grammatik und Compiler. In der Sprache der Compilerbauer heißt der Prozess *Bootstrapping*. Bei Java war das ein Prozess über mehrere Stufen. Patrick Naughton schreibt im Buch »The Java Handbook« dazu: »Arthur van Hoff rewrote the compiler in Oak itself, replacing the C version that James originally wrote.« Und aus Oak wurde später Java.

<sup>3</sup> <https://web.archive.org/web/20120219133307/http://gcc.gnu.org/java/>

<sup>4</sup> <http://tutego.de/go/jikes>

<sup>5</sup> Das Zurückverwandeln von unstrukturiertem Binärkode in Quellcode

Die Open-Source-Software *launch4j* (<http://launch4j.sourceforge.net>) kapselt ein Java-Archiv mit Klassen und Ressourcendateien in ein komprimiertes, ausführbares Programm für Windows, Linux, macOS 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>).

## 24.3 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(String[])`-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`.

### 24.3.1 Schalter der JVM

Diverse Schalter sind bei der Laufzeitumgebung möglich:

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 Klassenpfad</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) bzw. der Doppelpunkt (Unix) trennt mehrere Verzeichnisse.
<code>-DProperty=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.

Tabelle 24.3 Optionen des Interpreters *java*

Option	Bedeutung
-jar	Startet eine Klasse aus dem JAR-Archiv, falls sie in der Manifest-Datei genannt ist. Die Hauptklasse lässt sich aber immer noch angeben.
-verbose	Informationen über die Laufzeitumgebung: ► -verbose:class gibt Informationen über geladene Klassen. ► -verbose:gc informiert über GC-Aufrufe. ► -verbose:jni informiert über native Aufrufe.
-version	Zeigt die aktuelle Version an.
-X	Zeigt nicht standardisierte Optionen an.
-Xdebug	Startet mit Debugger.
-Xincgc	Schaltet die inkrementelle automatische Speicherbereinigung ein.
-Xmsn	Anfangsgröße des Speicherbereichs für die Allokation von Objekten (in MiB), voreingestellt sind 2 MiB.
-Xmxn	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: -Xms128m
-Xnoclasssgc	Schaltet den GC für geladene, aber nicht mehr benötigte Klassen aus.
-Xrs	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.
-Xssn	Setzt die Größe des Stacks.

Tabelle 24.3 Optionen des Interpreters java (Forts.)

**Hinweis**

Je länger es die JVM von Oracle gibt, desto länger wird die Liste der Optionen. Die Standarddokumentation der JVM unter <https://docs.oracle.com/en/java/javase/11/tools/java.html> listet alle Optionen kurz auf.

## Class-Path-Wildcard

Die Option `-cp` erweitert den Klassenpfad durch Java-Archive (*.jar*-Dateien) und einzelne Klassendateien (*.class*-Dateien). Der Class-Path-Wildcard über `*` erlaubt eine noch einfachere Angabe von Java-Archiven. Es empfiehlt sich, die Angaben in Anführungszeichen zu setzen, damit die Shell keine Expansionen vornimmt – es sei denn, das ist gewünscht.

### Beispiel

Füge *log.jar* und alle Java-Archive im Verzeichnis *lib* dem Klassenpfad hinzu:

```
$ java -cp "log.jar;lib/*" MainClass
$ java -cp "log.jar:lib/*" MainClass
```

Windows trennt mit »;«, und Unix trennt »:«.



## Zusatzaoptionen

Mit der Option `-X` lassen sich weitere Schalter setzen und dann der Laufzeitumgebung Zusatzaweisungen geben, etwa über den maximal zu verwendenden Speicher. Ein interessanter Schalter ist `-XshowSettings`, der die Zustände der Standardeigenschaften ausgibt. Das ist sehr nützlich, um etwa abzulesen, welche Pfade gesetzt sind. Angewendet auf das Quadratprogramm aus [Kapitel 1](#), »Java ist auch eine Sprache«, ergibt sich dann:

```
$ java -XshowSettings Quadrat
VM settings:
  Max. Heap Size (Estimated): 884.00M
  Ergonomics Machine Class: client
  Using VM: Java HotSpot(TM) 64-Bit Server VM
Property settings:
  awt.toolkit = sun.awt.windows.WToolkit
  file.encoding = Cp1252
  file.encoding.pkg = sun.io    ...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,
...
Quadrat(1) = 1
Quadrat(2) = 4
Quadrat(3) = 9
Quadrat(4) = 16
```



### Hinweis

Neben den einfachen -X-Optionen gibt es weitere spezielle HotSpot-Optionen, die mit -XX gesetzt werden. Zunächst müssen sie mit -XX:+UnlockDiagnosticVMOptions freigeschaltet werden. Dann lässt sich zum Beispiel mit -XX:+PrintAssembly<sup>6</sup> der von HotSpot generierte Assemblercode ausgeben (allerdings nur, wenn die *hsdis-i386.dll* im Pfad ist).

### 24.3.2 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*.

## 24.4 jlink: der Java Linker

Mit dem Werkzeug *jlink* werden Laufzeitimages produziert. Damit ist es möglich, kleine und kompakte Teilmengen einer Laufzeitumgebung zu bauen, die nur das enthalten, was ein Programm an Modulen auch referenziert. Java-Programme lassen sich dann leicht verteilen, ohne dass die Kunden eine JVM installiert haben müssen.

Die Hilfe von *jlink* dokumentiert den Aufruf:

```
$ jlink --help
Usage: jlink <options> --module-path <modulepath> --add-modules <module>[,<module>...]
Possible options include:
  --add-modules <mod>[,<mod>...]           Root modules to resolve
  --bind-services                                Link in service provider modules and
                                                their dependences
  -c, --compress=<0|1|2>                         Enable compression of resources:
                                                Level 0: No compression
                                                Level 1: Constant string sharing
                                                Level 2: ZIP
  --disable-plugin <pluginname>                 Disable the plugin mentioned
  --endian <little|big>                          Byte order of generated jimage
                                                (default:native)
```

<sup>6</sup> <https://wiki.openjdk.java.net/display/HotSpot/PrintAssembly>

-h, --help	Print this help message
--ignore-signing-information	Suppress a fatal error when signed modular JARs are linked in the image. The signature related files of the signed modular JARs are not copied to the runtime image.
--launcher <name>=<module>[/<mainclass>]	Add a launcher command of the given name for the module and the main class if specified
--limit-modules <mod>[,<mod>...]	Limit the universe of observable modules
--list-plugins	List available plugins
-p, --module-path <path>	Module path
--no-header-files	Exclude include header files
--no-man-pages	Exclude man pages
--output <path>	Location of output path
--save-opt <filename>	Save jlink options in the given file
-G, --strip-debug	Strip debug information
--suggest-providers [<name>,...]	Suggest providers that implement the given service types from the module path
-v, --verbose	Enable verbose tracing
--version	Version information
@<filename>	Read options from file

## 24.5 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 verstehen, 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>7</sup>

#### 24.5.1 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) sowie Methoden und Variablen üblich. Im folgenden Beispiel gibt Javadoc Kommentare für die Klasse, für Attribute und Methoden an:

**Listing 24.1** 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.
     * Decrements the number of persons.
     */
    public void leavePerson() {
        if ( numberOfPersons > 0 )
            numberOfPersons--;
    }
}
```

---

<sup>7</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.

```

/**
 * 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 @see java.lang.String#length()
@version	Version	@version 1.12
@author	Schöpfer	@author Christian Ullensboom
@return	Rückgabewert einer Methode	@return Number of elements
@exception/@throws	Ausnahmen, die ausgelöst werden können	@exception 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-Schriftsatz – auch mit HTML-Sonderzeichen	{@code 1 ist < 2}
{@literal Literale}	Maskiert HTML-Sonderzeichen. Kein Code-Schriftsatz	{@literal 1 < 2 && 2 > 1}

Tabelle 24.4 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.

#### 24.5.2 Mit dem Werkzeug javadoc eine Dokumentation erstellen

Aus dem mit Kommentaren versehenen Quellcode generiert ein externes Programm die Zieldokumente. Das JDK liefert das Konsolenprogramm *javadoc* mit aus, dem als Parameter ein Dateiname der zu kommentierenden Klasse übergeben wird; aus kompilierten 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 eine HTML-Seite ist.



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.

### 24.5.3 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.



### 24.5.4 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 unter anderem Farben und Schriftarten einstellen lassen, die dann alle HTML-Dateien nutzen.
- ▶ `packages.htm` ist eine veraltete Datei. Sie verweist auf die neuen Dateien.

### 24.5.5 Dokumentationskommentare im Überblick \*

Einige Javadoc-Kommentare müssen isoliert hinter der Hauptbeschreibung folgen, wie `@param` (Beschreibung der Parameter) oder `@return` (Beschreibung der Rückgaben). Sie heißen *Block-Tags*. Andere Tags können im Text auftauchen, wie `{@link}` zum Setzen eines Verwei-

ses auf einen anderen Typ oder eine andere Methode. Wir nennen sie *Inline-Tags*. Das Java-doc-Tool erkennt unter anderem die folgenden Tags:

- **Block-Tags:** @author, @deprecated, @exception, @param, @return, @see, @serial, @serialData, @serialField, @since, @throws, @version, @apiNote, @implSpec, @implNote
- **Inline-Tags:** {@code}, {@docRoot}, {@inheritDoc}, {@link}, {@linkplain}, {@literal}, {@value}

### Beispiele

Eine externe Zusatzquelle geben wir wie folgt an:

```
@see <a href="spec.html#section">Java Spec</a>.
```

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,...)
@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;
```

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 ==}.
 */
```

#### 24.5.6 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 (schon etwas angestaubte) Website <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 sind.

#### 24.5.7 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 ab 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(...).

Es ist ungünstig, die Methoden jetzt einfach zu löschen, weil es dann zu Compilerfehlern kommt. Eine Lösung wäre daher, die Methode oder den Konstruktor als deprecated zu deklarieren. @deprecated ist ein eigener Dokumentationskommentar. Sein Einsatz sieht dann etwa folgendermaßen aus (Ausschnitt aus der Klasse java.util.Date):

**Listing 24.2** java.util.Date.java, Ausschnitt

```
/*
 * 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 bzw. 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 24.3** src/main/java/com/tutego/insel/tools/OldSack.java

```
package com.tutego.insel.tool;

//@SuppressWarnings( "deprecation" )
public class OldSack {
    java.util.Date d = new java.util.Date( 62, 3, 4 );
}
```

Jetzt rufen wir ganz normal den Compiler auf:

```
$ javac com/tutego/insel/tool/OldSack.java
Note: com\tutego\insel\tool\OldSack.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Der Compiler sagt uns, dass der Schalter `-deprecation` weitere Hinweise gibt:

```
$ javac -deprecation com.tutego.insel.tool.OldSack.java
com\tutego\insel\tool\OldSack.java:5: warning: [
  deprecation] Date(int,int,int) in Date has been deprecated
    java.util.Date d = new java.util.Date( 62, 3, 4 );
                           ^
1 warning
```

Die Ausgabe gibt genau die Zeile mit der veralteten Anweisung an; Alternativen nennt der Compiler 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` markieren (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

Annotationen sind eine Art zusätzlicher Modifizierer. Die 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 Rückgabewert 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.

In Java 9 wurde der Annotationstyp `@Deprecated` um zwei Eigenschaften erweitert:

- ▶ `String since() default ""`: Dokumentiert die Version, seit der das Element veraltet ist.
- ▶ `boolean forRemoval() default false`: Zeigt an, dass das Element in Zukunft gelöscht werden soll.

## [zB]

### Beispiel

Ein Beispiel aus der Thread-Klasse:

```
@Deprecated(since="1.2", forRemoval=true)
public final synchronized void stop(Throwable obj) {
    throw new UnsupportedOperationException();
}
```

### Veraltete Bibliotheken

Veraltetes hat sich im Laufe der Zeit genug angesammelt. Eine Übersicht liefert <https://docs.oracle.com/en/java/javase/11/docs/api/deprecated-list.html>.

### 24.5.8 Javadoc-Überprüfung mit DocLint

Das Javadoc-Tool kann in den Javadoc-Kommentaren Fehler aufspüren; aktiviert wird diese Prüfung mit dem Schalter `Xdoclint`. DocLint erkennt folgende Gruppen von Fehlern, die auch als Option angegeben werden können:

Gruppe	Beschreibung
accessibility	Prüft auf Probleme mit der Zugänglichkeit der Dokumentation, dass etwa Tabellen immer eine Zusammenfassung besitzen.
html	Erkennt HTML-Fehler, wie fehlende schließende spitze Klammern.
missing	Prüft auf fehlende Elemente, wenn zum Beispiel ein Kommentar fehlt oder @return.
reference	Prüft alle Referenzen in den Javadoc-Tags, etwa bei @see oder auch ungültige Namen bei @param.
syntax	Prüft allgemeine Syntaxfehler wie nicht ausmaskierte HTML-Zeichen wie <, > oder & und nicht existierende Javadoc-Tags.

Tabelle 24.5 Mögliche Gruppen bei Javadoc und dem DocLint-Werkzeug

Zu weiteren Optionen der Javadoc-Erweiterung siehe <https://docs.oracle.com/en/java/javase/11/tools/javadoc.html>.

## 24.6 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 7-Zip 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.

### 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 sich jeder Zertifikate ausstellen lassen kann, auch unter dem Namen Mickey Mouse.<sup>8</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.

---

<sup>8</sup> Obwohl dieser schon vergeben ist; doch vielleicht ist »Darkwing Duck« ja noch frei.

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. Beispiele sind die 3D- und Java-Mail-API. Eigene Pakete sehen also so aus, als gehörten sie zum Standard.

### 24.6.1 Das Dienstprogramm jar benutzen

*jar* ist ein Kommandozeilenprogramm und verfügt über verschiedene Optionen, 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.

#### 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.



#### Beispiel

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
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 %)
```

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. 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 0 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.

### Beispiel

[zB]

Erstellen wir das *sliders*-Archiv neu mit zusätzlichen Sound-Dateien.

```
$ jar cfv0 slider.jar Slider.class images -C sounds
```

Zweierlei ist neu: Zum einen komprimiert *jar* nicht mehr (der Schalter 0 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 Archivdatei lassen sich später mit u (für engl. *update*) noch Dateien hinzufügen.

### Beispiel

[zB]

Nehmen wir an, es kommt eine Bilddatei hinzu, so schreiben wir:

```
$ jar vuf slider.jar images/buchsbaum.gif
```

## JAR-Dateien betrachten

Die zusammengepackten Dateien zeigen die Option tf an.

### Beispiel

[zB]

```
$ 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 24.6.2, »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. Mit dem Schalter `f` geben wir den Dateinamen auf der Kommandozeile an und nicht von der Standardeingabe etwa über eine Pipe.



### Beispiel

Der Schalter `v` (für engl. *verbose*) gibt 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` (für engl. *file*) das Gleiche wie beim Anzeigen: Ohne den Schalter erwartet `jar` die Archivdatei 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.



### Beispiel

Wir wollen jetzt nur die Grafiken aus unserem Archiv `slider.jar` auspacken. Dazu schreiben wir:

```
$ jar vxf slider.jar images\*
extracted: images\volti.gif
```

Mit der Option `v` sehen wir, was genau `jar` entpackt. Sonst erfolgt keine Ausgabe auf der Konsole.

### 24.6.2 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 wichtige Zusatzinformationen für ein Archiv, wie die Signatur, die für jede Datei aufgeführt ist.



## Beispiel

Sehen wir uns einmal die Manifest-Datei an, die sich ergibt.

```
$ jar cvf slider.jar Slider.class images/volti.gif
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: 9zeehlViDy0fpfvOKkPECiMYvHO=
MD5-Digest: qv913K1ZFt5tdPr2BjatIg==
```

Die Einträge im Manifest erinnern an eine Property-Datei, denn auch hier gibt es immer Schlüssel und Werte, die durch einen Doppelpunkt voneinander getrennt sind:

### 24.6.3 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 Ressourcendateien ausliefern müssen, sondern nur eine einzige Datei. Ein anderer Vorteil ist, dass ein Betriebssystem wie Windows oder macOS 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(String[])` welcher Klasse sie aufrufen soll, ist eine kleine Notiz mit dem Schlüssel `Main-Class` in der Manifest-Datei nötig:

`Main-Class: voll.qualifizierter.Klassenname.der.Klasse.mit.main`

Dies ist sehr angenehm für den Benutzer eines Archivs, denn nun ist der Hersteller für den Eintrag des Einstiegspunktes 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(String[])`-Methode mitzuteilen. Vor der Erzeugung eines Archivs erstellen wir eine Textdatei, die wir hier `MainfestMain.txt` nennen wollen, mit dem Eintrag `Main-Class`:

**Listing 24.4 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**

Der Schalter `e` (für engl. *endpoint*) bestimmt direkt die ausführbare 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. 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 auf dem Desktop als ausführbare Programme und startet sie selbstständig mit `java -jar`.

**Hinweis**

`java` (oder `javaw`) ignoriert die Angaben über `-cp` bzw. Einträge in der Umgebungsvariablen `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. Zur Installation betrachte <https://sourceforge.net/p/fjep/discussion/396532/thread/4edb64ed>. Üblicherweise erstellt Maven die Fat-JAR.

**24.7 Zum Weiterlesen**

Die Webseiten mit den Beschreibungen der Dienstprogramme sind umfangreich und auf jeden Fall aufmerksam zu lesen.

# Anhang

## Java SE-Module und Paketübersicht

»Einer der Vorteile der Unordentlichkeit liegt darin, dass man dauernd tolle Entdeckungen macht.«  
– Alan Alexander Milne (1882–1956)

### A.1 Alle Module von Java 11

#### A.1.1 Java SE-Module

Die API der *Java Platform, Standard Edition* (»Java SE«) besteht aus folgenden Modulen, die alle mit dem Präfix `java` beginnen.

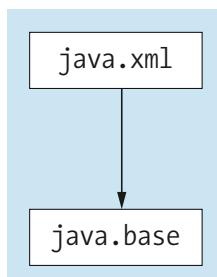
Modul	Beschreibung
<code>java.base</code>	Fundamentale Typen der Java SE-Plattform
<code>java.compiler</code>	Java-Sprachmodell, Annotationsverarbeitung, Java Compiler API
<code>java.datatransfer</code>	API für den Datentransfer zwischen Applikationen, in der Regel die Zwischenablage
<code>java.desktop</code>	Grafische Oberflächen mit AWT und Swing, Accessibility-API, Audio, Drucken und JavaBeans
<code>java.instrument</code>	Intrumentalisierung ist die Veränderung der Java-Programme zur Laufzeit.
<code>java.logging</code>	Logging-API
<code>java.management</code>	Java Management Extensions (JMX)
<code>java.management.rmi</code>	RMI-Connector für den Remote-Zugriff auf die JMX-Beans
<code>java.naming</code>	Java Naming and Directory Interface (JNDI) API
<code>java.net.http</code>	HTTP- und WebSocket-Client

Tabelle A.1 Module der Java SE

Modul	Beschreibung
java.prefs	Die Preferences API dient zum Speichern von Benutzereinstellungen.
java.rmi	Entfernte Methodenaufrufe; Remote Method Invocation (RMI) API
java.scripting	Scripting API
java.security.jgss	Java-Binding der IETF Generic Security Services API (GSS-API)
java.security.sasl	Java-Unterstützung für IETF Simple Authentication and Security Layer (SASL)
java.sql	JDBC API für den Zugriff auf relationale Datenbanken
java.sql.rowset	JDBC RowSet API
java.transaction.xa	Unterstützung verteilter Transaktionen in JDBC
java.xml	XML-Klassen: Java API for XML Processing (JAXP), Streaming API for XML (StAX), Simple API for XML (SAX), W3C Document Object Model (DOM) API
java.xml.crypto	API für XML-Kryptografie

Tabelle A.1 Module der Java SE (Forts.)

Das `java.base`-Modul ist das wichtigste Modul, und es enthält Kernklassen wie `Object` und `String` usw. Es ist das einzige Modul, das selbst keine Abhängigkeit zu anderen Modulen entält. Jedes andere Modul jedoch bezieht sich mindestens auf `java.base`. Die Javadoc stellt das schön grafisch dar (siehe [Abbildung A.1](#)).

Abbildung A.1 Das Modul `java.xml` hat eine Abhängigkeit zum `java.base`-Modul.

Zum Teil gibt es mehr Abhängigkeiten, etwa beim Modul `java.desktop` (siehe [Abbildung A.2](#)).

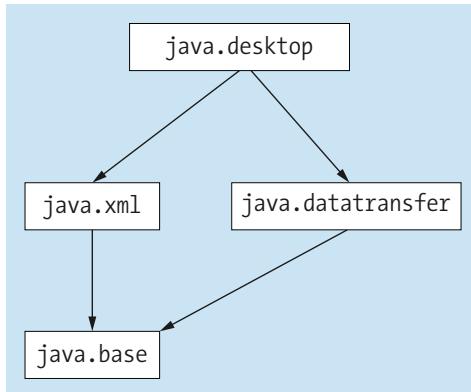


Abbildung A.2 Abhängigkeiten vom Modul »java.desktop«

### A.1.2 JDK-Modul

Das JDK ist die Standard-Implementierung der Java SE. Es liefert den Entwicklern weitere Pakete und Klassen, etwa mit einem HTTP-Server, und Java-Werkzeuge wie Compiler und Javadoc-Tool. Es gibt mehrere Module, die alle mit dem Präfix `jdk` beginnen.

Modul	Beschreibung
<code>jdk.accessibility</code>	Hilfsklassen zur Zugänglichkeit
<code>jdk.attach</code>	Attach-API, zur Verbindung mit der JVM
<code>jdk.charsets</code>	Weitere Zeichenkodierungen
<code>jdk.compiler</code>	Compiler-Implementierung
<code>jdk.crypto.cryptoki</code>	SunPKCS11 Security-Provider
<code>jdk.crypto.ec</code>	SunEC Security-Provider
<code>jdk.dynalink</code>	Unterstützung für dynamisches Linken von Operationen
<code>jdk.editpad</code>	Editor-Service, zum Beispiel für JShell
<code>jdk.hotspot.agent</code>	Implementierung vom HotSpot Serviceability Agent
<code>jdk.httpserver</code>	Eingebauter HTTP-Server
<code>jdk.jartool</code>	Werkzeuge für JAR-Dateien, wie <code>jar</code> und <code>jarsigner</code>
<code>jdk.javadoc</code>	Implementierung des Werkzeugs Javadoc

Tabelle A.2 JDK-Module

Modul	Beschreibung
jdk.jcmd	Diagnose-Tools wie <i>jcmd</i> , <i>jps</i> , <i>jstat</i>
jdk.jconsole	Implementierung von <i>jconsole</i>
jdk.jdeps	Implementierung von <i>jdeps</i> und <i>javap</i>
jdk.jdi	Java Debug Interface
jdk.jdwp.agent	Java Debug Wire Protocol
jdk.jfr	API für den JDK Flight Recorder
jdk.jlink	Definiert das Java Linker Tool, <i>jlink</i> .
jdk.jshell	Implementierung der JShell
jdk.jsobject	Definiert die API für den Zugriff auf JavaScript-Objekte.
jdk.jststad	Werkzeug <i>jstatd</i>
jdk.localedata	Weitere Sprachen neben US-Englisch*
jdk.management	JDK-spezifische Management-Interfaces für die JVM
jdk.management.agent	JMX Management-Agent
jdk.management.jfr	Management-Interface für Flight Recorder
jdk.naming.dns	DNS Java Naming Provider
jdk.naming.rmi	RMI Java Naming Provider
jdk.net	JDK-spezifische Netzwerk-API
jdk.pack	Unterstützung des Pack200-Formats
jdk.rmic	RMI-Compiler
jdk.scripting.nashorn	JavaScript-Laufzeitumgebung Nashorn (ECMAScript 5.1)
jdk.sctp	JDK-spezifische API für SCTP
jdk.security.auth	Implementierung der Schnittstellen aus <i>javax.security.auth.*</i> und Authentifikationsmodule
jdk.security.jgss	Java-Erweiterungen zur GSS-API und Implementierung des SASL-GSSAPI-Mechanismus

Tabelle A.2 JDK-Module (Forts.)

Modul	Beschreibung
jdk.xml.dom	Definiert eine Teilmenge der <i>W3C Document Object Model</i> (DOM) API, die nicht Teil der Java SE-API ist.
jdk.zipfs	Dateisystem-Provider für ZIP-Dateien
* <a href="http://www.oracle.com/technetwork/java/javase/documentation/java9locales-3559485.html">http://www.oracle.com/technetwork/java/javase/documentation/java9locales-3559485.html</a>	

Tabelle A.2 JDK-Module (Forts.)

### A.1.3 Java Smart Card I/O-Modul

Ein Modul außerhalb der Java SE und des JDK verbleibt: `java.smartcardio`. Es enthält die Java Smart Card I/O API und dient dem Ansprechen von Kartenlesern. Nur das Paket `javax.smartcardio` wird exportiert.

## A.2 Pakete der Java SE-Module

Es folgt eine Übersicht über alle Pakete von Java SE 11. Sie enthalten rund 4.400 Typen. In Java 1.0 verteilten sich 212 Klassen auf 8 Pakete; damals war es übersichtlicher. Die wichtigen Pakete sind fett hervorgehoben. Nicht genannt sind SPI-Pakete (SPI steht für *Service Provider Implementation*); sie definieren Verhalten für Anbieter bestimmter Funktionalitäten.

Nicht aufgeführt sind die Pakete aus den Modulen JDK und Java Smart Card I/O.

### A.2.1 `java.base`

Exportiertes Paket	Beschreibung
<code>java.io</code>	Möglichkeiten zur Ein- und Ausgabe. Dateien werden als Objekte repräsentiert. Datenströme erlauben den sequenziellen Zugriff auf die Dateiinhalte.
<code>java.lang</code>	Automatisch eingebundenes Paket. Enthält unverzichtbare Klassen wie String-, Thread- oder Wrapper-Klassen.
<code>java.lang.annotation</code>	Zur Deklaration eigener Annotationstypen
<code>java.lang.invoke</code>	Spezielle Typen mit Sonderbehandlung in der JVM
<code>java.lang.module</code>	Modul-Deskriptor aufbauen, Objektmodell

Exportiertes Paket	Beschreibung
java.lang.ref	Behandelt Referenzen.
java.lang.reflect	Mit Reflection ist es möglich, Klassen und Objekte über sich erzählen zu lassen.
java.math	Beliebig lange Ganzzahlen oder Fließkommazahlen
<b>java.net</b>	Kommunikation über Netzwerke. Bietet Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP/UDP mit dem Internet verbinden lassen.
java.nio	Neue IO-Implementierung (daher NIO) für performante Ein- und Ausgabe. NIO steht ja für <i>New IO</i> . Eine Aktualisierung ist NIO.2.
java.nio.channels	Datenkanäle für nicht blockierende Ein- und Ausgabeoperationen
java.nio.charset	Kodierungen für die Übersetzung zwischen Byte- und Unicode-Zeichen
<b>java.nio.file</b>	NIO.2-Dateisysteme, Datei- und Pfadzugriff
java.nio.file.attribute	Datei- und Dateisystemattribute
java.security	Klassen und Schnittstellen für Sicherheit
java.security.acl	Unwichtig, da sie durch Klassen in java.security ersetzt wurden. Daher deprecated und wird in der Zukunft gelöscht.
java.security.cert	Analysieren und Verwalten von Zertifikaten, Pfaden und Rückruf (Verfall) von Zertifikaten
java.security.interfaces	Schnittstellen für RSA- und DSA-Schlüssel
java.security.spec	Parameter der Schlüssel und Algorithmen für die Verschlüsselung
<b>java.text</b>	Unterstützung für internationalisierte Programme. Behandlung von Text, Formatierung von Datums- werten und Zahlen
<b>java.time</b>	Kernklassen aus der Date-Time-API
java.time.chrono	Nicht-ISO-Kalender der Date-Time-API

Exportiertes Paket	Beschreibung
java.time.format	Formatieren von Werten aus der Date-Time-API
java.time.temporal	Zugriff auf Basiseigenschaften von temporalen Klassen
java.time.zone	Zeitzonen aus der Date-Time-API
<b>java.util</b>	Datenstrukturen, Raum und Zeit sowie Teile der Internationalisierung, Zufallszahlen
java.util.concurrent	Hilfsklassen für nebenläufiges Programmieren, etwa Thread-Pools
java.util.concurrent.atomic	Atomare Operationen auf Variablen
java.util.concurrent.locks	Lock-Objekte zum Sperren kritischer Bereiche
java.util.function	Funktionale Schnittstellen
java.util.jar	Zugriffe auf Dateien im Archiv-Format JAR (Java Archive)
java.util.regex	Unterstützung von regulären Ausdrücken
<b>java.util.stream</b>	Stream-API
java.util.zip	Zugriff auf komprimierte Daten mit GZip und Archive (ZIP)
javax.crypto	Klassen und Schnittstellen für kryptografische Operationen
javax.crypto.interfaces	Schnittstellen für Diffie-Hellman-Schlüssel
javax.crypto.spec	Klassen und Schnittstellen für Schlüssel und Parameter zur Verschlüsselung
javax.net	Klassen mit einer Socket-Fabrik
javax.net.ssl	SSL-Verschlüsselung
javax.security.auth	Framework für Authentifizierung und Autorisierung
javax.security.auth.callback	Informationen wie Benutzernamen oder Passwort vom Server beziehen
javax.security.auth.login	Framework für die Authentifizierungsdienste

Exportiertes Paket	Beschreibung
javax.security.auth.x500	Für X.509-Zertifikate, X.500 Principal und X500 PrivateCredential
javax.security.cert	Public-Key-Zertifikate

#### A.2.2 java.compiler

Exportiertes Paket	Beschreibung
javax.annotation.processing	Schnittstellen für Annotation Processors
javax.lang.model	Eine Aufzählung für Java-Versionen
javax.lang.model.element	Repräsentiert Elemente der Java-Sprache (Methode, Annotation ...).
javax.lang.model.type	Repräsentiert Java-Typen (Fehlertyp, Referenztyp usw.).
javax.lang.model.util	Utility-Klassen für Programmelemente und Typen
javax.tools	Ansprachen von Java-Tools; im Moment nur der Compiler

#### A.2.3 java.datatransfer

Exportiertes Paket	Beschreibung
java.awt.datatransfer	Informationsaustausch zwischen (Java-)Programmen über die Zwischenablage des Betriebssystems

#### A.2.4 java.desktop

Exportiertes Paket	Beschreibung
java.applet	Stellt Klassen für Java-Applets bereit, damit diese auf Webseiten ihr Leben führen können.
java.awt	Das Paket AWT ( <i>Abstract Windowing Toolkit</i> ) bietet Klassen zur Grafikausgabe und zur Nutzung von grafischen Bedienoberflächen.
java.awt.color	Unterstützung von Farträumen und Farbmodellen

Exportiertes Paket	Beschreibung
java.awt.desktop	Interaktion mit dem Desktop, neu in Java 9
java.awt.dnd	Drag & Drop zum Übertragen oder Manipulieren von Informationen unter grafischen Oberflächen
java.awt.event	Schnittstellen für die verschiedenen Ereignisse unter grafischen Oberflächen
java.awt.font	Klassen, mit denen Schriftarten genutzt und modifiziert werden können
java.awt.geom	Paket für die Java 2D-API, die ähnlich wie im Grafikmodell von PostScript und PDF affine Transformationen auf beliebige 2D-Objekte anwenden kann
java.awt.im	Klassen für alternative Eingabegeräte
java.awt.image	Erstellen und Manipulieren von Rastergrafiken
java.awt.image.renderable	Klassen und Schnittstellen zum allgemeinen Erstellen von Grafiken
java.awt.print	Bietet Zugriff auf Drucker und kann Druckaufträge erzeugen.
java.beans	JavaBeans definieren wiederverwendbare Komponenten auf der Client-Seite, die beim Programmieren visuell konfiguriert werden können.
java.beans.beancontext	Beans lassen sich in einem Bean-Kontext zusammenbringen.
javax.accessibility	Schnittstellen zwischen Eingabegeräten und Benutzerkomponenten
javax.imageio	Schnittstellen zum Lesen und Schreiben von Bilddateien in verschiedenen Formaten
javax.imageio.event	Ereignisse, die während des Ladens und Speicherns bei Grafiken auftauchen
javax.imageio.metadata	Unterstützung für beschreibende Metadaten in Bilddateien
javax.imageio.plugins.bmp	Klassen, die das Lesen und Schreiben von BMP-Bilddateien unterstützen
javax.imageio.plugins.jpeg	Lesen und Schreiben von JPEG-Bildern

Exportiertes Paket	Beschreibung
javax.imageio.plugins.tiff	Lesen und Schreiben von TIFF-Bildern
javax.imageio.stream	Unterstützt das Einlesen und Schreiben von Bildern durch die Behandlung der unteren Ebenen.
javax.print	Java Print Service API
javax.print.attribute	Attribute (wie Anzahl der Seiten, Ausrichtung) beim Java Print Service
javax.print.attribute.standard	Druckerattribute
javax.print.event	Ereignisse beim Drucken
javax.sound.midi	Ein- und Ausgabe, Synthesisierung von MIDI-Daten
javax.sound.sampled	Schnittstellen zur Ausgabe und Verarbeitung von Audiodaten
<b>javax.swing</b>	Einfache Swing-Komponenten
javax.swing.border	Grafische Rahmen für die Swing-Komponenten
javax.swing.colorchooser	Anzeige vom JColorChooser, einer Komponente für die Farbauswahl
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	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

Exportiertes Paket	Beschreibung
javax.swing.text	Unterstützung für Textkomponenten
javax.swing.text.html	HTMLEditorKit zur Anzeige und Verwaltung eines HTML-Texteditors
javax.swing.text.html.parser	Einlesen 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

### A.2.5 java.instrument

Exportiertes Paket	Beschreibung
java.lang.instrument	Bezieht der Klassenlader eine Klasse, so wird der Bytecode nicht direkt der JVM übergeben, sondern vorher modifiziert. Das nennt sich <i>Instrumentalisierung</i> .

### A.2.6 java.logging

Exportiertes Paket	Beschreibung
java.util.logging	Protokollieren von Programmabläufen

### A.2.7 java.management

Exportiertes Paket	Beschreibung
java.lang.management	Management-Interfaces zum Monitoring und Management einer JVM oder anderer Komponenten
javax.management	Zentrale Typen für Java Management Extensions (JMX)
javax.management.loading	Fortgeschrittenes dynamische Laden ermöglichen

Exportiertes Paket	Beschreibung
javax.management.modelmbean	Beschreibung von Model-MBeans
javax.management.monitor	Monitorklassen beobachten Model-Beans.
javax.management.openmbean	Definition von Open Data Types und der Open MBean
javax.management.relation	Verbindet MBeans im MBean-Server.
javax.management.remote	Remote-Zugriff auf einen JMX-MBean-Server
javax.management.timer	Timer-MBean meldet nach Zeitablauf Ereignisse.

#### A.2.8 java.management.rmi

Exportiertes Paket	Beschreibung
javax.management.remote.rmi	Über RMI Remote-Zugriff auf MBean-Server

#### A.2.9 java.naming

Exportiertes Paket	Beschreibung
javax.naming	Zugriff auf Namensdienste
javax.naming.directory	Zugriff auf Verzeichnisdienste, erweitert das javax.naming-Paket.
javax.naming.event	Ereignisse, wenn sich etwas beim Verzeichnisdienst ändert
javax.naming.ldap	Unterstützung von LDAPv3-Operationen

#### A.2.10 java.net.http

Exportiertes Paket	Beschreibung
java.net.http	HTTP- und WebSocket-Client

### A.2.11 java.prefs

Exportiertes Paket	Beschreibung
java.util.prefs	Verwalten von Benutzer- und Systemeigenschaften

### A.2.12 java.rmi

Exportiertes Paket	Beschreibung
java.rmi	Aufruf von Methoden auf entfernten Rechnern
java.rmi.activation	Unterstützung für die RMI-Aktivierung, wenn Objekte auf ihren Aufruf warten
java.rmi.dgc	Der verteilte Garbage-Collector DGC ( <i>Distributed Garbage Collection</i> )
java.rmi.registry	Zugriff auf den Namensdienst unter RMI, die Registry
java.rmi.server	Die Serverseite von RMI
javax.rmi.ssl	Mit SSL mehr Sicherheit bei RMI-Verbindungen

### A.2.13 java.scripting

Exportiertes Paket	Beschreibung
javax.script	Scripting-API zum Einbinden von Skriptsprachen

### A.2.14 java.security.jgss

Exportiertes Paket	Beschreibung
javax.security.auth.kerberos	Unterstützung von Kerberos zur Authentifizierung in Netzwerken
org.ietf.jgss	Framework für Sicherheitsdienste wie Authentifizierung, Integrität, Vertraulichkeit

**A.2.15 java.security.sasl**

Exportiertes Paket	Beschreibung
javax.security.sasl	Unterstützung für SASL (Simple Authentication and Security Layer)

**A.2.16 java.sql**

Exportiertes Paket	Beschreibung
java.sql	Zugriff auf relationale Datenbanken über SQL
javax.sql	Datenquellen auf Serverseite
javax.transaction.xa	Typen für verteilte Transaktionen

**A.2.17 java.sql.rowset**

Exportiertes Paket	Beschreibung
javax.sql.rowset	Implementierung von RowSet
javax.sql.rowset.serial	Mappt SQL-Typen auf serialisierbare Java-Typen.

**A.2.18 java.transaction.xa**

Exportiertes Paket	Beschreibung
javax.transaction.xa	Unterstützung verteilter Transaktionen in JDBC

**A.2.19 java.xml**

Exportiertes Paket	Beschreibung
javax.xml	Konstanten aus der XML-Spezifikation
javax.xml.catalog	Catalog-API, neu in Java 9
javax.xml.datatype	Schema-Datentypen für Dauer und gregorianischen Kalender
javax.xml.namespace	QName für den Namensraum

Exportiertes Paket	Beschreibung
javax.xml.parsers	Einlesen von XML-Dokumenten
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.xpath	XPath-API
org.w3c.dom	Klassen für die Baumstruktur eines XML-Dokuments nach DOM-Standard
org.w3c.dom.bootstrap	Enthält Fabrik zum Erfragen einer DOM-Implementation.
org.w3c.dom.events	DOM-Events
org.w3c.dom.ls	Laden und Speichern von XML-Strukturen
org.w3c.dom.ranges	Unterstützung für DOM Level 2 Range
org.w3c.dom.traversal	Unterstützung für DOM Level 2 Traversal
org.w3c.dom.views	Für DOM-Ansichten
org.xml.sax	Ereignisse, die beim Einlesen eines XML-Dokuments nach dem SAX-Standard auftreten
org.xml.sax.ext	Zusätzliche Behandlungs routinen für SAX2-Ereignisse
org.xml.sax.helpers	Adapterklassen und Standardimplementierungen

### A.2.20 java.xml.crypto

Exportiertes Paket	Beschreibung
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 spezielle Unterpakete dom, keyinfo und spec.

## A.3 java.lang-Paket

Die folgenden Schnittstellen, Klassen und Aufzählungen deklariert das Paket `java.lang` (die Ausnahmen und Error-Klassen werden in [Kapitel 8](#), »Ausnahmen müssen sein«, erklärt, die fünf Annotationen in [Kapitel 3](#), »Klassen und Objekte«). Generische Typen sind erkennbar.

Schnittstelle	Beschreibung
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<T>	Erlaubt das Vergleichen.
Iterable<T>	Kann einen Iterator liefern.
ProcessHandle/ProcessHandle.Info	Identifiziert native Prozesse.
Readable	Liefert aus einer Ressource Zeichen oder Zeichenfolgen.
Runnable	Programmcode, den ein Thread starten kann
StackWalker.StackFrame	StackFrame ist das Ergebnis vom StackWalker.

Tabelle A.3 Schnittstellen im Paket »java.lang«

Schnittstelle	Beschreibung
System.Logger	Logger für die internen Java-Bibliotheken
Thread.UncaughtExceptionHandler	An den Thread gehängt, fängt er Laufzeitfehler ab.

**Tabelle A.3** Schnittstellen im Paket »java.lang« (Forts.)

Abstrakte Klassen sind kursiv dargestellt.

Klasse	Beschreibung
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<T>	Typen in der Laufzeitumgebung
ClassLoader	Klassenlader
ClassValue<T>	Verbindet einen Wert mit einem Klassentyp.
Compiler	Nur für den JIT-Compiler nötig, veraltet
Double	Wrapper-Klasse für double
enum<E extends Enum<E>>	Basisklasse für Aufzählungen
Float	Wrapper-Klasse für float
InheritableThreadLocal<T>	Verbindet Werte mit einem Thread.
Integer	Wrapper-Klasse für int
Long	Wrapper-Klasse für long
Math	Utility-Klasse für numerische Operationen
Module	Laufzeitmodul, neu seit Java 9
ModuleLayer	Ebenen von Modulen, neu seit Java 9
ModuleLayer.Controller	Kontrolliert Module in den Ebenen; neu in Java 9.

**Tabelle A.4** Klassen im Paket »java.lang«

Klasse	Beschreibung
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
Runtime.Version	Versionen verwalten
RuntimePermission	Rechte mit Laufzeiteigenschaften
SecurityManager	Sicherheitsmanager
Short	Wrapper-Klasse für short
StackTraceElement	Element für den Stack-Trace
StackWalker	Abläufen des Aufruf-Stacks
StrictMath	Numerische Operationen strikt gerechnet
String	Immutable Zeichenketten
StringBuffer	Veränderbare, nicht threadsichere Zeichenketten
StringBuilder	Veränderbare, threadsichere Zeichenketten
System	Utility-Klasse mit diversen Klassenmethoden
System.LoggerFinder	Systemlogger anlegen, verwalten, konfigurieren
Thread	Nebenläufige Programme
ThreadGroup	Gruppert Threads.
ThreadLocal<T>	Verbindet Werte mit einem Thread.
Throwable	Basisotyp für Ausnahmen
Void	Spezieller Typ für void-Rückgabe

Tabelle A.4 Klassen im Paket »java.lang« (Forts.)

Aufzählung	Beschreibung
Character.UnicodeScript	Unicode-Skripte, also Zeichenfamilien
ProcessBuilder.Redirect.Type	Art der Umleitung bei externen Prozessen
StackWalker.Option	Konfiguration vom StackWalker
System.Logger.Level	Level vom System-Logger
Thread.State	Thread-Status wie WAITING, BLOCKED

Tabelle A.5 Aufzählungen im Paket »java.lang«



# Index

^, logischer Operator .....	153
-, Subtraktionsoperator .....	142
!, logischer Operator .....	153
? , Generics .....	762
... variable Argumentliste .....	284
.class .....	645, 904
.NET Core .....	68
.NET Framework .....	68
@After .....	1212
@AfterClass .....	1212
@author, Javadoc .....	1227
@Before .....	1212
@BeforeClass .....	1212
@code, Javadoc .....	1227
@Deprecated, Annotation .....	721, 1233
@deprecated, Javadoc .....	1232
@exception, Javadoc .....	1227
@FunctionalInterface .....	721, 794
@Ignore .....	1207
@link, Javadoc .....	1227
@linkplain, Javadoc .....	1227
@literal, Javadoc .....	1227
@Override .....	486, 512, 646, 721
@param, Javadoc .....	1227
@return, Javadoc .....	1227
@SafeVarargs .....	734
@see, Javadoc .....	1227
@SuppressWarnings .....	721
@Test .....	1197
@throws, Javadoc .....	1227
@version, Javadoc .....	1227
@XmlElement, Annotation .....	1128
@XmlRootElement, Annotation .....	1125
*, Multiplikationsoperator .....	142
*? .....	50
/, Divisionsoperator .....	142
//, Zeilenkommentar .....	112
&, Generics .....	755
&&, logischer Operator .....	153
&amp .....	1110
&apos .....	1110
&gt .....	1110
&lt .....	1110
&quot .....	1110
#ifdef .....	63
#IMPLIED .....	1114
#REQUIRED .....	1114
%, Modulo-Operator .....	142, 144, 1158
%%, Formatspezifizierer .....	391
%b, Formatspezifizierer .....	391
%c, Formatspezifizierer .....	391
%d, Formatspezifizierer .....	391
%e, Formatspezifizierer .....	391
%f, Formatspezifizierer .....	391
%n, Formatspezifizierer .....	391
%s, Formatspezifizierer .....	391
%t, Formatspezifizierer .....	391
%x, Formatspezifizierer .....	391
+, Additionsoperator .....	142
=, Zuweisungsoperator .....	141
==, Referenzvergleich .....	265, 647
, logischer Operator .....	153
\$, innere Klasse .....	626, 633
1.1.1970 .....	922
<b>A</b>	
Abdeckung, Test .....	1196
Abrunden .....	1156
abs(), Math .....	1154
Absolutwert .....	174
Abstract Window Toolkit .....	1036
abstract, Schlüsselwort .....	503, 504
Abstrakte Klasse .....	502
Adapter .....	991
-add-exports, Schalter .....	879, 882
Addition .....	142
-add-modules, Schalter .....	882
addPropertyChangeListener(), PropertyChangeSupport .....	864
Adjazenzmatrix .....	291
Adobe Flash .....	71
Ahead-of-time-Compiler .....	1220
Akkumulator .....	223
Aktives Warten .....	954
Aktor .....	232
Algebra boolesche .....	153
lineare .....	1192
Alias .....	260
Allgemeiner Konstruktor .....	438
American Standard Code for Information Interchange .....	317
Amigos .....	231

Android .....	74
Anführungszeichen .....	138
Angepasster Exponent .....	1153
Annotation .....	485
Anonyme innere Klasse .....	631
Anpassung .....	862
Anweisung .....	111
<i>atomare</i> .....	115
<i>elementare</i> .....	115
<i>geschachtelte</i> .....	171
<i>leere</i> .....	114
Anweisungssequenz .....	115
Anwendungsfall .....	232
Anwendungsfalldiagramm .....	232
ANY .....	1112
Aonix Perc Pico .....	77
Apache Commons CLI .....	313
Apache Commons Lang .....	647
Apache Harmony .....	70
Apache Maven .....	933
Apache NetBeans .....	89
append(), StringBuilder/StringBuffer .....	362
Appendable, Schnittstelle .....	364, 1086
Applet .....	51
Applikations-Klassenlader .....	874, 907
Äquivalenz .....	153
Archetype, Maven .....	937
Arcus-Funktion .....	1164
Argument .....	114
<i>der Funktion</i> .....	206
<i>optionales</i> .....	217
Argumentanzahl, variable .....	283
Ariane 5, Absturz .....	1140
ArithmeticException .....	143, 580, 1179
Arithmetischer Operator .....	142
ARM-Block .....	603
Array .....	269
<i>mehrdimensionales</i> .....	285
arraycopy(), System .....	294
Array-Grenze .....	60
ArrayListOutOfBoundsException .....	580
ArrayList, Klasse .....	461, 979, 988
ArrayStoreException .....	493, 996
Array-Typ .....	237
ASCII .....	317
ASCII-Zeichen .....	104
asin(), Math .....	1164
asList(), Arrays .....	305, 991
Assert, Klasse .....	1198
assert, Schlüsselwort .....	617
Assertion .....	617, 618
AssertionError .....	618, 1198
assertThat(), Assert .....	1201
assertXXX(), Assert .....	1198
Assignment .....	141
Assoziation .....	457
<i>reflexive</i> .....	458
<i>rekursive</i> .....	458
<i>zirkuläre</i> .....	458
Attribut .....	229, 230, 1099, 1108
Aufgeschobene Initialisierung .....	218
Aufruf, kaskadierter .....	241
Aufrunden .....	1156
Aufzählungstyp .....	430
Ausdruck .....	118
<i>purer</i> .....	848
Ausdrucksanweisung .....	119, 142
Ausführungsstrang .....	940
Ausgabeformatierung .....	922
Ausmaskierung .....	356
Ausnahme .....	60
<i>geprüfte</i> .....	559, 581
<i>geschachtelte</i> .....	599
<i>nicht geprüfte</i> .....	581
Ausprägung .....	229
Ausprägungsspezifikation .....	232
Ausprägungsvariable .....	238
Auszeichnungssprache .....	1107
Äußere Schleife .....	189
Autoboxing .....	163, 699
autobuild .....	88
Automatic Resource Management (ARM) .....	603
Automatic-Module-Name .....	891
Automatische Speicherbereinigung .....	422, 434
Automatische Speicherfreigabe .....	59
Automatisches Modul .....	888
AWT .....	1036

## B

Basic Multilingual Plane (BMP) .....	325
Bedingte Compilierung .....	63
Bedingung, zusammengesetzte .....	168
Bedingungsoperator .....	156, 173
Beispielprogramme der Insel .....	43
Benutzeroberfläche, grafische .....	1035
Beobachter-Pattern .....	856
Betrag .....	1154
Betriebssystemunabhängigkeit .....	53
Bezeichner .....	104
Beziehung, bidirektionale .....	457
Bias .....	1153

biased exponent .....	1153	Callable, Schnittstelle .....	968
Bidirektionale Beziehung .....	457	CamelCase-Schreibweise .....	106
Big Endian .....	1177	Cast .....	155, 158
BigDecimal, Klasse .....	1175, 1184	catch, Schlüsselwort .....	554
BigInteger, Klasse .....	1176	CDATA .....	1113
billion-dollar mistake .....	838	ceil(), Math .....	1156
Binärer Name .....	908	Central Repository, Maven .....	936
<i>innerer Typ</i> .....	626	char, Datentyp .....	125, 138, 1136
Binärer Operator .....	140	charAt(), String .....	336
Binär-Kompatibilität .....	528	CharSequence, Schnittstelle .....	333, 370
Binärrepräsentation .....	378	checked exception .....	581
Binärsystem .....	1137	class literal .....	645
Binary Floating-Point Arithmetic .....	1148	Class Loader .....	870
binarySearch(), Arrays .....	303	Class, Klasse .....	644, 904
Bindung, späte dynamische .....	495	class, Schlüsselwort .....	397
Binnenmajuskel .....	106	ClassCastException .....	478, 580
bin-Pfad .....	82	ClassLoader, Klasse .....	908
Bitweises exklusives Oder .....	1134	ClassNotFoundException .....	904, 906
Bitweises Nicht .....	1134	CLASSPATH .....	875, 1221, 1240
Bitweises Oder .....	1134	-classpath .....	874, 1221
Bitweises Und .....	1134	Classpath-Hell .....	875
Block .....	121	Class-Path-Wildcard .....	1223
<i>leerer</i> .....	122	Cleanable, Schnittstelle .....	669
BMP (Basic Multilingual Plane) .....	325	Cleaner, Klasse .....	669
BOM (Byte Order Mark) .....	326	Clipping-Bereich .....	1055
Booch, Grady .....	231	clone(), Arrays .....	293
boolean, Datentyp .....	125	clone(), Object .....	653
Boolean, Klasse .....	697	Cloneable, Schnittstelle .....	654
Boolesche Algebra .....	153	CloneNotSupportedException .....	654, 656
Bootstrap-Klasse .....	874, 875, 907	Closeable, Schnittstelle .....	1090
bound property .....	862, 864	Closure .....	788
Boxing .....	643, 699	COBOL .....	32
break, Schlüsselwort .....	192, 193	Codepage .....	326
Bruch .....	1192	Codepoint .....	104, 317
Bruchzahl .....	1148	Codeposition .....	317
Brückenmethode .....	777	Collator, Klasse .....	675
busy waiting .....	954	Command not found .....	86
Byte .....	1133	Comparable, Schnittstelle .....	516, 674, 690
Byte Order Mark → BOM .....		Comparator, Schnittstelle .....	674
byte, Datentyp .....	125, 135, 1136	compare(), Comparator .....	676
Byte, Klasse .....	688	compare(), Wrapper-Klassen .....	690
Bytecode .....	52	compareTo(), Comparable .....	676
BYTES, Wrapper-Konstante .....	694	compareTo(), String .....	344
		compareIgnoreCase(), String .....	344
<b>C</b>		Compilation Unit .....	112, 254
C .....	49	Compiler .....	83
C++ .....	49, 229	<i>inkrementeller</i> .....	88
Calendar, Klasse .....	922	<i>nativer</i> .....	1220
Call by Reference .....	264	Compilerfehler .....	85
Call by Value .....	206, 263	Compilierung, bedingte .....	63
		CompletionService, Schnittstelle .....	975

compound assignment operator,	
Verbundoperator .....	149
Connector/J .....	1101
const, Schlüsselwort .....	265
const-korrekt .....	265
constraint property .....	862
contains(), String .....	337
contentEquals(), String .....	368
continue .....	195
Copy-Konstruktor .....	245, 440, 653
copyOf(), Arrays .....	302
copyOfRange(), Arrays .....	302
CopyOnWriteArrayList, Klasse .....	979
cos(), Math .....	1163
cosh(), Math .....	1164
-cp .....	874, 1221, 1223
Currency, Klasse .....	1191
currentThread(), Thread .....	944
currentTimeMillis(), System .....	901, 923
Cursor, Iterator .....	993
Customization .....	862

**D**

-D .....	913
Dalvik Virtual Machine .....	74
dangling pointer .....	445
Dangling-else-Problem .....	169
Data Hiding .....	411
Database Management System (DBMS) .....	1099
DataInput, Schnittstelle .....	1076
DataOutput, Schnittstelle .....	1076
DatatypeConverter, Klasse .....	381
Date, Klasse .....	922
DateFormat, Klasse .....	922
Dateinamenendung .....	345
Datenbankausprägung .....	1100
Datenbankschema .....	1100
Datenbanksystem, relationales .....	1102
Datenbankverwaltungssystem .....	1099
Datenbasis .....	1099
Datenstrukturen .....	979
Datentyp .....	122
ganzzahliger .....	135
Datenzeiger .....	1077
Datumswerte .....	922
DB2 .....	1101
DBMS (Database Management System) .....	1099
DbUnit .....	1216
Deadlock .....	941
deep copy .....	656
deepEquals(), Arrays .....	298, 663
deepHashCode(), Arrays .....	663
default, switch-case .....	177
Default-Konstruktor .....	436, 666
Default-Paket .....	253
Deklarative Programmierung .....	817
Dekonstruktor .....	445
Dekrement .....	155
delegate .....	66
delete() .....	60
delete(), StringBuffer/StringBuilder .....	366
Delimiter .....	384
Dependency Inversion Principle (DIP) .....	551
Deployment .....	874
deprecated .....	1219, 1232
-deprecation .....	1233
Dereferenzierung .....	58, 165
Design by Contract .....	617
Design-Pattern .....	853
Destruktor .....	666
Dezimalpunkt .....	1148
Dezimalsystem .....	1137
Diamantoperator .....	735
Diamanttyp .....	735
digit(), Character .....	331
DIP (Dependency Inversion Principle) .....	551
DirectX .....	66
Disjunktion .....	153
Dispatcher, Betriebssystem .....	940
Dividend .....	143, 145, 1160
Division .....	142
Rest .....	1159
Divisionsoperator .....	143
Divisor .....	143, 145, 1160
Doc Comment .....	1226
Doclet .....	1231
DOCTYPE .....	1115
Document Object Model (DOM) .....	1121
Document Type Definition (DTD) .....	1111
DocumentBuilderFactory .....	1124
Dokumentationskommentar .....	1226
Dokumenttypdefinition .....	1111
DOM (Document Object Model) .....	1121
Doppelklammer-Initialisierung .....	635
double brace initialization .....	635
double, Datentyp .....	125, 1148
Double, Klasse .....	688
doubleToLongBits(), Double .....	664, 1153
do-while-Schleife .....	184
Drei-Wege-Vergleich .....	676
DRY .....	549

DTD (Document Type Definition) .....	1111
Duck-Typing .....	201
Dummy-Objekt .....	1215
<b>E</b>	
e(fx)clipse .....	1040
e(fx)clipse, Eclipse-Plugin .....	1042
-ea, Schalter .....	618, 1221
EasyMock .....	1215
Echtzeit-Java .....	76
Eclipse .....	87
Eclipse Translation Pack .....	90
ECMAScript .....	66
eFace .....	1040
Effektiv final .....	630, 800
Eigenschaft .....	861
<i>einfache</i> .....	862
<i>eingeschränkte</i> .....	862
<i>gebundene</i> .....	862, 863
<i>indizierte</i> .....	862
<i>objektorientierte</i> .....	54
<i>statische</i> .....	420
Einfache Eigenschaft .....	862
Einfaches Hochkomma .....	138
Einfachvererbung .....	467
Eingeschränkte Eigenschaft .....	862
Element, XML .....	1108
else, Schlüsselwort .....	169
Elternklasse .....	465
EmptyStackException .....	580
enable assertions .....	618
Endlosschleife .....	183
Endrekursion .....	220, 222
endsWith(), String .....	345
ENGLISH, Locale .....	919
ensureCapacity(), List .....	989
Enterprise Edition .....	75
Entität .....	1099, 1110
Entwurfsmuster .....	853
Enum, Klasse .....	710
enum, Schlüsselwort .....	429, 617
EOFException .....	1075
equals(), Arrays .....	298, 663
equals(), Object .....	266, 647, 648
equals(), String .....	368
equals(), StringBuilder/StringBuffer .....	370
equals(), URL .....	652
equalsIgnoreCase(), String .....	342, 343
Ereignis .....	862
Ergebnistyp .....	200
Erreichbar, catch .....	615
Erreichbarer Quellcode .....	208
Erreichbarkeit, Modulsystem .....	889
Error, Klasse .....	570, 581
Erweiterte for-Schleife .....	279
Erweiterte Schnittstelle .....	529
Erweiterungsklasse .....	465
Erweiterungsmethode, virtuelle .....	529
Escape-Sequenz .....	322
Escape-Zeichen .....	338
Escher, Maurits .....	226
Eulersche Zahl .....	1154
Euro-Zeichen .....	323
Event .....	862
EventListener, Schnittstelle .....	857
EventObject, Klasse .....	857
Exception .....	60
Exception, Klasse .....	570
Exception-Handler .....	557
ExceptionInInitializerError .....	449
Exception-Parameter .....	557
Exception-Transparenz .....	805
Execute-around-Method-Muster .....	824
Executor, Schnittstelle .....	965
ExecutorCompletionService, Klasse .....	975
ExecutorService, Schnittstelle .....	965
Exemplar .....	229
Exemplarinitialisierer .....	451
Exemplarinitialisierungsblock .....	634
Exemplarvariable .....	238, 419
exit(), System .....	313
Explizites Klassenladen .....	871
Exponent .....	1152
<i>angepasster</i> .....	1153
Exponentialwert .....	1162
Expression .....	118
extends, Schlüsselwort .....	464, 521
eXtensible Markup Language .....	1108
externe Iteration .....	1010
<b>F</b>	
Fabrik .....	853
Fabrikmethode .....	854
Factory .....	853
Faden .....	940
Fake-Objekt .....	1214
Fakultät .....	1183
Fall-through .....	178
FALSE, Boolean .....	697
false, Schlüsselwort .....	125

fastutil	998
Fee, die gute	219
Fehler	571
Fehlercode	553
Fehlermeldung, non-static method	205
Feld	269
<i>nichtprimitives</i>	281
<i>nichtrechteckiges</i>	289
<i>zweidimensionales</i>	286
Feldtyp	237
Fencepost-Error	187
FileNotFoundException	572
FileSystem, Klasse	1063
fill(), Arrays	300
fillInStackTrace(), Throwable	595
final, Schlüsselwort	218, 421, 426, 489, 490
Finale Klasse	489
Finale Methode	490
Finale Werte	454
finalize(), Object	666
Finalizer	666
finally, Schlüsselwort	566
FindBugs	588
findClass(), ClassLoader	908
firePropertyChange(), PropertyChangeSupport	864
fireVetoableChange(), VetoableChangeSupport	867
First Person, Inc.	50
Fixture	1211
Flache Kopie, clone()	656
Flache Objektkopie	656
Fließkommazahl	123, 133, 1148
Fließpunktzahl	1148
float, Datentyp	125, 1148
Float, Klasse	688
floatToIntBits(), Float	664
floor(), Math	1156
Fluchtsymbol	322
Flushable, Schnittstelle	1091
forDigit(), Character	331
for-each-Loop	182
format(), PrintWriter/PrintStream	391
format(), String	390
Formatspezifizierer	390
Format-String	390
forName(), Class	904
for-Schleife <i>erweiterte</i>	186 279
Fortschaltausdruck	187
Fragezeichen-Operator	140
FRANCE, Locale	919
free()	60
FRENCH, Locale	919
Funktion, variadische	283
Funktionale Schnittstelle	789
Funktionsdeskriptor	789
Future, Schnittstelle	970
FutureTask, Klasse	972
FXGraph	1040
FXML	1040
<b>G</b>	
Ganzzahl	123
Garbage-Collector	59, 236, 422, 434, 444
Gaußsche Normalverteilung	1172
GC → Garbage-Collector	
Gebundene Eigenschaft	862, 863
Geltungsbereich	216
Generics	730
Generische Methode	739
Generischer Typ	730
Geprüfte Ausnahme	559, 581
GERMAN, Locale	919
GERMANY, Locale	919
Geschachtelte Ausnahme	599
Geschachtelte Top-Level-Klasse	624
Geschachtelter Typ	623
get(), List	980
getBoolean(), Boolean	686
getChars(), String	349
getClass(), Object	904
getInteger(), Integer	686
getNumericValue(), Character	331
getPriority(), Thread	963
getProperties(), System	911
getResource()	1096
getResourceAsStream()	1096
Getter	413, 863
ggT	1176
Gleichheit	266, 267
Gleichwertigkeit	266
Gleitkommazahl	1148
Globale Variable	216
Glyph	320
GNU Classpath	70
GNU Trove	998
Goal, Maven	937
Gosling, James	49
goto, Schlüsselwort	195
Grafische Benutzeroberfläche	1035

Grammatik .....	103
Granularität, Threads .....	964
Graphical User Interface .....	1035
Graphics, Klasse .....	1055
Graphics2D, Klasse .....	1055
Green-OS .....	50
Green-Projekt .....	50
Green-Team .....	50
Groovy .....	53
Groß-/Kleinschreibung .....	105, 345, 351
Größter gemeinsamer Teiler .....	1176
GUI (Graphical User Interface) .....	1035
Gültigkeit, XML .....	1111
Gültigkeitsbereich .....	216
<b>H</b>	
Hamcrest .....	1201
Hangman .....	340
Harmony, Apache .....	70
Hashcode .....	658
hashCode(), Arrays .....	663
hashCode(), Object .....	658
Hashfunktion .....	658
HashSet, Klasse .....	1003
Hashwert .....	658
hasNextLine(), Scanner .....	387
Hauptklasse .....	112
Header-Datei .....	63
Heap .....	235
Hexadezimale Zahl .....	1137
Hexadezimalrepräsentation .....	378
Hexadezimalsystem .....	1137
Hilfsklasse .....	437
Hoare, Tony .....	838
Hochkomma, einfaches .....	138
HotJava .....	51
HotSpot .....	56
HSQldb .....	1101
HTML .....	1107
HttpUnit .....	1216
Hyperbolicus-Funktionen .....	1164
<b>I</b>	
Ich-Ansatz .....	230
Identifizierer .....	104
Identität .....	267, 647
identityHashCode(), System .....	660, 665
IEEE 754 .....	133, 145, 1148
IEEEremainder(), Math .....	1158
if-Anweisung .....	166
angehäufte .....	172
if-Kaskade .....	172
Ignorierter Statusrückgabewert .....	558
IKVM.NET .....	67
IllegalArgumentException .....	580, 582, 585, 587, 589
IllegalMonitorStateException .....	581
IllegalStateException .....	585
IllegalThreadStateException .....	946
Imagination .....	49
immutable .....	332
Imperative Programmiersprache .....	111
Implikation .....	153
Implizites Klassenladen .....	871
import, Schlüsselwort .....	250
Import, statischer .....	255
Index .....	269, 275
indexed property .....	862
Indexierte Variable .....	274
indexOf(), String .....	338
IndexOutOfBoundsException .....	276, 586
Indizierte Eigenschaft .....	862
Infinity .....	1148
Inhalt, XML-Element .....	1108
Initiales Modul .....	887
Initialisierung, aufgeschobene .....	218
Inkrement .....	155
Inkrementeller Compiler .....	88
Inline-Tag .....	1230
Innere Schleife .....	189
Innerer Typ .....	623
InputStream, Klasse .....	1081
instanceof, Schlüsselwort .....	507
Instanz .....	229
Instanzinitialisierer .....	451
Instanzvariable .....	238
int, Datentyp .....	125, 135, 1136
Integer, Klasse .....	688
Integrationstest .....	1196
IntelliJ IDEA .....	88
Interaktionsdiagramm .....	232
Interface .....	65, 502, 510
Interface Segregation Principle (ISP) .....	550
interface, Schlüsselwort .....	510
Interface-Typ .....	237
Intermediäre Operation .....	1011, 1028
Interne Iteration .....	708, 1010
Interrupt .....	955
interrupt(), Thread .....	956
interrupted(), Thread .....	958

InterruptedException .....	953, 957
Interval .....	191
Introspection .....	861
Invariant, Generics .....	760
IOException .....	559, 572
isInterrupted(), Thread .....	956
is-Methode .....	414
isNaN(), Double/Float .....	1150
ISO 8859-1 .....	104, 319
ISO Country Code .....	918
ISO Language Code .....	918
ISO/IEC 8859-1 .....	318
ISO-639-Code .....	918
ISO-Abkürzung .....	921
ISP (Interface Segregation Principle) .....	550
Ist-eine-Art-von-Beziehung .....	502
ITALIAN, Locale .....	919
Iterable, Schnittstelle .....	707, 708
Iteration, interne .....	708
Iterator, Schnittstelle .....	703
iterator(), Iterable .....	707

**J**

J/Direct .....	66
J2ME .....	74
Jacobson, Ivar .....	231
Jakarta EE .....	75
JamaicaVM .....	77
JAPAN, Locale .....	919
JAPANESE, Locale .....	919
JAR .....	1235
jar, Dienstprogramm .....	1236
-jar, java .....	1240
JAR-Datei .....	872
jarsigner, Dienstprogramm .....	1217
Java .....	50
Java 2D-API .....	1059
Java API for XML Parsing .....	1123
Java Card .....	75
Java Community Process (JCP) .....	900
Java Database Connectivity (JDBC) .....	1102
Java Document Object Model .....	1122
Java EE .....	75
Java Foundation Classes .....	1036
Java Foundation Classes (JFC) .....	1059
Java ME .....	74
Java Mission Control, Software .....	903
Java Plug-in .....	71
Java Runtime Environment .....	1239
Java SE .....	70
Java TV .....	75
Java Virtual Machine .....	52
java, Dienstprogramm .....	1221
java, Paket .....	249
java.awt.geom, Paket .....	1059
java.lang.ref, Paket .....	669
java.nio.file, Paket .....	1062
java.util.jar, Paket .....	1236
Java-Archiv .....	872
JavaBeans .....	861
javac, Dienstprogramm .....	1218
Javadoc .....	1226
javadoc, Dienstprogramm .....	1228
JavaFX .....	1036
JavaFX Scene Builder .....	1041
JavaFX Script .....	71
JavaFX-Plattform .....	71
JavaScript .....	66
Java-Security-Modell .....	57
JavaSoft .....	51
javaw, Dienstprogramm .....	1224
javax, Paket .....	249, 900
javax.xml.bind.annotation, Paket .....	1125
JAXB .....	1124
JAXBContext, Klasse .....	1129
JAXP .....	1123
JCP (Java Community Process) .....	900
JDBC (Java Database Connectivity) .....	1102
JDBC 2.0 Optional Package API .....	1103
JDBC 2.1 core API .....	1103
JDBC-Versionen .....	1103
JDOM .....	1122
JFC .....	1036
JFrame, Klasse .....	1057
Jigsaw .....	876
JIT .....	56
jlink .....	1224
join(), Thread .....	961
JOptionPane, Klasse .....	558
Joy, Bill .....	49
JPMS (Java Platform Module System) .....	876
JProfiler, Software .....	903
JRE .....	1239
JRuby .....	53
JSmooth .....	1221
JSR (Java Specification Request) .....	70
JSR 203 .....	1062
JSR 354: Money and Currency API .....	1190
JUnit .....	1194
Just-in-time-Compiler .....	56

JXPath .....	1131	Konkrete Klasse .....	503
Jython .....	53	Konstante	
		<i>symbolische</i> .....	426
		<i>vererbte</i> .....	522
		Konstantenpool .....	358
		Konstruktor .....	245
		<i>allgemeiner</i> .....	438
		<i>parameterloser</i> .....	434
		<i>parametrisierter</i> .....	438
		<i>Vererbung</i> .....	469
		Konstruktoraufruf .....	235
		Konstruktorreferenz .....	813
		Konstruktor-Weiterleitung .....	470
		Kontravalenz .....	153
		Kontrollstruktur .....	166
		Koordinaten, Maven .....	934
		Kopf .....	200
		Kopfdefinition .....	1110
		Kopie	
		<i>flache</i> .....	656
		<i>tiefe</i> .....	656
		KOREA, Locale .....	919
		KOREAN, Locale .....	919
		Kosinus .....	1164
		Kotlin .....	89
		Kovariant, Array .....	996
		Kovariant, Generics .....	760
		Kovarianter Rückgabetyp .....	491
		Kovariantes Überschreiben .....	779
		Kovarianz bei Arrays .....	492
		Kreiszahl .....	1154
		Kurzschluss-Operator .....	154
<hr/>			
		L	
		Lambda-Ausdruck .....	787
		lastIndexOf(), String .....	339
		Latin-1 .....	318
		Laufzeitumgebung .....	52
		launch4j .....	1221
		Lebensdauer .....	216
		Leeres XML-Element .....	1109
		Leerraum entfernen .....	353
		Leer-String .....	356, 359
		Leertaste .....	329
		Leerzeichen .....	329
		length(), String .....	335
		Lesbarkeit, Modul .....	889
		LESS-Prinzip .....	769
		Lexikalik .....	103
		line.separator .....	915

Lineare Algebra .....	1192
Lineare Kongruenz .....	1171
lineSeparator(), System .....	915
LinkedHashSet, Klasse .....	1008
LinkedList, Klasse .....	979, 989
Linking .....	870
Linksassoziativität .....	157
Liskov Substitution Principle (LSP) .....	550
Liskov, Barbara .....	479
Liskovsches Substitutionsprinzip .....	479
List, Schnittstelle .....	979
Liste .....	979
verkettete .....	989
Listener .....	856
ListIterator, Schnittstelle .....	993
Literal .....	107
loadClass(), ClassLoader .....	908
Locale .....	919
Locale, Klasse .....	352, 917
log(), Math .....	1162
log4j .....	930
Logischer Operator .....	153
Log-Level .....	932
Lokale Klasse .....	629
long, Datentyp .....	125, 135, 1136
Long, Klasse .....	688
longBitsToDouble(), Double .....	1153
Lower-bounded Wildcard-Typ .....	763
LSP (Liskov Substitution Principle) .....	550
LU-Zerlegung .....	1192

## M

magic number .....	427
Magische Zahl .....	427
main() .....	86, 113
Main-Class .....	1239
Main-Thread .....	944
Makro .....	63
MANIFEST.MF .....	1238
Man-in-the-Middle-Angriff .....	57
Mantelklasse .....	683
Mantisse .....	1152
Marke .....	195
marker interface .....	547
Markierungsschnittstelle .....	547
Marshaller, Schnittstelle .....	1129
Maschinenzeit .....	926
Matcher, Hamcrest .....	1202
Math, Klasse .....	1154
MathContext, Klasse .....	1187
MAX_PRIORITY, Thread .....	963
MAX_RADIX .....	331
max(), Math .....	1155
Maximum .....	174, 1155
McNealy, Scott .....	50
Mehrdimensionales Array .....	285
Mehrfachvererbung .....	515
Mehrfachverzweigung .....	171
member class .....	624
memory leak .....	445
Menschenzeit .....	926
MESA, Programmiersprache .....	49
Metadaten .....	485
META-INF/MANIFEST.MF .....	1238
Meta-Objekt .....	904
Methode .....	199, 229
finale .....	490
generische .....	739
native .....	65
parametrisierte .....	205
rekursive .....	219
statische .....	204
synthetische .....	626, 781
überdeckte .....	497
überladene .....	116, 214
überschreiben .....	482
verdeckte .....	497
Methodenaufruf .....	114, 203, 402
Methodenkopf .....	200
Methodenreferenz .....	810
Methodenrumpf .....	200
Micro Edition .....	74
Microsoft Development Kit .....	66
Microsoft SQL Server .....	1102
MIN_PRIORITY, Thread .....	963
MIN_RADIX .....	331
MIN_VALUE .....	1165
min(), Math .....	1155
Minimum .....	174, 1155
Minus, unäres .....	146, 155
Mitgliedsklasse .....	624
Mixin .....	538
mockito .....	1215
Mock-Objekt .....	1215
Model-View-Controller .....	856
Modifizierer .....	121, 202
Modul	
automatisches .....	888
initiales .....	887
unbenanntes .....	889
Modulo .....	145

Modulpfad .....	874
Moneta .....	1190
multicast .....	66
multi-catch .....	577
Multilevel continue .....	195
Multiplikation .....	142
Multiplizität .....	457
Multitasking .....	940
Multitaskingfähig .....	940
MySQL .....	1100
 <b>N</b>	
Nachkomme, Vererbung .....	465
name(), Enum .....	711
Namensraum, XML .....	1118
NaN .....	143
<i>signaling</i> .....	1152
<i>stilles</i> .....	1151
NaN (Not a Number) .....	1148, 1165
nanoTime(), System .....	901
narrowing conversion .....	160
Native Methode .....	65
Nativer Compiler .....	1220
natural ordering .....	674
Natürliche Ordnung .....	674
Naughton, Patrick .....	50
NavigableSet, Schnittstelle .....	1005
Nebeneffekt .....	402
Nebenläufig .....	939
NEGATIVE_INFINITY .....	1165
Negatives Vorzeichen .....	140
nested exception .....	599
nested type .....	623
NetBeans .....	89
Netscape .....	66
new line .....	915
new, Schlüsselwort .....	235, 434
nextLine(), Scanner .....	387
Nicht .....	153
<i>bitweises</i> .....	1134
Nicht geprüfte Ausnahme .....	581
Nichtprimitives Feld .....	281
NIO.2 .....	1062
no-arg-constructor .....	245, 434
NoClassDefFoundError .....	906
Normalverteilung .....	1172
<i>gaußsche</i> .....	1172
NoSuchElementException .....	704
Not a Number (NaN) .....	1148, 1165
Notation .....	231
<i>wissenschaftliche</i> .....	1152
nowarn .....	1219
NULL .....	553
null, Schlüsselwort .....	256
nullary constructor .....	434
NullPointerException .....	258, 276, 581, 587, 836
Null-Referenz .....	257
Null-String .....	359
Number, Klasse .....	688
NumberFormatException .....	376, 555, 558
Numeric promotion .....	143
Numerische Umwandlung .....	143
 <b>O</b>	
Oak .....	50
Oberklasse .....	465
Obfuscator .....	906
Object Management Group (OMG) .....	231, 900
Object, Klasse .....	467, 644
Objective-C .....	65
Objects, Klasse .....	670
Objektansatz .....	230
Objektdiagramm .....	232
Objektgleichheit .....	647
Objektgraph .....	445
Objektidentifikation .....	645
Objektkopie, flache .....	656
Objektorientierter Ansatz .....	65
Objektorientierung .....	54, 120
Objekttyp .....	58, 477
Objektvariable .....	238
<i>initialisieren</i> .....	446
ODBC .....	1102
Oder .....	153
<i>ausschließendes</i> .....	153
<i>bitweises</i> .....	156, 1134
<i>exklusives</i> .....	153
<i>exklusives bitweises</i> .....	1134
<i>logisches</i> .....	156
Off-by-one-Error .....	187
Oktalsystem .....	1137
Oktalzahlrepräsentation .....	378
Olson, Ken .....	1193
OMG .....	231
OMG (Object Management Group) .....	900
OO-Methode .....	231
Open/closed principle .....	550
OpenJDK .....	63, 77
OpenJFX .....	71, 896, 1038

Operation	
<i>intermediäre</i>	1011
<i>reduzierende</i>	1011
<i>terminale</i>	1011
Operator	141
<i>arithmetisches</i>	142
<i>binärer</i>	140
<i>einstelliger</i>	140
<i>Fragezeichen-</i>	140
<i>logischer</i>	153
<i>relationaler</i>	150
<i>ternärer</i>	173
<i>trinärer</i>	173
<i>überladener</i>	62
<i>unärer</i>	140
<i>zweistelliger</i>	140
operator precedence	155
Operator-Rang	155
Operatorrangfolge	155
Optional, Klasse	838
OptionalDouble, Klasse	841
OptionalInt, Klasse	841
OptionalLong, Klasse	841
Oracle Corporation	51
Oracle Database	1101
Oracle JDK	78
ordinal(), Enum	714
Ordinalzahl, Enum	713
Ordnung, natürliche	674
OutOfMemoryError	236, 582, 654
OutputStream, Klasse	1079
 <b>P</b>	
package, Schlüsselwort	252
paint(), Frame	1054
paintComponent()	1057
Paket	249
<i>unbenanntes</i>	253
Paketsichtbarkeit	415
Parallel	939
Parallele Präfix-Berechnung	307
Parameter	205
<i>aktueller</i>	206
<i>formaler</i>	205
Parameterliste	200, 203
Parameterloser Konstruktor	434
Parameterübergabe-Mechanismus	206
Parametrisierter Konstruktor	438
Parametrisierter Typ	731
parseBoolean(), Boolean	375
parseByte(), Byte	375
Parsed Character Data	1112
parseDouble(), Double	375
parseFloat(), Float	375
parseInt(), Integer	375, 379, 558, 696
parseLong(), Long	375, 380
parseShort(), Short	375
Partiell abstrakte Klasse	504
PATH	82
Path, Schnittstelle	1063
Paths, Klasse	1063
Payne, Jonathan	51
PCDATA	1112
PDA	74
Persistenz	862
PhantomReference, Klasse	669
Phasen, Maven	936
Pipeline, Stream-API	1010
Plattformunabhängigkeit	53
Plugin	88
Plugin Eclipse	101
Plugin, Maven	936
Plus, überladenes	164
Plus/Minus, unäres	155
Point, Klasse	230, 234
Pointer	58
Polar-Methode	1172
policytool	1217
polling	954
Polymorphie	496
POM-Datei	933
POSITIVE_INFINITY	1165
Post-Dekrement	148
PostgreSQL	1101
Post-Inkrement	148
Potenz	1162
Prä-Dekrement	148
Präfix	345
Präfix-Berechnung, parallele	307
Präfix-Mechanismus, JAXB	1128
Prä-Inkrement	148
print()	115, 215
printf()	116
printf(), PrintWriter/PrintStream	392
println()	115
printStackTrace(), Throwable	570
Priorität, Thread	963
Prioritätswarteschlange	963
private, Schlüsselwort	410
Privatsphäre	409
Profiler	903

Profiling .....	901	Reference, abstrakte Klasse .....	669
Programm .....	112	Referenz	
Programmieren gegen Schnittstellen .....	515, 551	<i>schwache</i> .....	669
Programmierparadigma .....	817	<i>starke</i> .....	669
Programmiersprache, imperative .....	111	Referenzielle Transparenz .....	848
Programmierung, deklarative .....	817	Referenzierung .....	165
Project Object Model .....	933	Referenztyp .....	58, 119, 123, 477
Properties, Bean .....	862	<i>Vergleich mit ==</i> .....	265
Properties, Klasse .....	911	Referenzvariable .....	58, 236
Property .....	413, 861	regionMatches(), String .....	345
PropertyChangeEvent, Klasse .....	864	Reified Generics .....	746
PropertyChangeListener, Schnittstelle .....	864	Reihung .....	269
PropertyChangeSupport, Klasse .....	864	Reine abstrakte Klasse .....	504
Property-Design-Pattern .....	862	Rekursionsform .....	220
Property-Sheet .....	861	Rekursive Methode .....	219
PropertyVetoException .....	867	Rekursiver Type-Bound .....	755
protected, Schlüsselwort .....	469	Relation .....	1099
Protocol .....	65	Relationales Datenbanksystem .....	1102
Provider .....	824	reliable configuration .....	889
Pseudo-Primzahltest .....	1176	remainder operator .....	144
Pseudozufallszahl .....	1170	Rendezvous .....	960
public, Schlüsselwort .....	409	repaint() .....	1058
Punkt-Operator .....	238	replace(), String .....	354
Pure abstrakte Klasse .....	504	replaceAll(), String .....	355
Purer Ausdruck .....	848	replaceFirst(), String .....	355

## Q

qNaN .....	1151
Quadratwurzel .....	1161
Quasiparallelität .....	939
Quellcode, erreichbarer .....	208
quiet NaN .....	1151
quote(), Pattern .....	356

## R

Random, Klasse .....	1170
random(), Math .....	282, 1165
RandomAccess, Schnittstelle .....	988
RandomAccessFile, Klasse .....	1072
Range-Checking .....	60
Rangordnung .....	155
Raw-Type .....	750
readability, Modul .....	889
Reader, Klasse .....	1087
readUTF(), RandomAccessFile .....	1075
Real-time Java .....	76
Rechenungsgenauigkeit .....	190
Rechtsassoziativität .....	157
Reduzierende Operation .....	1011

Reference, abstrakte Klasse .....	669
Referenz	
<i>schwache</i> .....	669
<i>starke</i> .....	669
Referenzielle Transparenz .....	848
Referenzierung .....	165
Referenztyp .....	58, 119, 123, 477
<i>Vergleich mit ==</i> .....	265
Referenzvariable .....	58, 236
regionMatches(), String .....	345
Reified Generics .....	746
Reihung .....	269
Reine abstrakte Klasse .....	504
Rekursionsform .....	220
Rekursive Methode .....	219
Rekursiver Type-Bound .....	755
Relation .....	1099
Relationales Datenbanksystem .....	1102
reliable configuration .....	889
remainder operator .....	144
Rendezvous .....	960
repaint() .....	1058
replace(), String .....	354
replaceAll(), String .....	355
replaceFirst(), String .....	355
requireNonNull(), Objects .....	588
Rest einer Division .....	1159
Restwert-Operator .....	142, 144, 1158
Resultat .....	119
resume(), Thread .....	962
rethrow, Ausnahmen .....	593
return, Schlüsselwort .....	207, 292
Reverse-Engineering-Tool .....	233
Rich Internet Application (RIA) .....	71
rint(), Math .....	1157
round(), Math .....	1157
RoundingMode, Aufzählung .....	1187
Roundtrip-Engineering .....	233
RTSJ .....	76
Rückgabetyp .....	200
<i>kovarianter</i> .....	491
Rumbaugh, Jim .....	231
Rumpf .....	200
run(), Runnable .....	945
Runden .....	1156
<i>kaufmännisches</i> .....	1157
Rundungsfehler .....	145
Rundungsmodi, BigDecimal .....	1185
runFinalizersOnExit(), System .....	668
Runnable, Schnittstelle .....	944

RuntimeException .....	580
Runtime-Interpreter .....	52
<b>S</b>	
Scala .....	54
Scanner, Klasse .....	386, 559
Schablonenmuster .....	1081
Schaltjahr .....	922
ScheduledExecutorService, Schnittstelle .....	976
ScheduledThreadPoolExecutor, Klasse .....	965, 976
Scheduler, Betriebssystem .....	940
Schema .....	1115
Schleife .....	182
<i>äußere</i> .....	189
<i>innere</i> .....	189
Schleifenbedingung .....	184, 190
Schleifen-Inkrement .....	187
Schleifentest .....	187
Schleifenzähler .....	187
Schlüsselwort, reserviertes .....	107
Schnittstelle .....	65, 510
<i>erweiterte</i> .....	529
<i>funktionale</i> .....	789
Schnittstellenmethode, statische .....	525
Schnittstellentyp .....	237
Schriftarten .....	320
Schwache Referenz .....	669
Scope .....	216
Sealing, Jar .....	640
SecureRandom, Klasse .....	1174
Security-Manager .....	57
Sedezimalsystem .....	1137
Seed .....	1171
Seiteneffekt .....	402
Selbstbeobachtung .....	861
Selenium .....	1216
Semantik .....	103
Separator .....	104
Sequenzdiagramm .....	232
Sequenzpunkte in C(++) .....	148
Server JRE .....	80
Service Provider Implementation .....	1245
Set, Schnittstelle .....	998
setPriority(), Thread .....	963
Setter .....	413, 863
Set-Top-Box .....	50
shadowed variable .....	404
shallow copy .....	656
Sheridan, Mike .....	50
Shift .....	156
Shift-Operator .....	1143
short, Datentyp .....	125, 135, 1136
Short, Klasse .....	688
Short-Circuit-Operator .....	154
Sichtbarkeit .....	216, 409
Sichtbarkeitsmodifizierer .....	409
Signaling NaN .....	1152
Signatur .....	200
Silverlight .....	71
Simple API for XML Parsing .....	1121
SIMPLIFIED_CHINESE, Locale .....	919
SIMULA .....	65
Simula-67 .....	227
sin(), Math .....	1163
single inheritance .....	467
Single Responsibility Principle (SRP) .....	549
Singleton .....	431, 853
sinh(), Math .....	1164
Sinus .....	1164
sizeof .....	165
sleep(), Thread .....	952
Slivka, Ben .....	66
Smalltalk .....	54, 227
Smiley .....	323
sNaN .....	1152
SoftReference, Klasse .....	669
Softwarearchitektur .....	851
SOLID .....	549
sort(), Arrays .....	296
Späte dynamische Bindung .....	495
Speicherbereinigung, automatische .....	422
Speicherfreigabe, automatische .....	59
spin-loop .....	954
spinning .....	954
SPI-Paket .....	1245
split(), String .....	385
SplittableRandom, Klasse .....	1175
Sprungmarke, switch .....	176
Sprungziel, switch .....	176
SRP (Single Responsibility Principle) .....	549
Stack .....	222
Stack-Case-Label .....	180
StackOverflowError .....	222, 582
Stack-Speicher .....	236
Stack-Trace .....	556
Standard Directory Layout .....	98, 934
Standard Extension API .....	900
Standard Widget Toolkit .....	1038
Standard-Konstruktor .....	436
Star Seven .....	50
Stark typisiert .....	123

Starke Referenz .....	669	Swing .....	1036
start(), Thread .....	946	Swing JavaBuilder .....	1041
startsWith(), String .....	345	switch-Anweisung .....	175
Start-Tag .....	1109	SWT .....	1038
Statement .....	111	Symbolische Konstante .....	426
static, Schlüsselwort .....	121, 421	Symmetrie, equals() .....	651
Statisch typisiert .....	123	sync() .....	1092
Statische Eigenschaft .....	420	Synchronisation .....	668
Statische innere Klasse .....	624	SynerJ .....	89
Statische Schnittstellenmethode .....	525	Syntax .....	103
Statischer (Initialisierungs-)Block .....	448	Synthetische Methode .....	626, 781
Statischer Import .....	255	System.err .....	120, 570
Statusrückgabewert, ignoriert .....	558	System.in .....	1081
Stellenwertsystem .....	1137	System.out .....	120
Steuerelement, grafisches .....	1036	Systemeigenschaft .....	911
Stilles NaN .....	1151	System-Klassenlader .....	874, 907
stop(), Thread .....	955		
Streng typisiert .....	123		
strictfp, Schlüsselwort .....	1189		
StrictMath, Klasse .....	1189		
String .....	114		
<i>anhängen an einen</i> .....	350		
<i>Länge</i> .....	335		
StringBuffer, Klasse .....	333, 360		
StringBuilder, Klasse .....	333, 360		
StringIndexOutOfBoundsException .....	336, 347		
String-Konkatenation .....	156		
String-Literal .....	334		
String-Teil			
<i>extrahieren</i> .....	336, 346		
<i>vergleichen</i> .....	345		
Stromklasse .....	1079		
Stroustrup, Bjarne .....	229		
Stub-Objekt .....	1214		
STUPID .....	551		
Subinterface .....	521		
Subklasse .....	465		
Substitutionsprinzip .....	479		
substring(), String .....	346		
Subtraktion .....	142		
Suffix .....	345		
Sun Microsystems .....	51		
SunWorld .....	51		
super .....	475		
super, Schlüsselwort .....	487		
super() .....	469, 472, 475		
Superklasse .....	465		
Supplier .....	824		
suppressed exception .....	569		
Surrogate-Paar .....	325		
suspend(), Thread .....	962		
Swing .....	1036		
Swing JavaBuilder .....	1041		
switch-Anweisung .....	175		
SWT .....	1038		
Symbolische Konstante .....	426		
Symmetrie, equals() .....	651		
sync() .....	1092		
Synchronisation .....	668		
SynerJ .....	89		
Syntax .....	103		
Synthetische Methode .....	626, 781		
System.err .....	120, 570		
System.in .....	1081		
System.out .....	120		
Systemeigenschaft .....	911		
System-Klassenlader .....	874, 907		

**T**

Tag .....	1107
tail call optimization .....	223
tail call recursion .....	222
TAIWAN .....	919
tan(), Math .....	1163
tanh(), Math .....	1164
TCFTC .....	566
TCK (Technology Compatibility Kit) .....	77
TDD → Test-Driven Development (TDD)	
Technology Compatibility Kit (TCK) .....	77
Teil-String .....	337
template pattern .....	1081
Terminale Operation .....	1011
Test-Driven Development (TDD) .....	1195
Test-First .....	1195
Test-Runner .....	1194
this, Vererbung .....	637
this() .....	474
this(), Beschränkungen .....	443
this(), Konstruktorauftruf .....	442
this\$0, innere Klasse .....	629
this-Referenz .....	405, 474
this-Referenz, innere Klasse .....	627
Thread .....	54, 940
<i>nativer</i> .....	940
Thread, Klasse .....	946
ThreadLocalRandom, Klasse .....	1015
Thread-Pool .....	965
ThreadPoolExecutor, Klasse .....	965
three-way comparison .....	676
throw, Schlüsselwort .....	582
Throwable, Klasse .....	570

throws, Schlüsselwort .....	563
Tiefe Kopie, clone() .....	656
TimeUnit, Aufzählung .....	923
toArray(), Collection .....	994
toBinaryString(), Integer/Long .....	378
toCharArray(), String .....	349
toHexString(), Integer/Long .....	378
Token .....	103
toLowerCase(), Character .....	330
toLowerCase(), String .....	352
toOctalString(), Integer/Long .....	378
Top-Level-Klasse, geschachtelte .....	624
toString(), Arrays .....	295
toString(), Object .....	483, 645
toString(), Point .....	240, 241
toUpperCase(), Character .....	330
toUpperCase(), String .....	352
Trait .....	538
Transparenz, referenzielle .....	848
TreeSet, Klasse .....	1003
Trennzeichen .....	104, 384
trim(), String .....	353
true .....	125
TRUE, Boolean .....	697
try mit Ressourcen .....	387, 602
try, Schlüsselwort .....	554
try-Block .....	557
Tupel .....	1099
Türme von Hanoi .....	224
Typ .....	
<i>arithmetischer</i> .....	125
<i>generischer</i> .....	730
<i>geschachtelter</i> .....	623
<i>innerer</i> .....	623
<i>integraler</i> .....	125
<i>numerischer</i> .....	119
<i>parametrisierter</i> .....	731
<i>primitiver</i> .....	123
Typ-Annotation .....	721
type erasure .....	744
Type, Schnittstelle .....	906
TYPE, Wrapper-Klasse .....	645, 904
Type-Bound, rekursiver .....	755
Typecast .....	158
Typ-Inferenz .....	735, 740
Typisiert .....	
<i>stark</i> .....	123
<i>statisch</i> .....	123
<i>streng</i> .....	123
Typlöschung .....	744, 745
Typ-Token .....	774
Typumwandlung .....	158
<i>automatische</i> .....	158, 475
<i>explizite</i> .....	158, 477
Typvariable .....	730
Typvergleich .....	156
<b>U</b> .....	
U+, Unicode .....	319
Überdeckt, Attribut .....	483
Überdeckte Methode .....	497
Überladene Methode .....	116, 214
Überladener Operator .....	62
Überlauf .....	1166
Überschreiben .....	
<i>kovariantes</i> .....	779
<i>Methode</i> .....	482
Übersetzer .....	83
Übersetzungsfehler .....	85
UI (User Interface) .....	1035
UK, Locale .....	919
Umbrella-JSR .....	70
Umgebungsvariablen, Betriebssystem .....	916
Umkehrfunktion .....	1164
UML .....	230
Umlaut .....	323
Umwandlung, numerische .....	143
Unärer Operator .....	140
Unäres Minus .....	146
Unäres Plus/Minus .....	155
Unbenanntes Modul .....	889
Unbenanntes Paket .....	253
Unboxing .....	699
UncaughtExceptionHandler, Schnittstelle .....	958, 959
unchecked exception .....	581
UncheckedIOException .....	601
Und .....	153
<i>bitweises</i> .....	156, 1134
<i>logisches</i> .....	156
Unendlich .....	1148
Ungeprüfte Ausnahme .....	590
Unicode 5.1 .....	319
Unicode-Escape .....	323
Unicode-Konsortium .....	319
Unicode-Zeichen .....	104
Unidirektionale Beziehung .....	457
Unified Method .....	231
Unit-Test .....	1194
Unix Epoche .....	922
Unixzeit .....	922

Unmarshaler, Schnittstelle .....	1129
Unnamed package .....	253
UnsupportedOperationException .....	581, 612, 706, 992
Unterklasse .....	465
Unterstrich in Zahlen .....	137
Unzahl .....	143
Upper-bounded Wildcard-Typ .....	763
URL, Klasse .....	559
US, Locale .....	919
Use-Cases-Diagramm .....	232
User Interface .....	1035
UTF-16 .....	1111
UTF-16-Kodierung .....	321, 325
UTF-8 .....	1111
UTF-8-Format, modifiziertes .....	1075
UTF-8-Kodierung .....	321, 1075
Utility-Klasse .....	437
<b>V</b>	
Valid, XML .....	1111
valueOf(), Enum .....	711
valueOf(), String .....	373
valueOf(), Wrapper-Klassen .....	685
Vararg .....	283
Variable	
<i>globale</i> .....	216
<i>indexierte</i> .....	274
<i>verdeckte</i> .....	404, 405
Variablendeklaration .....	127
Variableninitialisierung .....	498
Variadische Funktion .....	283
Vector, Klasse .....	980
-verbose .....	1219, 1222
Verbundoperator .....	149
Verdeckte Methode .....	497
Verdeckte Variable .....	404, 405
Vererbte Konstante .....	522
Vererbung .....	242, 464
Vergleichsoperator .....	150, 151
Vergleichs-String .....	345
Verkettete Liste .....	989
Verklemmung .....	941
Verschiebeoperator .....	1143
veto property .....	862
Vetorecht .....	862
Virtuelle Erweiterungsmethode .....	529
Virtuelle Maschine .....	52
Visual Age for Java .....	87
VisualVM, Software .....	903
void, Schlüsselwort .....	203
void-kompatibel .....	798
Vorfahre, Vererbung .....	465
Vorzeichen, negatives .....	140
Vorzeichenerweiterung .....	156
Vorzeichenumkehr .....	146
<b>W</b>	
W3C (World Wide Web Consortium) .....	900
Wahrheitswert .....	123
Warten, aktives .....	954
WeakReference, Klasse .....	669
Web-Applet .....	51
WebRunner .....	51
Weißraum .....	328, 329
Wertebereich .....	412
Werte-Objekt .....	686
Wertoperation .....	141
Wertübergabe .....	205, 206
while-Schleife .....	182
Whitespace .....	104
widening conversion .....	159
Widget .....	1036
Wiederverwendung per Copy & Paste .....	1140
Wildcard .....	762
Wildcard-Capture .....	771
Win32-API .....	66
WindowBuilder, Google .....	1042
Windows-1252 .....	319
Winkelfunktion .....	1163
Wissenschaftliche Notation .....	1152
Wohlgemacht .....	1109
Workbench, Eclipse .....	98
Workspace .....	91
World Wide Web .....	50
World Wide Web Consortium (W3C) .....	900
Wrapper-Klasse .....	375, 683
write once, run anywhere .....	80
Writer, Klasse .....	1084
writeUTF(), RandomAccessFile .....	1075
<b>X</b>	
XAML .....	1040
Xerces, Apache .....	1122
XML .....	1108
-Xms .....	1222
-Xmx .....	1222
-Xnoclassgc .....	1222

XOR .....	153, 1055
<i>bitweises</i> .....	156
<i>logisches</i> .....	156
-Xrs .....	1222
XSD .....	1115
-Xss .....	1222
-Xss:n .....	222
-XX:ThreadStackSize=n .....	222
 <b>Y</b>	
YAGNI .....	549
yield(), Thread .....	954
Yoda-Stil .....	152
YourKit, Software .....	903
 <b>Z</b>	
Zahl	
<i>eulersche</i> .....	1154
<i>hexadezimale</i> .....	1137
<i>komplexe</i> .....	1192
<i>magische</i> .....	427
Zahlenwert, Unicode-Zeichen .....	104
Zehnersystem .....	1137
Zeichen .....	123, 138
<i>anhängen von</i> .....	367
<i>ersetzen</i> .....	354
Zeichenkette .....	114
<i>konstante</i> .....	334
<i>veränderbare</i> .....	360
Zeiger .....	58
Zeilenkommentar .....	112
Zeilenumbruch .....	915
Zeilenumbruchzeichen .....	915
Zeitmessung .....	901
Zeitzone .....	922
Zero-Assembler Project .....	57
Zieltyp, Lambda-Ausdruck .....	790
Zufallszahl .....	1165, 1178
Zufallszahlengenerator .....	1171
Zugriffsmethode .....	413
Zustand, Thread .....	951
Zustandsänderung .....	870
Zuverlässige Konfiguration .....	889
Zuweisung .....	142, 156
<i>mit Operation</i> .....	156
<i>mit String-Konkatenation</i> .....	156
Zweidimensionales Feld .....	286
Zweierkomplement .....	1135



# Die Serviceseiten

Im Folgenden finden Sie Hinweise, wie Sie Kontakt zu uns aufnehmen können.

## Lob und Tadel

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Wenn Sie zufrieden waren, empfehlen Sie das Buch bitte weiter. Wenn Sie meinen, es gebe doch etwas zu verbessern, schreiben Sie direkt an die Lektorin dieses Buches: [anne.scheibe@rheinwerk-verlag.de](mailto:anne.scheibe@rheinwerk-verlag.de). Wir freuen uns über jeden Verbesserungsvorschlag, aber über ein Lob freuen wir uns natürlich auch!

Auch auf unserer Webkatalogseite zu diesem Buch haben Sie die Möglichkeit, Ihr Feedback an uns zu senden oder Ihre Leseerfahrung per Facebook, Twitter oder E-Mail mit anderen zu teilen. Folgen Sie einfach diesem Link: <http://www.rheinwerk-verlag.de/4804>.

## Zusatzmaterialien

Zusatzmaterialien (Beispielcode, Übungsmaterial, Listen usw.) finden Sie in Ihrer Online-Bibliothek sowie auf der Webkatalogseite zu diesem Buch: <http://www.rheinwerk-verlag.de/4804>. Wenn uns sinnentstellende Tippfehler oder inhaltliche Mängel bekannt werden, stellen wir Ihnen dort auch eine Liste mit Korrekturen zur Verfügung.

## Technische Probleme

Im Falle von technischen Schwierigkeiten mit dem E-Book oder Ihrem E-Book-Konto beim Rheinwerk Verlag steht Ihnen gerne unser Leserservice zur Verfügung: [ebooks@rheinwerk-verlag.de](mailto:ebooks@rheinwerk-verlag.de).

## Über uns und unser Programm

Informationen zu unserem Verlag und weitere Kontaktmöglichkeiten bieten wir Ihnen auf unserer Verlagswebsite <http://www.rheinwerk-verlag.de>. Dort können Sie sich auch umfassend und aus erster Hand über unser aktuelles Verlagsprogramm informieren und alle unsere Bücher und E-Books schnell und komfortabel bestellen. Alle Buchbestellungen sind für Sie versandkostenfrei.

# Rechtliche Hinweise

In diesem Abschnitt finden Sie die ausführlichen und rechtlich verbindlichen Nutzungsbedingungen für dieses E-Book.

## Copyright-Vermerk

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Nutzungs- und Verwertungsrechte liegen beim Autor und beim Rheinwerk Verlag. Insbesondere das Recht der Vervielfältigung und Verbreitung, sei es in gedruckter oder in elektronischer Form.

© Rheinwerk Verlag GmbH, Bonn 2019

## Ihre Rechte als Nutzer

Sie sind berechtigt, dieses E-Book ausschließlich für persönliche Zwecke zu nutzen. Insbesondere sind Sie berechtigt, das E-Book für Ihren eigenen Gebrauch auszudrucken oder eine Kopie herzustellen, sofern Sie diese Kopie auf einem von Ihnen alleine und persönlich genutzten Endgerät speichern. Zu anderen oder weitergehenden Nutzungen und Verwertungen sind Sie nicht berechtigt.

So ist es insbesondere unzulässig, eine elektronische oder gedruckte Kopie an Dritte weiterzugeben. Unzulässig und nicht erlaubt ist des Weiteren, das E-Book im Internet, in Intranets oder auf andere Weise zu verbreiten oder Dritten zur Verfügung zu stellen. Eine öffentliche Wiedergabe oder sonstige Weiterveröffentlichung und jegliche den persönlichen Gebrauch übersteigende Vervielfältigung des E-Books ist ausdrücklich untersagt. Das vorstehend Gesagte gilt nicht nur für das E-Book insgesamt, sondern auch für seine Teile (z.B. Grafiken, Fotos, Tabellen, Textabschnitte).

Urheberrechtsvermerke, Markenzeichen und andere Rechtsvorbehalte dürfen aus dem E-Book nicht entfernt werden, auch nicht das digitale Wasserzeichen.

## Digitales Wasserzeichen

Dieses E-Book-Exemplar ist mit einem **digitalen Wasserzeichen** versehen, einem Vermerk, der kenntlich macht, welche Person dieses Exemplar nutzen darf. Wenn Sie, lieber Leser, diese Person nicht sind, liegt ein Verstoß gegen das Urheberrecht vor, und wir bitten Sie freundlich, das E-Book nicht weiter zu nutzen und uns diesen Verstoß zu melden. Eine kurze E-Mail an [service@rheinwerk-verlag.de](mailto:service@rheinwerk-verlag.de) reicht schon. Vielen Dank!

## Markenschutz

Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

## Haftungsausschluss

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.

# Über den Autor

**Christian Ullenboom**, Dipl.-Informatiker, ist Oracle-zertifizierter Java-Programmierer und seit 1997 Trainer und Berater für Java-Technologien und objektorientierte Analyse und Design. Seit Jahren teilt er sein Wissen mit unzähligen Besuchern seiner Website, wo er Fragen beantwortet, Inhalte bereitstellt und diskutiert. Seine Sympathie gilt Java Performance Tuning und den sinnlichen Freuden des Lebens.