5. Auflage

Michael Inden

Der Weg zum Java-Profi

Konzepte und Techniken für die professionelle Java-Entwicklung

dpunkt.verlag

Dipl.-Inform. Michael Inden ist Oracle-zertifizierter Java-Entwickler. Nach seinem Studium in Oldenburg hat er bei diversen internationalen Firmen in verschiedenen Rollen etwa als Softwareentwickler, -architekt, Consultant, Teamleiter, CTO sowie Leiter Academy gearbeitet. Zurzeit ist er freiberuflich als Autor und Trainer in Zürich tätig.

Michael Inden hat über zwanzig Jahre Berufserfahrung beim Entwurf komplexer Softwaresysteme gesammelt, an diversen Fortbildungen und mehreren Java-One-Konferenzen teilgenommen. Sein besonderes Interesse gilt dem Design qualitativ hochwertiger Applikationen sowie dem Coaching. Sein Wissen gibt er gerne als Trainer in internen und externen Schulungen und auf Konferenzen weiter, etwa bei der JAX/W-JAX, JAX London, Oracle Code One, ch.open sowie bei der Java User Group Switzerland.

Papier plus+ PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.plus

Michael Inden

Der Weg zum Java-Profi

Konzepte und Techniken für die professionelle Java-Entwicklung

5., überarbeitete und aktualisierte Auflage

dpunkt.verlag

Michael Inden
michael_inden@hotmail.com

Lektorat: Dr. Michael Barabas
Projektkoordinierung/Lektoratsassistenz: Anja Weimer
Fachgutachten: Torsten Horn, Aachen
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Michael Inden
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck: Schleunungdruck GmbH, Marktheidenfeld
Bindung: Hubert & Co. GmbH & Co. KG. BuchPartner, Göttingen

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:
Print 978-3-86490-707-4
PDF 978-3-96088-842-0
ePub 978-3-96088-843-7
mobi 978-3-96088-844-4

5., überarbeitete und aktualisierte Auflage 2021
Copyright © 2021 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Der Umwelt zuliebe verzichten wir auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es unswissen: hallo@dpunkt.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Inhaltsübersicht

1 Einleitung 1

I Java-Grundlagen, Analyse und Design 11

2 Professionelle Arbeitsumgebung 13

3 Objektorientiertes Design 85

4 Lambdas, Methodenreferenzen und Defaultmethoden 199

5 Java-Grundlagen 221

II Bausteine stabiler Java-Applikationen 325

6 Das Collections-Framework 327

7 Das Stream-API 453

8 Datumsverarbeitung seit JDK 8 497

9 Applikationsbausteine 529

10 Multithreading-Grundlagen 581

11 Modern Concurrency 633

12 Fortgeschrittene Java-Themen 681

13 Basiswissen Internationalisierung 743

III Wichtige Neuerungen in Java 12 bis 15 771

14 Neues und Änderungen in den Java-Versionen 12 bis 15 773

IV Modularisierung 823

15 Modularisierung mit Project Jigsaw 825

V Fallstricke und Lösungen im Praxisalltag 877

16 Bad Smells 879

17 Refactorings 953

18 Entwurfsmuster 1023

VI Qualitätssicherungsmaßnahmen 1099

19 Programmierstil und Coding Conventions 1101

20 Unit Tests 1143

21 Codereviews 1265

22 Optimierungen 1273

23 Schlussgedanken 1349

VII Anhang 1351

A Grundlagen zur Java Virtual Machine 1353

Literaturverzeichnis 1357

Index 1361

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Über dieses Buch 1
1.1.1 Motivation 1
1.1.2 Was leistet dieses Buch und was nicht? 2
1.1.3 Wie und was soll mithilfe des Buchs gelernt werden? 2
1.1.4 Wer sollte dieses Buch lesen? 4
1.2 Aufbau des Buchs 4
1.3 Konventionen und ausführbare Programme 6

I Java-Grundlagen, Analyse und Design **11**

2 Professionelle Arbeitsumgebung **13**
2.1 Vorteile von IDEs am Beispiel von Eclipse 14
2.2 Projektorganisation 16
2.2.1 Projektstruktur in Eclipse 16
2.2.2 Projektstruktur für Maven und Gradle 18
2.3 Einsatz von Versionsverwaltungen 20
2.3.1 Arbeiten mit zentralen Versionsverwaltungen 23
2.3.2 Dezentrale Versionsverwaltungen 28
2.3.3 VCS und DVCS im Vergleich 34
2.4 Einsatz eines Unit-Test-Frameworks 37
2.4.1 Das JUnit-Framework 37
2.4.2 Parametrierte Tests mit JUnit 5 43
2.4.3 Vorteile von Unit Tests 45
2.5 Debugging 46
2.5.1 Fehlersuche mit einem Debugger 47
2.5.2 Remote Debugging 50
2.6 Deployment von Java-Applikationen 55
2.6.1 Das JAR-Tool im Kurzüberblick 56
2.6.2 JAR inspizieren und ändern, Inhalt extrahieren 57
2.6.3 Metainformationen und das Manifest 58
2.6.4 Inspizieren einer JAR-Datei 60

2.7 Einsatz eines IDE-unabhängigen Build-Prozesses 63
2.7.1 Maven im Überblick 65
2.7.2 Builds mit Gradle 70
2.7.3 Vorteile von Maven und Gradle 82
2.8 Weiterführende Literatur 83

3 Objektorientiertes Design 85
3.1 OO-Grundlagen 86
3.1.1 Grundbegriffe 86
3.1.2 Beispielentwurf: Ein Zähler 98
3.1.3 Vom imperativen zum objektorientierten Entwurf 106
3.1.4 Diskussion der OO-Grundgedanken 111
3.1.5 Wissenswertes zum Objektzustand 114
3.2 Grundlegende OO-Techniken 123
3.2.1 Schnittstellen (Interfaces) 123
3.2.2 Basisklassen und abstrakte Basisklassen 128
3.2.3 Interfaces und abstrakte Basisklassen 130
3.3 Wissenswertes zu Vererbung 132
3.3.1 Probleme durch Vererbung 132
3.3.2 Delegation statt Vererbung 138
3.4 Fortgeschrittenere OO-Techniken 142
3.4.1 Read-only-Interface 142
3.4.2 Immutable-Klasse 148
3.4.3 Marker-Interface 153
3.4.4 Konstantensammlungen und Aufzählungen 154
3.4.5 Value Object (Data Transfer Object) 160
3.5 Prinzipien guten OO-Designs 162
3.5.1 Geheimnisprinzip nach Parnas 163
3.5.2 Law of Demeter 164
3.5.3 SOLID-Prinzipien 167
3.6 Formen der Varianz 178
3.6.1 Grundlagen der Varianz 178
3.6.2 Kovariante Rückgabewerte 182
3.7 Generische Typen (Generics) 184
3.7.1 Einführung 184
3.7.2 Generics und Auswirkungen der Type Erasure 189
3.8 Weiterführende Literatur 197

4 Lambdas, Methodenreferenzen und Defaultmethoden 199
4.1 Einstieg in Lambdas 199
4.1.1 Syntax von Lambdas 199
4.1.2 Functional Interfaces und SAM-Typen 200
4.1.3 Exceptions in Lambdas 204

4.2 Syntaxerweiterungen in Interfaces ... 208
4.2.1 Defaultmethoden ... 208
4.2.2 Statische Methoden in Interfaces ... 211
4.3 Methodenreferenzen ... 213
4.4 Externe vs. interne Iteration ... 214
4.5 Wichtige Functional Interfaces für Collections ... 215
4.5.1 Das Interface `Predicate<T>` ... 215
4.5.2 Das Interface `UnaryOperator<T>` ... 218
4.6 Praxiswissen: Definition von Lambdas ... 219

5 Java-Grundlagen ... 221
5.1 Die Klasse `Object` ... 221
5.1.1 Die Methode `toString()` ... 223
5.1.2 Die Methode `equals()` ... 227
5.2 Primitive Typen und Wrapper-Klassen ... 239
5.2.1 Grundlagen ... 240
5.2.2 Konvertierung von Werten ... 247
5.2.3 Wissenswertes zu Auto-Boxing und Auto-Unboxing ... 250
5.2.4 Ausgabe und Verarbeitung von Zahlen ... 253
5.3 Stringverarbeitung ... 256
5.3.1 Die Klasse `String` ... 257
5.3.2 Die Klassen `StringBuffer` und `StringBuilder` ... 261
5.3.3 Ausgaben mit `format()` und `printf()` ... 264
5.3.4 Die Methode `split()` und reguläre Ausdrücke ... 265
5.3.5 Optimierung bei Strings in JDK 9 ... 270
5.3.6 Neue Methoden in der Klasse `String` in JDK 11 ... 270
5.4 Varianten innerer Klassen ... 272
5.5 Ein- und Ausgabe (I/O) ... 276
5.5.1 Dateibehandlung und die Klasse `File` ... 276
5.5.2 Ein- und Ausgabestreams im Überblick ... 280
5.5.3 Zeichencodierungen bei der Ein- und Ausgabe ... 283
5.5.4 Speichern und Laden von Daten und Objekten ... 285
5.5.5 Dateiverarbeitung mit dem NIO ... 293
5.5.6 Neue Hilfsmethoden in der Klasse `Files` in JDK 11 ... 296
5.6 Fehlerbehandlung ... 298
5.6.1 Einstieg in die Fehlerbehandlung ... 298
5.6.2 Checked Exceptions und Unchecked Exceptions ... 304
5.6.3 Besonderheiten beim Exception Handling ... 306
5.6.4 Exception Handling und Ressourcenfreigabe ... 310
5.6.5 Assertions ... 314

5.7 Weitere Neuerungen in JDK 9, 10 und 11 318
5.7.1 Erweiterung der `@Deprecated`-Annotation in JDK 9 318
5.7.2 Syntaxerweiterung `var` in JDK 10 und 11 319
5.7.3 Versionsverarbeitung mit JDK 9 und 10 323
5.8 Weiterführende Literatur 324

II Bausteine stabiler Java-Applikationen 325

6 Das Collections-Framework 327
6.1 Datenstrukturen und Containerklassen 327
6.1.1 Wahl einer geeigneten Datenstruktur 328
6.1.2 Arrays 330
6.1.3 Das Interface `Collection` 332
6.1.4 Das Interface `Iterator` 334
6.1.5 Listen und das Interface `List` 338
6.1.6 Mengen und das Interface `Set` 346
6.1.7 Grundlagen von hashbasierten Containern 350
6.1.8 Grundlagen automatisch sortierender Container 359
6.1.9 Die Methoden `equals()`, `hashCode()` und `compareTo()` im Zusammenspiel 367
6.1.10 Schlüssel-Wert-Abbildungen und das Interface `Map` 369
6.1.11 Erweiterungen am Beispiel der Klasse `HashMap` 378
6.1.12 Erweiterungen im Interface `Map` in JDK 8 381
6.1.13 Collection-Factory-Methoden in JDK 9 386
6.1.14 Unveränderliche Kopien von Collections mit Java 10 390
6.1.15 Entscheidungshilfe zur Wahl von Datenstrukturen 392
6.2 Suchen und Sortieren 393
6.2.1 Suchen 393
6.2.2 Sortieren von Arrays und Listen 396
6.2.3 Sortieren mit Komparatoren 399
6.2.4 Erweiterungen im Interface `Comparator` mit JDK 8 401
6.3 Utility-Klassen und Hilfsmethoden 406
6.3.1 Nützliche Hilfsmethoden 406
6.3.2 Dekorierer `synchronized` und `unmodifiable` 411
6.3.3 Vordefinierte Algorithmen in der Klasse `Collections` 415
6.4 Containerklassen: Generics und Varianz 417
6.5 Die Klasse `Optional<T>` 431
6.5.1 Grundlagen zur Klasse `Optional` 432
6.5.2 Weiterführendes Beispiel und Diskussion 436
6.5.3 Verkettete Methodenaufrufe 438
6.5.4 Erweiterungen in der Klasse `Optional<T>` in JDK 9 439
6.5.5 Erweiterung in `Optional<T>` in JDK 10 und 11 442

6.6 Fallstricke im Collections-Framework ... 443
6.6.1 Wissenswertes zu Arrays ... 443
6.6.2 Wissenswertes zu `Stack`, `Queue` und `Deque` ... 447
6.7 Weiterführende Literatur ... 450

7 Das Stream-API ... 453
7.1 Grundlagen zu Streams ... 453
7.1.1 Streams erzeugen – Create Operations ... 454
7.1.2 Intermediate und Terminal Operations im Überblick ... 456
7.1.3 Zustandslose Intermediate Operations ... 458
7.1.4 Zustandsbehaftete Intermediate Operations ... 465
7.1.5 Terminal Operations ... 467
7.1.6 Wissenswertes zur Parallelverarbeitung ... 476
7.1.7 Neuerungen im Stream-API in JDK 9 ... 481
7.1.8 Neuerungen im Stream-API in JDK 10 ... 484
7.2 Filter-Map-Reduce ... 485
7.2.1 Herkömmliche Realisierung ... 486
7.2.2 Filter-Map-Reduce mit JDK 8 ... 488
7.3 Praxisbeispiele ... 490
7.3.1 Aufbereiten von Gruppierungen und Histogrammen ... 491
7.3.2 Maps nach Wert sortieren ... 492

8 Datumsverarbeitung seit JDK 8 ... 497
8.1 Überblick über die neu eingeführten Typen ... 497
8.1.1 Neue Aufzählungen, Klassen und Interfaces ... 498
8.1.2 Die Aufzählungen `DayOfWeek` und `Month` ... 500
8.1.3 Die Klassen `MonthDay`, `YearMonth` und `Year` ... 500
8.1.4 Die Klasse `Instant` ... 501
8.1.5 Die Klasse `Duration` ... 502
8.1.6 Die Aufzählung `ChronoUnit` ... 506
8.1.7 Die Klassen `LocalDate`, `LocalTime` und `LocalDateTime` 507
8.1.8 Die Klasse `Period` ... 510
8.1.9 Die Klasse `ZonedDateTime` ... 511
8.1.10 Zeitzonen und die Klassen `ZoneId` und `ZoneOffset` ... 512
8.1.11 Die Klasse `Clock` ... 514
8.1.12 Formatierung und Parsing ... 516
8.2 Datumsarithmetik ... 517
8.2.1 Einstieg in die Datumsarithmetik ... 517
8.2.2 Real-World-Example: Gehaltszahltag ... 522
8.3 Interoperabilität mit Legacy-Code ... 526

9 Applikationsbausteine ... 529
9.1 Einsatz von Bibliotheken ... 530
9.2 Google Guava im Kurzüberblick ... 533
9.2.1 String-Aktionen ... 534
9.2.2 Stringkonkatenation und -extraktion ... 535
9.2.3 Erweiterungen für Collections ... 540
9.2.4 Weitere Utility-Funktionalitäten ... 544
9.3 Wertebereichs- und Parameterprüfungen ... 548
9.4 Logging-Frameworks ... 551
9.4.1 Apache log4j2 ... 552
9.4.2 Tipps und Tricks zum Einsatz von Logging mit log4j2 ... 556
9.5 Konfigurationsparameter und -dateien ... 561
9.5.1 Einlesen von Kommandozeilenparametern ... 561
9.5.2 Verarbeitung von Properties ... 569
9.5.3 Weitere Möglichkeiten zur Konfigurationsverwaltung ... 575

10 Multithreading-Grundlagen ... 581
10.1 Threads und Runnables ... 583
10.1.1 Definition der auszuführenden Aufgabe ... 583
10.1.2 Start, Ausführung und Ende von Threads ... 584
10.1.3 Lebenszyklus von Threads und Thread-Zustände ... 588
10.1.4 Unterbrechungswünsche durch Aufruf von `interrupt()` ... 591
10.2 Zusammenarbeit von Threads ... 594
10.2.1 Konkurrierende Datenzugriffe ... 594
10.2.2 Locks, Monitore und kritische Bereiche ... 595
10.2.3 Deadlocks und Starvation ... 602
10.2.4 Kritische Bereiche und das Interface `Lock` ... 604
10.3 Kommunikation von Threads ... 606
10.3.1 Kommunikation mit Synchronisation ... 606
10.3.2 Kommunikation über die Methoden `wait()`, `notify()` und `notifyAll()` ... 610
10.3.3 Abstimmung von Threads ... 615
10.3.4 Unerwartete `IllegalMonitorStateException`s ... 617
10.4 Das Java-Memory-Modell ... 619
10.4.1 Sichtbarkeit ... 619
10.4.2 Atomarität ... 620
10.4.3 Reorderings ... 622
10.5 Besonderheiten bei Threads ... 626
10.5.1 Verschiedene Arten von Threads ... 626
10.5.2 Exceptions in Threads ... 627
10.5.3 Sicheres Beenden von Threads ... 628
10.6 Weiterführende Literatur ... 632

11 Modern Concurrency ... 633
11.1 Concurrent Collections ... 634
11.1.1 Thread-Sicherheit und Parallelität mit »normalen« Collections ... 634
11.1.2 Parallelität mit den Concurrent Collections ... 636
11.1.3 Blockierende Warteschlangen und das Interface `BlockingQueue<E>` ... 639
11.2 Das Executor-Framework ... 642
11.2.1 Einführung ... 642
11.2.2 Definition von Aufgaben ... 645
11.2.3 Parallele Abarbeitung im `ExecutorService` ... 649
11.3 Das Fork-Join-Framework ... 652
11.3.1 Einführendes Beispiel ... 653
11.3.2 Real-World-Example: Merge Sort ... 656
11.4 Die Klasse `CompletableFuture<T>` ... 659
11.4.1 Einführung ... 659
11.4.2 Beispiel: Parallele Verarbeitung von Dateiinhalten ... 662
11.4.3 Erweiterungen in JDK 9 ... 665
11.4.4 Beispiel: Von synchron zu mutlithreaded ... 667
11.5 Reactive Streams und die Klasse `Flow` ... 670
11.5.1 Schnelleinstieg Reactive Streams ... 670
11.5.2 Reactive Streams im JDK ... 671
11.5.3 Beispiel zur Klasse `Flow` ... 675
11.5.4 Fazit ... 679
11.6 Weiterführende Literatur ... 680

12 Fortgeschrittene Java-Themen ... 681
12.1 Crashkurs Reflection ... 681
12.1.1 Grundlagen ... 683
12.1.2 Zugriff auf Methoden und Attribute ... 686
12.1.3 Spezialfälle ... 691
12.1.4 Type Erasure und Typinformationen bei Generics ... 697
12.1.5 Fazit ... 699
12.2 Annotations ... 699
12.2.1 Einführung in Annotations ... 700
12.2.2 Standard-Annotations des JDKs ... 701
12.2.3 Definition eigener Annotations ... 703
12.2.4 Annotations zur Laufzeit auslesen ... 706
12.3 Serialisierung ... 708
12.3.1 Grundlagen der Serialisierung ... 708
12.3.2 Die Serialisierung anpassen ... 713
12.3.3 Versionsverwaltung der Serialisierung ... 716
12.3.4 Optimierung der Serialisierung ... 720

12.4 Garbage Collection ... 725
12.4.1 Grundlagen zur Garbage Collection ... 725
12.4.2 Der Garbage Collector »G1« ... 727
12.5 Dynamic Proxies ... 728
12.5.1 Statischer Proxy ... 730
12.5.2 Dynamischer Proxy ... 732
12.6 HTTP/2-API ... 737
12.6.1 Einführung ... 737
12.6.2 Real-World-Example: Wechselkurs mit REST ... 740
12.6.3 Fazit ... 742
12.7 Weiterführende Literatur ... 742

13 Basiswissen Internationalisierung ... 743
13.1 Internationalisierung im Überblick ... 743
13.1.1 Grundlagen und Normen ... 744
13.1.2 Die Klasse `Locale` ... 745
13.1.3 Die Klasse `PropertyResourceBundle` ... 749
13.1.4 Formatierte Ein- und Ausgabe ... 752
13.1.5 Datumswerte und die Klasse `DateFormat` ... 753
13.1.6 Zahlen und die Klasse `NumberFormat` ... 754
13.1.7 Textmeldungen und die Klasse `MessageFormat` ... 757
13.1.8 Stringvergleiche mit der Klasse `Collator` ... 759
13.2 Programmbausteine zur Internationalisierung ... 764
13.2.1 Unterstützung mehrerer Datumsformate ... 765
13.2.2 Fazit und Ausblick ... 770

III Wichtige Neuerungen in Java 12 bis 15 771

14 Neues und Änderungen in den Java-Versionen 12 bis 15 ... 773
14.1 Syntaxneuerungen ... 774
14.1.1 Text Blocks ... 774
14.1.2 Switch Expressions ... 779
14.1.3 Records (Preview) ... 786
14.1.4 Pattern Matching bei `instanceof` (Preview) ... 793
14.1.5 Sealed Types (Preview) ... 795
14.1.6 Lokale Enums und Interfaces (Preview) ... 797
14.2 API-Neuerungen ... 798
14.2.1 Neue Methoden in der Klasse `String` ... 798
14.2.2 Neue Hilfsmethode in der Utility-Klasse `Files` ... 801
14.2.3 Der `teeing()`-Kollektor ... 802

14.3 JVM-Neuerungen ... 805
14.3.1 Verbesserung bei `NullPointerExceptions` ... 805
14.3.2 Entfernung der JavaScript-Engine ... 808
14.4 Microbenchmark Suite ... 808
14.4.1 Eigene Microbenchmarks und Varianten davon ... 808
14.4.2 Microbenchmarks mit JMH ... 811
14.4.3 Fazit zu JMH ... 817
14.5 Java 15 – notwendige Anpassungen für Build-Tools und IDEs ... 818
14.5.1 Java 15 mit Gradle ... 818
14.5.2 Java 15 mit Maven ... 819
14.5.3 Java 15 mit Eclipse ... 820
14.5.4 Java 15 mit IntelliJ ... 821
14.5.5 Java 15 mit JShell oder der Kommandozeile ... 821
14.6 Fazit ... 822

IV Modularisierung 823

15 Modularisierung mit Project Jigsaw ... 825
15.1 Grundlagen ... 826
15.1.1 Begrifflichkeiten ... 826
15.1.2 Ziele von Project Jigsaw ... 827
15.2 Modularisierung im Überblick ... 828
15.2.1 Grundlagen zu Project Jigsaw ... 828
15.2.2 Beispiel mit zwei Modulen ... 834
15.2.3 Packaging ... 843
15.2.4 Abhängigkeiten und Modulgraphen ... 845
15.2.5 Module des JDKs einbinden ... 847
15.2.6 Arten von Modulen ... 853
15.3 Sichtbarkeiten und Zugriffsschutz ... 854
15.3.1 Sichtbarkeiten ... 854
15.3.2 Zugriffsschutz und Reflection ... 856
15.4 Empfehlenswertes Verzeichnislayout für Module ... 858
15.5 Kompatibilität und Migration ... 860
15.5.1 Kompatibilitätsmodus ... 860
15.5.2 Migrationsszenarien ... 863
15.5.3 Fallstrick bei der Bottom-up-Migration ... 867
15.5.4 Beispiel: Migration mit Automatic Modules ... 868
15.5.5 Beispiel: Automatic und Unnamed Module ... 870
15.5.6 Beispiel: Abwandlung mit zwei Automatic Modules ... 873
15.5.7 Fazit ... 875
15.6 Zusammenfassung ... 876

V Fallstricke und Lösungen im Praxisalltag 877

16 Bad Smells 879
16.1 Programmdesign 881
16.1.1 Bad Smell: Verwenden von Magic Numbers 881
16.1.2 Bad Smell: Konstanten in Interfaces definieren 882
16.1.3 Bad Smell: Zusammengehörende Konstanten nicht als Typ definiert 884
16.1.4 Bad Smell: Casts auf unbekannte Subtypen 886
16.1.5 Bad Smell: Programmcode im Logging-Code 888
16.1.6 Bad Smell: Dominanter Logging-Code 889
16.1.7 Bad Smell: Unvollständige Änderungen nach Copy-Paste . . 890
16.1.8 Bad Smell: Unvollständige Betrachtung aller Alternativen . . 892
16.1.9 Bad Smell: Prä-/Postinkrement in komplexeren Statements 893
16.1.10 Bad Smell: Mehrere aufeinanderfolgende Parameter gleichen Typs 896
16.1.11 Bad Smell: Grundloser Einsatz von Reflection 897
16.1.12 Bad Smell: `System.exit()` mitten im Programm 898
16.1.13 Bad Smell: Variablendeklaration nicht im kleinstmöglichen Sichtbarkeitsbereich 900
16.2 Klassendesign 902
16.2.1 Bad Smell: Unnötigerweise veränderliche Attribute 902
16.2.2 Bad Smell: Herausgabe von `this` im Konstruktor 904
16.2.3 Bad Smell: Aufruf abstrakter Methoden im Konstruktor 905
16.2.4 Bad Smell: Mix abstrakter und konkreter Basisklassen 910
16.2.5 Bad Smell: Referenzierung von Subklassen in Basisklassen 912
16.2.6 Bad Smell: Öffentlicher Defaultkonstruktor lediglich zum Zugriff auf Hilfsmethoden 914
16.3 Fehlerbehandlung und Exception Handling 915
16.3.1 Bad Smell: Unbehandelte Exception 915
16.3.2 Bad Smell: Unpassender Exception-Typ 917
16.3.3 Bad Smell: Fangen der allgemeinsten Exception 918
16.3.4 Bad Smell: Exceptions zur Steuerung des Kontrollflusses . . 920
16.3.5 Bad Smell: Unbedachte Rückgabe von `null` 922
16.3.6 Bad Smell: Rückgabe von `null` statt Exception im Fehlerfall 924
16.3.7 Bad Smell: Sonderbehandlung von Randfällen 926
16.3.8 Bad Smell: Keine Gültigkeitsprüfung von Eingabeparametern 927
16.3.9 Bad Smell: Fehlerhafte Fehlerbehandlung 929
16.3.10 Bad Smell: I/O ohne `finally` oder ARM 931
16.3.11 Bad Smell: Resource Leaks durch Exceptions im Konstruktor 932
16.4 Häufige Fallstricke 937
16.5 Weiterführende Literatur 952

17 Refactorings ... 953
17.1 Refactorings am Beispiel ... 954
17.2 Das Standardvorgehen ... 962
17.3 Kombination von Basis-Refactorings ... 965
17.3.1 Refactoring-Beispiel: Ausgangslage und Ziel ... 965
17.3.2 Auflösen der Abhängigkeiten ... 967
17.3.3 Vereinfachungen ... 974
17.3.4 Verlagern von Funktionalität ... 978
17.4 Der Refactoring-Katalog ... 979
17.4.1 Reduziere die Sichtbarkeit von Attributen ... 979
17.4.2 Minimiere veränderliche Attribute ... 982
17.4.3 Reduziere die Sichtbarkeit von Methoden ... 986
17.4.4 Ersetze Mutator- durch Business-Methode ... 988
17.4.5 Minimiere Zustandsänderungen ... 989
17.4.6 Führe ein Interface ein ... 989
17.4.7 Spalte ein Interface auf ... 990
17.4.8 Führe ein Read-only-Interface ein ... 991
17.4.9 Führe ein Read-Write-Interface ein ... 992
17.4.10 Lagere Funktionalität in Hilfsmethoden aus ... 993
17.4.11 Trenne Informationsbeschaffung und -verarbeitung ... 995
17.4.12 Wandle Konstantensammlung in `enum` um ... 1002
17.4.13 Entferne Exceptions zur Steuerung des Kontrollflusses ... 1004
17.4.14 Wandle in Utility-Klasse mit statischen Hilfsmethoden um ... 1007
17.4.15 Löse `if-else` / `instanceof` durch Polymorphie auf ... 1010
17.5 Defensives Programmieren ... 1013
17.5.1 Führe eine Zustandsprüfung ein ... 1013
17.5.2 Überprüfe Eingabeparameter ... 1014
17.6 Fallstricke bei Refactorings ... 1019
17.7 Weiterführende Literatur ... 1021

18 Entwurfsmuster ... 1023
18.1 Erzeugungsmuster ... 1026
18.1.1 Erzeugungsmethode ... 1026
18.1.2 Fabrikmethode (Factory Method) ... 1029
18.1.3 Erbauer (Builder) ... 1032
18.1.4 Singleton ... 1035
18.1.5 Prototyp (Prototype) ... 1040
18.2 Strukturmuster ... 1044
18.2.1 Fassade (Façade) ... 1044
18.2.2 Adapter ... 1047
18.2.3 Dekorierer (Decorator) ... 1049
18.2.4 Kompositum (Composite) ... 1052

18.3 Verhaltensmuster 1056
18.3.1 Iterator 1056
18.3.2 Null-Objekt (Null Object) 1058
18.3.3 Schablonenmethode (Template Method) 1061
18.3.4 Strategie (Strategy) 1065
18.3.5 Befehl (Command) 1077
18.3.6 Proxy 1084
18.3.7 Beobachter (Observer) 1086
18.3.8 MVC-Architektur 1095
18.4 Weiterführende Literatur 1097

VI Qualitätssicherungsmaßnahmen 1099

19 Programmierstil und Coding Conventions 1101
19.1 Grundregeln eines guten Programmierstils 1101
19.1.1 Keep It Human-Readable 1102
19.1.2 Keep It Simple And Short (KISS) 1102
19.1.3 Keep It Natural 1102
19.1.4 Keep It Clean 1103
19.2 Die Psychologie beim Sourcecode-Layout 1103
19.2.1 Gesetz der Ähnlichkeit 1103
19.2.2 Gesetz der Nähe 1105
19.3 Coding Conventions 1106
19.3.1 Grundlegende Namens- und Formatierungsregeln 1107
19.3.2 Namensgebung 1110
19.3.3 Dokumentation 1113
19.3.4 Programmdesign 1115
19.3.5 Klassendesign 1120
19.3.6 Parameterlisten 1124
19.3.7 Logik und Kontrollfluss 1125
19.4 Sourcecode-Prüfung mit Tools 1128
19.4.1 Metriken 1129
19.4.2 Sourcecode-Prüfung im Build-Prozess 1133

20 Unit Tests 1143
20.1 Testen im Überblick 1143
20.1.1 Was versteht man unter Testen? 1143
20.1.2 Testarten im Überblick 1144
20.1.3 Zuständigkeiten beim Testen 1147
20.1.4 Testen und Qualität 1150
20.2 Wissenswertes zu Testfällen 1154
20.2.1 Testfälle mit JUnit definieren 1154

20.2.2 Problem der Kombinatorik beim Bestimmen von Testfällen . 1163
20.3 Besondere Assertions und Annotations 1171
20.4 Parametrierte Tests mit JUnit 5 1182
20.4.1 Einstieg 1182
20.4.2 Verbesserung des Tests der Rabattberechnung 1189
20.4.3 Praxisbeispiel: Berechnung in Testfall vereinfachen 1191
20.4.4 Praxis-Trickkiste 1194
20.5 Fortgeschrittene Unit-Test-Techniken 1201
20.5.1 Stellvertreterobjekte / Test-Doubles 1204
20.5.2 Vorarbeiten für das Testen mit Stubs und Mocks 1208
20.5.3 Die Technik EXTRACT AND OVERRIDE 1210
20.5.4 Einstieg in das Testen mit Mocks und Mockito 1217
20.5.5 Abhängigkeiten mit Mockito auflösen 1226
20.5.6 Unit Tests von privaten Methoden 1228
20.6 Test Smells 1230
20.6.1 Test Smell: Unangebrachtes `assertTrue()` und `assertFalse()` 1230
20.6.2 Test Smell: Zu viele Asserts im Testfall 1231
20.6.3 Test Smell: Asserts ohne Hinweis 1233
20.6.4 Test Smell: Einsatz von `toString()` in `assertEquals()` 1234
20.6.5 Test Smell: Unit Tests zur Prüfung von Laufzeiten 1235
20.7 Nützliche Tools für Unit Tests 1237
20.7.1 Hamcrest 1237
20.7.2 AssertJ 1241
20.7.3 MoreUnit 1246
20.7.4 Infinitest 1247
20.7.5 JaCoCo 1248
20.7.6 EclEmma 1252
20.8 Umstieg von JUnit 4 auf JUnit 5 1254
20.8.1 Erweiterung im Gradle-Build für JUnit-5-Tests 1254
20.8.2 Veränderungen in den Annotations 1254
20.8.3 Alternativen für JUnit Rules 1256
20.8.4 Veränderungen bei parametrierten Tests 1260
20.8.5 Alternative zur Hamcrest-Integration in JUnit 4 1261
20.9 Fazit 1262
20.10 Weiterführende Literatur 1263

21 Codereviews 1265
21.1 Definition 1265
21.2 Probleme und Tipps zur Durchführung 1267
21.3 Vorteile von Codereviews 1269
21.4 Codereview-Checkliste 1272

22 Optimierungen ... 1273
22.1 Grundlagen ... 1274
22.1.1 Optimierungsebenen und Einflussfaktoren ... 1275
22.1.2 Optimierungstechniken ... 1276
22.1.3 CPU-bound-Optimierungsebenen am Beispiel ... 1278
22.1.4 Messungen – Erkennen kritischer Bereiche ... 1282
22.1.5 Abschätzungen mit der O-Notation ... 1289
22.2 Einsatz geeigneter Datenstrukturen ... 1292
22.2.1 Einfluss von Arrays und Listen ... 1293
22.2.2 Optimierungen für `Set` und `Map` ... 1297
22.2.3 Design eines Zugriffsinterface ... 1299
22.3 Lazy Initialization ... 1302
22.3.1 Konsequenzen des Einsatzes der Lazy Initialization ... 1305
22.3.2 Lazy Initialization mithilfe des PROXY-Musters ... 1307
22.4 Optimierungen am Beispiel ... 1310
22.5 I/O-bound-Optimierungen ... 1318
22.5.1 Technik – Wahl passender Strategien ... 1318
22.5.2 Technik – Caching und Pooling ... 1321
22.5.3 Technik – Vermeidung unnötiger Aktionen ... 1321
22.6 Memory-bound-Optimierungen ... 1325
22.6.1 Technik – Wahl passender Strategien ... 1325
22.6.2 Technik – Caching und Pooling ... 1326
22.6.3 Optimierungen der Stringverarbeitung ... 1333
22.6.4 Technik – Vermeidung unnötiger Aktionen ... 1336
22.7 CPU-bound-Optimierungen ... 1338
22.7.1 Technik – Wahl passender Strategien ... 1339
22.7.2 Technik – Caching und Pooling ... 1342
22.7.3 Technik – Vermeidung unnötiger Aktionen ... 1344
22.8 Weiterführende Literatur ... 1347

23 Schlussgedanken ... 1349

VII Anhang 1351

A Grundlagen zur Java Virtual Machine ... 1353
A.1 Wissenswertes rund um die Java Virtual Machine ... 1353
A.1.1 Ausführung eines Java-Programms ... 1353
A.1.2 Speicherverwaltung und Classloading ... 1354

Literaturverzeichnis ... 1357

Index ... 1361

Vorwort

Vorwort zur 5. Auflage

Sie halten die mittlerweile 5. Auflage dieses Buchs in den Händen. Das wurde nur durch den großen Zuspruch und das auch nach Jahren anhaltende Interesse für dieses Buch möglich. Somit geht zunächst ein herzlicher Dank an alle Leser der vorherigen Auflagen.

Diese 5. Auflage wurde vollständig auf Java 11 als derzeitige LTS-Version (Long Term Support) aktualisiert sowie in diversen Teilen überarbeitet und erweitert. Dieses Buch soll Ihnen einen fundierten Einstieg in die professionelle Java-Programmierung ermöglichen und damit Ihren Weg zum Java-Profi erleichtern. Wie schon aus den Vorgängern gewohnt, gebe ich immer wieder Tipps aus dem Praxisalltag, weise auf Fallstricke hin und zeige Lösungswege auf. Damit Sie aber am Puls der Zeit sind und über alles Wesentliche bis hin zum aktuellen Java 15 Bescheid wissen, behandle ich die vielfältigen Neuerungen ebenso wie die Modularisierung in jeweils eigenen Kapiteln. Für eine noch umfassendere Behandlung der Thematik verweise ich Sie auf mein Buch »Java – die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41].

Änderungen in dieser 5. Auflage

Im Rahmen der Überarbeitung für diese 5. Auflage habe ich das Buch nochmals vollständig gelesen und kritisch beleuchtet. Dadurch konnten kleinere Unstimmigkeiten, missverständliche Formulierungen und ein paar verbliebene Tippfehler erkannt und korrigiert werden. Zudem habe ich die Anregungen und Wünsche von Lesern sowie von Kollegen und Freunden mit eigenen Ideen kombiniert. Daraus sind diverse Ergänzungen und Überarbeitungen in den bereits vorhandenen Kapiteln entstanden. Auch wurden verschiedene Dinge restrukturiert und thematisch neu gegliedert.

Nachfolgend liste ich wesentliche Änderungen dieser 5. Auflage im Vergleich zum Vorgänger auf:

- Kapitel 2 »**Professionelle Arbeitsumgebung**« – Der Text wurde leicht überarbeitet und aktualisiert, das gilt etwa für die Einführung ins Unit-Testen. Diese setzt nun auf JUnit 5. Darüber hinaus behandle ich das Build-Tool Ant in dieser 5. Auflage nicht mehr, da es in der Praxis kaum noch eine Rolle spielt. Zudem wurde die Beschreibung von Gradle auf die im September 2020 aktuelle Version 6.6.1 angepasst.

- Kapitel 3 »**OO-Design**« – In diesem Kapitel wurden ein paar Details und Beispiele leicht überarbeitet, um die Verständlichkeit weiter zu verbessern, etwa im Bereich der kovarianten Rückgabewerte. Zudem weise ich, wo sinnvoll, auf mögliche Vereinfachungen durch aktuelle Java-14- bzw. Java-15-Sprachmittel hin.
- Kapitel 4 »**Lambdas, Methodenreferenzen und Defaultmethode**n« – Dieses Kapitel enthält nun Neuerungen für Interfaces in Java 9. Außerdem wurde eine Beschreibung zu einer kosmetischen Erweiterung in `Predicate<T>` in JDK 11 ergänzt. Schließlich thematisiere ich nun ein kleines, aber wichtiges Detail bei der Definition von Lambdas und dem Zugriff auf Variablen.
- Kapitel 5 »**Java-Grundlagen**« – Die Behandlung der Java-Grundlagen wurde gestrafft, etwa im Bereich des alten APIs zur Dateiverwaltung. Zudem habe ich ganze Abschnitte wie denjenigen zum alten Datums-API entfernt und diverse Beispiele auf das moderne Date and Time API umgestellt. Außerdem wurden wesentliche Neuerungen aus Java 9 bis 11 an passenden Stellen, etwa im Bereich für Strings und auch für die Utility-Klasse `Files` oder die Syntaxneuerung `var`, integriert.
- Kapitel 6 »**Das Collections-Framework**« – Neben Detailkorrekturen inklusive inhaltlicher Straffung wurden einige Neuerungen aus Java 8 und 9 an passender Stelle hinzugefügt, insbesondere Collection-Factory-Methoden sowie Hilfsmethoden in der Klasse `Arrays`. Die Beschreibung zur Klasse `Optional<T>` wurde überarbeitet und um die Neuerungen aus Java 10 und 11 erweitert.
- Kapitel 7 »**Stream-API**« – In diesem Kapitel wurden einige inhaltliche Ergänzungen, etwa für die Kombination von `skip()` und `limit()`, sowie kleinere sprachliche Korrekturen vorgenommen. Darüber hinaus ist vor allem ein Abschnitt zu den Neuerungen im Stream-API in Java 9 hinzugefügt worden. Die Verarbeitung von ZIP-Dateien wurde als Praxisbeispiel entfernt.
- Kapitel 8 »**Datumsverarbeitung seit JDK 8**« – Dieses Kapitel hat ebenfalls kleine sprachliche Korrekturen sowie minimale Anpassungen zur Stringenz der Beispiele erfahren. Zudem werden nun Erweiterungen aus Java 9 beschrieben. Um die Möglichkeiten der Datumsarithmetik noch besser nachvollziehen zu können, habe ich zwei Beispiele ergänzt. Besonders erwähnenswert ist dasjenige zur Berechnung des Gehaltszahltags als Real-World-Example. Dieses verfolgt ein schrittweises Vorgehen mit dem Hinzufügen von jeweils kleinen Funktionsblöcken und zeigt auch gleich geeignetes Testing mit JUnit 5.
- Kapitel 9 »**Applikationsbausteine**« – In diesem Kapitel wurden diverse Kleinigkeiten angepasst und Kürzungen vorgenommen. Das betrifft vor allem die Themen Preferences sowie Wertebereichsprüfungen. Bei Letzterem wurde auf die Darstellung eigener Lösungen zugunsten von Google Guava verzichtet. Zudem habe ich das abschließende Beispiel zur Verarbeitung von CSV so umgestaltet, dass es nun das Java-14-Feature `record`s nutzt.

- Kapitel 10 »**Multithreading**« – Dieses Kapitel hat diverse Detailänderungen erfahren und der Teil zu Producer-Consumer wurde gekürzt. Weil mittlerweile modernere und zu bevorzugende Konstrukte existieren, habe ich die Beschreibung zur zeitgesteuerten Ausführung mit `Timer` und `TimerTask` entfernt. Insbesondere wurden fortgeschrittenere Themen und die Beschreibung moderner APIs in ein eigenständiges Kapitel ausgelagert.
- Kapitel 11 »**Modern Concurrency**« – Dieses Kapitel wurde komplett neu gestaltet und diverse Beispiele ergänzt und prägnanter formuliert. Ziel ist es, modernere Concurrency-Konzepte vorzustellen. Die hier beschriebenen Concurrency Utilities und das Executor- sowie das Fork-Join-Framework erlauben es, statt auf Low-Level-Ebene zu parallelisierende Abläufe auf konzeptionell höherer Ebene umzusetzen. Aber es wird noch besser: Verarbeitungsabläufe lassen sich mithilfe von `CompletableFuture` sehr elegant beschreiben und parallelisieren. Abschließend thematisiere ich die Reactive Streams, die eine Verbindung zum Reactive Programming bilden.
- Kapitel 12 »**Fortgeschrittene Java-Themen**« – Der Text hat ein Facelift erhalten. Die Beispiele wurden durchgehend auf das Date and Time API aktualisiert. Insbesondere habe ich den Teil zur Garbage Collection deutlich gekürzt. Vollständig entfernt wurde die Beschreibung zur JavaScript-Verarbeitung, da diese mit Java 15 nicht mehr Bestandteil des JDKs ist. Eine Beschreibung zum HTTP/2-API wurde in diesem Kapitel ergänzt. Als Schmankerl schauen wir uns dort an, wie man damit REST-Calls absetzen und auf diese Weise Wechselkurse ermitteln kann.
- Kapitel 13 »**Basiswissen Internationalisierung**« – Der Text wurde in den Bereichen gekürzt, die das alte Datums-API behandelt haben. Diverse Teile verwenden jetzt das modernere Date and Time API aus Java 8. Da Java im Bereich Desktop und GUIs kaum mehr verwendet wird, habe ich die Beispiele, die JavaFX oder Swing nutzen, entfernt.
- Kapitel 17 »**Refactorings**« – Auch das Kapitel zu Refactorings hat einen Facelift erfahren. Vor allem wurden alle begleitenden Unit Tests auf JUnit 5 umgestellt. Zudem kam in einem Beispiel noch die Bibliothek Joda-Time zum Einsatz. Das wurde nun auf das Date and Time API des JDKs umgestellt.
- Kapitel 19 »**Programmierstil und Coding Conventions**« – Dieses Kapitel enthält Detailkorrekturen. Dabei wurden vor allem die Abschnitte zu den Tools überarbeitet und aktualisiert. Die Beschreibung zum Tool FindBugs sowie dessen Nachfolger SpotBugs wurde entfernt, da diese keine neuen Java-Versionen unterstützen, derzeit nicht einmal Java 10 und 11.

- Kapitel 20 »**Unit Tests**« – Auch hier wurde der gesamte Text überarbeitet und umstrukturiert. Vor allem wird nun konsequent das aktuelle JUnit 5 genutzt, wodurch sich einige Testfälle deutlich leichter als mit den Vorgängern beschreiben lassen. Insbesondere den parametrierten Tests wird viel Aufmerksamkeit gewidmet. Test Smells werden nun ausführlicher behandelt und zeigen konkrete Lösungsmöglichkeiten.
- Kapitel 22 »**Optimierungen**« – Die Performance-Messungen wurden mit JDK 14 wiederholt. Einige Messungen erfolgten ergänzend mit dem seit JDK 12 ins JDK integrierten JMH (Java Microbenchmark Harness). Beim Kopieren von Dateiinhalten wird nun auch die Variante mit `transferTo()` als Neuerung aus Java 9 betrachtet. Bei den CPU-bound-Optimierungen zeige ich mit Memoization eine sehr effiziente Technik, um rekursive Berechnungen dramatisch beschleunigen zu können.

Erweiterungen

- Kapitel 14 »**Ergänzungen in Java 12 bis 15**« – In den letzten 3 Jahren sind diverse Java-Versionen erschienen, die verschiedene interessante Neuerungen in der Syntax und den APIs mit sich bringen. Wichtige Themen werden in diesem Kapitel vorgestellt.

Entfallene Themen

Aus drucktechnischen Gründen mussten die Kapitel zu JavaFX sowie zur GUI-Programmierung mit Swing aus der Druckvariante des Buchs entfernt werden. Diese stehen aber – wie auch einige in früheren Auflagen entfernte Anhänge zu UML und dem Softwareentwicklungsprozess – als PDF auf der Seite des Verlags zum Download bereit.

Danksagung

Ein Fachbuch zu schreiben ist eine schöne, aber arbeitsreiche und langwierige Aufgabe. Alleine kann man dies kaum bewältigen, daher möchte ich mich an dieser Stelle bei allen bedanken, die direkt oder indirekt dazu beigetragen haben.

Bei der Erstellung des Manuskripts konnte ich auf ein starkes Team an Korrekturlesern zurückgreifen, insbesondere wieder einmal auf Michael Kulla, der akribisch gelesen und viele hilfreiche Anmerkungen eingebracht hat. Vielen Dank! Für das neu erstellte Kapitel Modern Concurrency haben Prof. Dr. Rainer Oechsle und René Preißel diverse nützliche Anmerkungen zur Verbesserung beigesteuert und kleinere Unstimmigkeiten aufgedeckt. In diesem Kontext erhielt ich ein paar hilfreiche Anmerkungen von Dr. Heinz Kabutz. Besten Dank an alle!

Wie immer geht natürlich auch ein Dankeschön an das Team des dpunkt.verlags (vor allem Dr. Michael Barabas, Anja Weimer und Stefanie Weidner) für die gute Zusammenarbeit. Außerdem möchte ich mich bei Torsten Horn für die fundierte fachliche Durchsicht sowie bei Ursula Zimpfer für ihre Adleraugen beim Copy-Editing bedanken.

Abschließend geht natürlich ein lieber Dank an meine Frau Lilija für ihr Verständnis und ihre Unterstützung. Bei der Erstellung dieser 5. Auflage war ich glücklicherweise weit weniger im Stress als bei den vorherigen Ausgaben, sodass sogar die eine oder andere gemeinsame Fahrradtour oder Wanderung möglich wurde. Auch die Vorbereitungen auf die Geburt unserer Tochter laufen auf Hochtouren und die Freude ist groß.

Anregungen und Kritik

Ich wünsche allen Lesern viel Freude und einige neue Erkenntnisse durch die Lektüre dieser 5. Auflage. Möge Ihnen der »Weg zum Java-Profi« mit meinem Buch ein wenig leichter fallen.

Trotz großer Sorgfalt lassen sich Fehler bei einem so umfangreichen Buch leider nicht vollständig vermeiden. Falls Ihnen ein solcher auffällt oder eine Formulierung missverständlich sein sollte, so zögern Sie bitte nicht, mir dies mitzuteilen. Haben Sie Anregungen, Verbesserungsvorschläge oder fehlt Ihnen noch eine Information? Sie erreichen mich per Mail unter: `michael_inden@hotmail.com`.

Zürich, im Oktober 2020
Michael Inden

Danksagung zur 4. Auflage

Bei der Erstellung des Manuskripts konnte ich auf ein starkes Team an Korrekturlesern zurückgreifen, insbesondere diesmal auch auf Michael Kulla, der akribisch gelesen und viele hilfreiche Anmerkungen eingebracht hat. Vielen Dank! Wieder einmal haben mich meine Freunde Merten Driemeyer, Dr. Clemens Gugenberger, Prof. Dr. Carsten Kern und Andreas Schöneck hervorragend unterstützt. Den einen oder anderen Hinweis und Tipp erhielt ich von Jeton Memeti, Marius Reusch und Prof. Dr. Andreas Spillner.

Weil der Java-9-Teil inhaltlich – wenn auch vom Umfang deutlich abgespeckt – weitestgehend meinem Buch »Java 9 – Die Neuerungen« [40] entstammt, danke ich allen dort Beteiligten ebenfalls.

Danksagung zur 3. Auflage

Bei der Erstellung des Manuskripts konnte ich auf ein starkes Team an Korrekturlesern zurückgreifen, insbesondere diesmal auch auf Benjamin Muschko und Hans Dockter als Experten zu Gradle sowie Hendrik Schreiber, selbst Autor eines Java-Fachbuchs zu Optimierungen. Vielen Dank an euch!

Einige Tipps erhielt ich von Tim Bötzmeyer und Reinhard Pupkes. Auch haben mich folgende Personen hervorragend unterstützt: Merten Driemeyer, Dr. Clemens Gugenberger, Dr. Carsten Kern, Florian Messerschmidt und Andreas Schöneck. Darüber hinaus kamen gute Anmerkungen von verschiedenen Zühlke-Kollegen: Michael Haspra, Jörg Keller, Rick Janda, Franziska Meyer, Sagi Nedunkanal, Joachim Prinzbach und Dr. Christoph Schmitz. Der Java-8-Teil entstammt weitestgehend meinem Buch »Java 8 – Die Neuerungen«. Allen dort Beteiligten danke ich ebenfalls.

Neben den Korrekturlesern möchte ich einen ganz herzlichen Dank an meinen damaligen Arbeitgeber Zühlke Engineering AG und insbesondere meinen Chef Wolfgang Giersche für die gewährte freie Zeit zum Finalisieren des Buchs aussprechen. Das war eine große Hilfe in der letzten heißen Phase vor der Abgabe des Manuskripts.

Danksagung zur 2. Auflage

Bei der Erstellung der 2. Auflage konnte ich wieder auf ein starkes Team an Korrekturlesern zurückgreifen. Einige Tipps erhielt ich von Dr. Alexander Kort und Reinhard Pupkes. Auch haben mich folgende Personen hervorragend unterstützt: Stefan Bartels, Tim Bötzmeyer, Rudolf Braun, Andreas Bubolz, Merten Driemeyer, Bernd Eckstein, Dr. Clemens Gugenberger, Peter Kehren, Dr. Carsten Kern, Dr. Iris Rottländer, Roland Schmitt-Hartmann und Andreas Schöneck.

Dabei möchte ich folgende vier Personen herausheben: Stefan Bartels für seine sprachliche Gründlichkeit, Andreas Bubolz für seine Genauigkeit und Dr. Clemens Gugenberger sowie Andreas Schöneck für die ganzen hilfreichen Anregungen.

Danksagung zur 1. Auflage

Zu meiner Zeit bei der Heidelberger Druckmaschinen AG in Kiel ist bei den Vorbereitungen zu Codereviews und der Ausarbeitung von Vorträgen zum ersten Mal der Gedanke an ein solches Buch entstanden. Danke an meine damaligen Kollegen, die an diesen Meetings teilgenommen haben. Als Veranstalter und Vortragender lernt man immer wieder neue Details. Dietrich Mucha und Reinhard Pupkes danke ich für ihre Korrekturen und Anmerkungen, die gemeinsamen Erfahrungen beim Ausarbeiten von Coding Conventions und Codereviews sowie die nette Zeit beim Pair Programming. Die Zusammenarbeit mit Tim Bötzmeyer hat mir viel Freude bereitet. Unsere langen, interessanten Diskussionen über Java und die Fallstricke beim OO-Design haben mir diverse neue Einblicke verschafft.

Auch einige Kollegen bei der IVU Traffic Technologies AG in Aachen haben mich mit Korrekturen und Anregungen unterstützt. Unter anderem danke ich Rudolf Braun, Christian Gehrmann, Peter Kehren, Felix Korb und Roland Schmitt-Hartmann für den einen oder anderen Hinweis und Tipp, um den Text weiter zu verbessern. Mein spezieller Dank gilt Merten Driemeyer, der sich sehr gründlich mit frühen Entwürfen des Manuskripts beschäftigt und mir an diversen Stellen fachliche und sprachliche Tipps gegeben hat. Gleiches gilt für Dr. Iris Rottländer, die sowohl formal als auch inhaltlich an vielen Stellen durch ihre Anmerkungen für eine Verbesserung des Textes gesorgt hat. Auch Dr. Carsten Kern und Andreas Schöneck haben gute Hinweise gegeben und einige verbliebene kleinere Fehler aufgedeckt. Last, but not least haben die Anmerkungen von Dr. Clemens Gugenberger und unsere nachfolgenden Diskussionen einigen Kapiteln den letzten Feinschliff gegeben. Gleiches gilt für Stefan Bartels, der mich immer wieder durch gute Anmerkungen unterstützt und damit zur Verständlichkeit des Textes beigetragen hat. Alle sechs haben mir entscheidend geholfen, inhaltlich für mehr Stringenz und Klarheit zu sorgen. Mein größter Dank geht an Andreas Bubolz, der mich immer wieder unterstützt und enorm viel Zeit und Mühe investiert hat. Als Korrekturleser und Sparringspartner in vielen Diskussionen hat er diverse Unstimmigkeiten im entstehenden Text aufgedeckt.

Einleitung

1 Einleitung

Bevor es mit den Programmierthemen losgeht, möchte ich Ihnen dieses Buch vorstellen. Ich beginne damit, warum dieses Buch entstanden ist und wie es Ihnen hoffentlich helfen kann, ein noch besserer Java-Entwickler zu werden. Danach folgt eine Gliederung des Inhalts, damit Sie sich gut im Buch zurechtfinden.

1.1 Über dieses Buch

1.1.1 Motivation

Mein Ziel war es, ein Buch zu schreiben, wie ich es mir selbst immer als Hilfe gewünscht habe. Die hier vorgestellten Hinweise und Techniken sollen Sie auf Ihrem Weg vom engagierten Hobbyprogrammierer oder Berufseinsteiger zum erfahrenen Softwareentwickler begleiten. Dieser Weg ist ohne Anleitung gewöhnlich steinig und mit einige Mühen, Irrwegen und Problemen verbunden. Einige dieser leidvollen Erfahrungen möchte ich Ihnen ersparen. Aber auch erfahreneren Softwareentwicklern soll dieses Buch die Möglichkeit geben, über die im täglichen Einsatz lieb gewonnenen Gewohnheiten nachzudenken und die eine oder andere davon zu ändern, um die Produktivität weiter zu steigern. Mein Wunsch ist, dass sich nach Lektüre des Buchs für Sie die Ingenieurdisziplin der Softwareentwicklung mit der Kunst des Programmierens verbindet und Dinge auf einmal einfach so auf Anhieb funktionieren. Das ist etwas ganz anderes, als vor jedem Gang in die Testabteilung Magenschmerzen zu bekommen.

Sowohl der Berufseinstieg als auch die tägliche Arbeit können manchmal frustrierend sein. Meiner Meinung nach soll Softwareentwicklung aber Freude bereiten, denn nur so können wir exzellente Resultate erzielen. In der Praxis besteht jedoch die Gefahr, in die Fettnäpfchen zu treten, die im Sourcecode hinterlassen wurden. Dies geschieht meistens dadurch, dass die existierende Lösung softwaretechnisch umständlich oder schlecht implementiert ist und/oder nicht bis zu Ende durchdacht wurde. Der Sourcecode ist dann häufig schwierig wart- und erweiterbar. Manchmal bereitet bereits das Auffinden der Stelle, an der man Modifikationen durchführen sollte, Probleme.

Dieses Buch soll aufzeigen, wie man die zuvor beschriebene »Altlasten«-Falle vermeidet oder aus ihr herauskommt und endlich Sourcecode schreiben kann und darf, der leicht zu lesen ist und in dem es Spaß macht, Erweiterungen zu realisieren. Grundlage dafür ist, dass wir uns einen Grundstock an Verhaltensweisen und an Wissen aneignen.

1.1.2 Was leistet dieses Buch und was nicht?

Wieso noch ein Buch über Java-Programmierung? Tatsächlich kann man sich diese Frage stellen, wo es doch unzählige Bücher zu diesem Thema gibt. Viele davon sind einführende Bücher, die häufig nur kurz die APIs anhand simpler Beispiele vorstellen. Die andere große Masse der Java-Literatur beschäftigt sich mit speziellen Themen, die für den »erfahrenen Einsteiger« bereits zu komplex geschrieben und in denen zu wenig erklärt ist. Genau hier setzt dieses Buch an und wagt den Spagat, den Leser nach der Lektüre einführender Bücher abzuholen und so weit zu begleiten, dass er mit einem guten Verständnis die Spezialliteratur lesen und gewinnbringend einsetzen kann.

Ziel dieses Buchs ist es, dem Leser fundierte Kenntnisse in Java und einigen praxisrelevanten Themenbereichen, unter anderem dem Collections-Framework und im Bereich Multithreading, zu vermitteln. Es werden vertiefende Blicke auf die zugrunde liegenden Details geworfen, um nach der Lektüre des Buchs professionelle Programme schreiben zu können. Wie bereits angedeutet, bietet dieses Buch keinen Einstieg in die Sprache selbst, sondern es wird einiges an Wissen vorausgesetzt. In den einleitenden Grundlagenkapiteln zu einer professionellen Arbeitsumgebung, über objektorientiertes Design und Java sowie über die funktionale Programmierung mit Lambdas wird allerdings die Basis für das Verständnis der Folgekapitel geschaffen.

In diesem Buch versuche ich, einen lockeren Schreibstil zu verwenden und nur an den Stellen formal zu werden, wo dies wirklich wichtig ist, etwa bei der Einhaltung von Methodenkontrakten. Da der Fokus dieses Buchs auf dem praktischen Nutzen und dem guten Verständnis von Konzepten liegt, werden neben APIs auch häufig vereinfachte Beispiele aus der realen Welt vorgestellt. Die meisten der abgebildeten Listings stehen als kompilierbare und lauffähige Programme auf der Webseite zum Buch zum Download bereit. Im Buch selbst werden aus Platzgründen und zugunsten einer besseren Übersichtlichkeit in der Regel nur die wichtigen Passagen abgedruckt.

1.1.3 Wie und was soll mithilfe des Buchs gelernt werden?

Dieses Buch zeigt und erklärt einige in der Praxis bewährte Ansätze, Vorgehens- und Verhaltensweisen, ohne dabei alle Themengebiete bis in kleinste Detail auszuleuchten. Wichtiges Hintergrundwissen wird jedoch bei Bedarf vermittelt. Es wird der pragmatische Weg gegangen und bevorzugt die in der täglichen Praxis relevanten Themen vorgestellt. Sollte ein Thema bei Ihnen besonderes Interesse wecken und Sie weitere Informationen wünschen, so finden sich in den meisten Kapiteln Hinweise auf weiterführende Literatur. Dies ist im Prinzip auch schon der erste Tipp: ***Lesen Sie viele Bücher und schaffen Sie sich damit eine breite Wissensbasis.*** Ich zitiere hier aus Jon Bentleys Buch »Perlen der Programmierkunst« [3]: ***»Im Stadium des Entwurfsprozesses ist es unschätzbar, die einschlägige Literatur zu kennen.«***

Diesem Hinweis kann ich mich nur anschließen und möchte Ihnen hier speziell einige – meiner Meinung nach – ganz besondere Bücher ans Herz legen und empfehle ausdrücklich, diese Bücher begleitend oder ergänzend zu diesem Buch zu lesen:

- »**SCJP – Sun Certified Programmer & Developer for Java 2**« [68] – Die Vorbereitung zur SCJP-Zertifizierung wird mit all seinen Fallstricken und kniffligen Details auf unterhaltsame Weise von Kathy Sierra und Bert Bates aufbereitet. Mittlerweile heißt die Zertifizierung OCPJP (Oracle Cerified Professional Java Programmer) und es sind diverse neue Bücher dazu erschienen. Das von Sierra und Bates war aber etwas Besonderes.
- »**The Java Programming Language**« [2] – Ein unglaublich gutes Buch von Ken Arnold, James Gosling und David Holmes über die Sprache Java, das detailreich, präzise und dabei angenehm verständlich zu lesen ist.
- »**Effective Java**« [5], [6] und [7] – Die erste Auflage dieses grandiosen Buchs von Joshua Bloch habe ich auf der Java One 2001 in San Francisco gekauft und es hat mein Denken und Programmieren in Java stark beeinflusst. Mittlerweile existiert eine dritte, auf JDK 9 aktualisierte Auflage (die leider kaum auf Modularisierung eingeht).
- »**Entwurfsmuster**« [23] – Das Standardwerk der sogenannten »Gang of Four« (Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides) habe ich 1998 kennengelernt und mit großem Interesse gelesen und gewinnbringend eingesetzt. Die vorgestellten Ideen sind beim Entwurf guter Software enorm hilfreich.
- »**Head First Design Patterns**« [21] – Dieses Buch einer anderen »Gang of Four« (Eric Freeman, Elizabeth Freeman, Kathy Sierra und Bert Bates) lässt Entwurfsmuster als ein unterhaltsames Thema erscheinen und erleichtert den Einstieg.
- »**Refactoring**« [20] – Einige Tricks und Kniffe zur Verbesserung von Sourcecode lernt man von Kollegen oder durch Erfahrung. Martin Fowler fasst dieses Wissen in dem genannten Buch zusammen und stellt ein systematisches Vorgehen zur Sourcecode-Transformation vor.
- »**Refactoring to Patterns**« [45] – Dieses Buch von Joshua Kerievsky verknüpft die Ideen von Refactorings mit denen zu Entwurfsmustern.
- »**Code Craft: The Practice of Writing Excellent Code**« [25] – Ein sehr lesenswertes Buch von Pete Goodliffe, das diverse gute Hinweise gibt, wie man exzellenten Sourcecode schreiben kann, der zudem (nahezu) fehlerfrei, gut testbar sowie einfach zu warten ist.

Lesen hilft uns bereits, aber nur durch Übung und Einsatz in der Praxis können wir unsere Fähigkeiten verbessern. Weil ein Buch jedoch nicht interaktiv ist, werde ich bevorzugt eine schrittweise Vorstellung der jeweiligen Themen vornehmen, wobei zum Teil auch bewusst zunächst ein Irrweg gezeigt wird. Anhand der vorgestellten Korrekturen erkennt man dann die Vorteile viel deutlicher, als wenn nur eine reine Präsentation der Lösung erfolgen würde. Mit dieser Darstellungsweise hoffe ich, dass Sie sich ein paar gute Gewohnheiten antrainieren. Das fängt mit scheinbar einfachen Dingen wie der Vergabe von sinnvollen Namen für Variablen, Methoden und Klassen an und endet in der Verwendung von problemangepassten Entwurfsmustern. Anfangs erfordert dies erfahrungsgemäß ein wenig Fleiß, Einarbeitung, Disziplin und eventuell sogar etwas

Überwindung. Daher werde ich bei der Vorstellung einer Technik jeweils sowohl auf die Vorteile als auch potenzielle Nachteile eingehen.

1.1.4 Wer sollte dieses Buch lesen?

Dieses Buch konzentriert sich auf Java als Programmiersprache – allerdings benötigen Sie bereits einige Erfahrung mit Java, um die Beispiele sowie die beschriebenen Tücken nachvollziehen zu können und möglichst viel von den Tipps und Tricks in diesem Buch zu profitieren. Wenn Sie dieses Buch in den Händen halten, gehe ich also davon aus, dass Sie sich schon (etwas) mit Java auseinandergesetzt haben.

Das Buch richtet sich im Speziellen an zwei Zielgruppen: Zum einen sind dies engagierte Hobbyprogrammierer, Informatikstudenten oder Berufseinsteiger, die von Anfang an lernen wollen, wie man professionell Software schreibt. Zum anderen sind dies erfahrenere Softwareentwickler, die ihr Wissen in einigen fortgeschritteneren Themen komplettieren wollen und vermehrt Priorität auf sauberes Design legen oder Coding Conventions, Codereviews und Unit-Testen bei der Arbeit etablieren wollen.

Abhängig vom Kenntnisstand zu Beginn der Lektüre starten Entwickler mit Erfahrung bei Teil II oder Teil III des Buchs und können die dort vorgestellten Techniken sofort gewinnbringend in der Praxis einsetzen. Lesern mit noch relativ wenig Erfahrung empfehle ich, den ersten Teil konzentriert und vollständig durchzuarbeiten, um sich eine gute Basis zu verschaffen. Dadurch wird das Verständnis der später vorgestellten Themen erleichtert, denn die nachfolgenden Teile des Buchs setzen die Kenntnis dieser Basis voraus und das Niveau nimmt ständig zu. Ziel ist es, nach Lektüre des Buchs den Einstieg in die professionelle Softwareentwicklung mit Java erreicht zu haben und viele dazu erforderliche Techniken sicher zu beherrschen. Das Buch ist daher mit diversen Praxistipps gespickt, mit denen Sie auf interessante Hintergrundinformationen oder auf mögliche Probleme hingewiesen werden und die wie folgt in den Text integriert sind:

Tipp: Praxistipp

In derart formatierten Kästen finden sich im späteren Verlauf des Buchs immer wieder einige wissenswerte Tipps und ergänzende Hinweise zum eigentlichen Text.

1.2 Aufbau des Buchs

Der Aufbau des Buchs gliedert sich in mehrere Teile. Folgende Aufzählung konkretisiert die dort vorgestellten Themen:

- **Teil I »Java-Grundlagen, Analyse und Design«** – Dieser Teil legt die Grundlagen für einen guten Softwareentwurf. Dazu wird sowohl auf objektorientiertes Design als auch auf eine produktive Arbeitsumgebung mit den richtigen Hilfsmitteln eingegangen. Teil I beginnt in Kapitel 2 mit der Vorstellung einer sinnvoll ausgestatte-

ten Arbeitsumgebung, die beim Entwickeln von professionellen Programmen hilft. Danach wird in Kapitel 3 das Thema objektorientiertes Design beschrieben. Damit sind die Grundlagen für einen professionellen, objektorientierten Softwareentwurf gelegt und die Vorbereitungen zum Implementieren getroffen. In Kapitel 4 erfolgt eine Vorstellung von Lambda-Ausdrücken, die die funktionale Programmierung mit Java ermöglichen. Zu Abschluss von Teil I stellt Kapitel 5 schließlich grundlegende Java-Sprachelemente vor, die zum Verständnis der Beispiele in den folgenden Kapiteln notwendig sind.

- **Teil II »Bausteine stabiler Java-Applikationen«** – Der zweite Teil beschäftigt sich mit Bausteinen stabiler Java-Applikationen. Dazu werden Kenntnisse zu fundamentalen Java-APIs vermittelt, aber auch fortgeschrittene Java-Techniken behandelt. Zunächst stellt Kapitel 6 das Thema Collections vor, um eine effiziente Wahl von Datenstrukturen zur Verwaltung von Daten zu ermöglichen. Im Anschluss gehe ich in Kapitel 7 auf das Stream-API ein. Danach schauen wir uns in Kapitel 8 die Verbesserungen bei der Datumsverarbeitung mit dem in Java 8 ergänzten Date and Time API an. In Kapitel 9 beschäftigen wir uns mit wiederverwendbaren Softwarebausteinen, die verhindern, das Rad ständig neu zu erfinden. Ein weiterer Baustein beim professionellen Programmieren ist Multithreading. Kapitel 10 bietet eine Einführung und thematisiert grundlegende Bausteine wie Threads, Synchronisierung usw. Damit lassen sich zwar Multithreading-Anwendungen schreiben, jedoch muss man als Programmierer sehr auf Details achten und kann sich weniger auf die Nebenläufigkeit konzentrieren. Mittlerweile gibt es diverse Abhilfen in Form verschiedener Bestandteile des JDKs, die es erlauben, Multithreading mehr auf konzeptioneller Ebene zu betrachten. Kapitel 11 beschreibt dies. Weitere fortgeschrittenere Themen, etwa Reflection, Annotations und Garbage Collection, werden in Kapitel 12 behandelt. Das Thema Internationalisierung und damit die Besonderheiten, die bei der Unterstützung verschiedener Länder und Sprachen zu beachten sind, werden in Kapitel 13 thematisiert.
- **Teil III »Wichtige Neuerungen in Java 12 bis 15«** – In diesem Teil gebe ich einen Überblick zu den Neuerungen aus verschiedenen Java-Versionen bis hin zu dem im September 2020 aktuellen Release 15. Dazu startet Kapitel 14 mit Erweiterungen in der Syntax. Danach werden diverse Ergänzungen in verschiedenen APIs sowie in der JVM vorgestellt. Insbesondere beschreibe ich auch das JMH-Framework zum Erstellen und Durchführen von Microbenchmarks.
- **Teil IV »Modularisierung«** – Die umfangreichste Neuerung in Java 9 war wohl die Modularisierung: Sowohl das JDK als auch eigene Programme lassen sich damit strukturieren und in Module untergliedern. Kapitel 15 behandelt dieses Themengebiet.

Nach der Lektüre dieser Teile sind Sie programmiertechnisch fit und bereit für das Schreiben eigener Anwendungen mit komplexeren Aufgabenstellungen. Auf dem Weg zu guter Software werden Sie aber vermutlich über das eine oder andere Problem stol-

pern. Wie Sie mögliche Fallstricke erkennen und beheben, ist Thema der folgenden Teile.

- **Teil V »Fallstricke und Lösungen im Praxisalltag«** – Der fünfte Teil beschreibt anhand von Beispielen mögliche Probleme aus dem Praxisalltag und stellt passende Lösungen vor. Auf diese Weise wird ein tieferes Verständnis für einen guten Softwareentwurf erlangt. Kapitel 16 betrachtet zunächst ausführlich mögliche Programmierprobleme, sogenannte »Bad Smells«. Diese werden analysiert und Lösungsmöglichkeiten dazu aufgezeigt. Diverse Umbaumaßnahmen werden in Kapitel 17 als Refactorings vorgestellt. Kapitel 18 rundet diesen Teil mit der Präsentation einiger für den Softwareentwurf wichtiger Lösungsideen, sogenannter Entwurfsmuster, ab. Diese sorgen zum einen dafür, ein Problem auf eine dokumentierte Art zu lösen, und zum anderen, Missverständnisse zu vermeiden, da die Entwickler eine eigene, gemeinsame Designsprache sprechen.
- **Teil VI »Qualitätssicherungsmaßnahmen«** – Qualitätssicherung ist für gute Software elementar wichtig und wird in diesem Teil VI vorgestellt. Einige im Kapitel zu Bad Smells gewonnene Erkenntnisse werden in Kapitel 19 zu einem Regelwerk beim Programmieren, sogenannten Coding Conventions, zusammengefasst. Um neue Funktionalität und Programmänderungen abzusichern, schreiben wir Unit Tests. Nur durch eine breite Basis an Testfällen haben wir die Sicherheit, Änderungen ohne Nebenwirkungen auszuführen. Kapitel 20 geht detailliert darauf ein. Eine Qualitätskontrolle über Codereviews wird in Kapitel 21 thematisiert. Zu einer guten Qualität gehören aber auch nicht funktionale Anforderungen. Diese betreffen unter anderem die Performance eines Programms. In Kapitel 22 stelle ich daher einige Techniken zur Performance-Steigerung vor.
- **Anhang** – Zum besseren Verständnis der Abläufe beim Ausführen eines Java-Programms stellt Anhang A einige Grundlagen zur Java Virtual Machine vor.

1.3 Konventionen und ausführbare Programme

Verwendete Zeichensätze

Im gesamten Text gelten folgende Konventionen bezüglich der Schriftart: Der normale Text erscheint in der vorliegenden Schriftart. Dabei werden wichtige Textpassagen *kursiv* oder ***kursiv und fett*** markiert. Englische Fachbegriffe werden eingedeutscht großgeschrieben. Zusammensetzungen aus englischen und deutschen (oder eingedeutschten) Begriffen werden mit Bindestrich verbunden, z. B. Plugin-Manager. Namen von Refactorings, Bad Smells, Idiomen sowie Entwurfsmustern u. Ä. werden bei ihrer Verwendung in KAPITÄLCHEN dargestellt. Sourcecode-Listings sind in der Schrift `courier` gesetzt, um zu verdeutlichen, dass dieser Text einen Ausschnitt aus einem realen Java-Programm wiedergibt. Auch im normalen Text werden Klassen, Methoden, Konstanten und Übergabeparameter in dieser Schriftart dargestellt.

Verwendete Abkürzungen

Im Buch verwende ich die in Tabelle 1-1 aufgelisteten Abkürzungen. Weitere Abkürzungen werden im laufenden Text in Klammern nach ihrer ersten Definition aufgeführt.

Tabelle 1-1 *Verwendete Abkürzungen*

Abkürzung	Bedeutung
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
GUI/UI	(Graphical) User Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JLS	Java Language Specification
JRE	Java Runtime Environment
JSR	Java Specification Request
JVM	Java Virtual Machine
OO	Objektorientierung
TDD	Test-Driven Development
UML	Unified Modeling Language
XML	Extensible Markup Language

Verwendete Java-Umgebungen und -Versionen

Wir werden uns in diesem Buch vorwiegend mit Beispielen aus dem Bereich der Java Standard Edition (Java SE) auseinandersetzen. Davon gibt es verschiedene Versionen. Seit September 2020 ist Version 15 aktuell, allerdings stellt diese kein sogenanntes LTS-Release (LTS = Long Term Support) dar, sondern dabei handelt es sich lediglich um eine kurzzeitig aktuelle Version von Java.

Weil aber ein Großteil der kommerziellen Java-Projekte auf LTS-Versionen beruht, bildet JDK 11 die Grundlage für die Beispiele im Buch – natürlich abgesehen von den Kapiteln zu den neueren Java-Versionen 12 bis 15.

Verwendete Klassen aus dem JDK

Werden Klassen des JDKs zum ersten Mal im Text erwähnt, so wird deren voll qualifizierter Name, d. h. inklusive der Package-Struktur, angegeben: Für die Klasse `String` würde dann etwa `java.lang.String` notiert. Dies erleichtert ein Auffinden im JDK. Im darauffolgenden Text wird zur besseren Lesbarkeit auf diese Angabe verzichtet und

nur der Klassenname genannt. Zudem sind aus Platzgründen in den Listings nur selten `import`-Anweisungen abgebildet.

Im Text beschriebene Methodenaufrufe enthalten in der Regel die Typen der Übergabeparameter, etwa `substring(int, int)`. Sind die Parameter in einem Kontext nicht entscheidend, wird auf deren Angabe aus Gründen der besseren Lesbarkeit verzichtet oder aber durch die Zeichenfolge `...` abgekürzt.

Download, Sourcecode und ausführbare Programme

Um den Rahmen des Buchs nicht zu sprengen, stellen die abgebildeten Programmlistings häufig nur Ausschnitte aus lauffähigen Programmen dar. Deren Formatierung weicht leicht von den Coding Conventions von Oracle[1] ab. Ich orientiere mich an denjenigen von Scott Ambler[2], der insbesondere (öffnende) Klammern in jeweils eigenen Zeilen vorschlägt. Dazu habe ich ein spezielles Format namens `Michaelis_CodeFormat` erstellt, dessen Anwendung nachfolgend gezeigt ist.

Abbildung 1-1 *Konfiguration von Java Code Style –> Formatter*

[1] *http://www.oracle.com/technetwork/java/codeconv-138413.html*
[2] *http://www.ambysoft.com/essays/javaCodingStandards.html*

Download Dieser Formatter steht ebenso wie der Sourcecode der Beispiele auf der Webseite

www.dpunkt.de/java-profi

zum Download bereit und ist in ein Eclipse-Projekt integriert. Viele der Programme lassen sich mithilfe von Gradle-Tasks (die wir in Kapitel 2 kennenlernen) ausführen. Deren Name wird in Kapitälchenschrift, etwa LOCALEEXAMPLE, angegeben.

Nacharbeiten nach Projekt-Import Nach dem erstmaligen Import müssen die Abhängigkeiten auf die externen Bibliotheken im Eclipse-Projekt mit dem Kommando `gradle cleanEclipse eclipse` neu initialisiert und auf Ihren Rechner aktualisiert werden.

I Java-Grundlagen, Analyse und Design

2 Professionelle Arbeitsumgebung

Das Bearbeiten von Sourcecode ist eine wichtige und elementare Aufgabe bei der Softwareentwicklung. Genau wie Handwerker benötigen auch wir Softwareentwickler eine gut strukturierte Werkbank (Arbeits-/Entwicklungsumgebung) mit nützlichen Werkzeugen (Tools). In diesem Kapitel werden wir damit beginnen, eine Arbeitsumgebung einzurichten, die sowohl das Entwickeln erleichtert als auch Unterstützung beim Testen bietet und dadurch hilft, Fehler zu reduzieren.

Den zentralen Anlaufpunkt, unsere Steuerzentrale, bildet eine sogenannte integrierte Entwicklungsumgebung, kurz IDE. Ich motiviere in Abschnitt 2.1 den Einsatz einer solchen und nenne drei mögliche Kandidaten, wobei in diesem Buch die Wahl auf Eclipse als frei verfügbare IDE fällt. In Abschnitt 2.2 betrachten wir zwei Vorschläge zur Organisation eines Softwareprojekts, also zur Strukturierung von Sourcecode und anderen Dateien (Bilder, Texte, Testcode usw.). Abschnitt 2.3 beschäftigt sich dann mit dem Thema Versionsverwaltungen. Diese helfen dabei, die Dateien eines Projekts sicher zu speichern und verschiedene Versionen davon abrufen zu können. Dazu werde ich einführend auf zentrale und dezentrale Versionsverwaltungen am Beispiel der Open-Source-Tools CVS, Subversion (SVN) sowie Git und Mercurial eingehen. Die professionelle Softwareentwicklung umfasst nicht nur die Implementierung und die spätere Auslieferung, sondern vor allem auch das Testen unserer Programme. In Abschnitt 2.4 werfen wir daher einen kurzen Blick auf das Erstellen von Unit Tests mithilfe von JUnit. Selbst wenn viele Unit Tests existieren und unsere Programme sehr gewissenhaft testen, so wird es doch immer mal wieder zu unerklärlichem Programmverhalten oder gar Fehlern kommen. Für solche Fälle ist es zur Fehlersuche wünschenswert und hilfreich, das Programm schrittweise ausführen zu können, es bei Bedarf an einer bestimmten Stelle zu unterbrechen und dann die Wertebelegungen von Variablen überprüfen zu können. Das Ganze ist mithilfe eines sogenannten Debuggers möglich. Eine Einführung in die Thematik bietet Abschnitt 2.5. Ausgewählte lauffähige Stände wollen wir an Kunden veröffentlichen. Abschnitt 2.6 gibt einen Überblick über die Auslieferung (Packaging und Deployment) von Java-Programmen. Dazu lernen wir JAR-Dateien (Java Archive) kennen. Abschließend stelle ich die Vorteile eines von der IDE unabhängigen Build-Prozesses heraus, mit dem es möglich ist, Programme zu kompilieren, zu testen, zu starten, Auslieferungen zu erzeugen usw. In Abschnitt 2.7 beschreibe ich dazu die Build-Tools Maven und insbesondere Gradle. Letzteres wird zur Ausführung aller im Buch vorgestellten Beispielapplikationen genutzt.

2.1 Vorteile von IDEs am Beispiel von Eclipse

Zum Bearbeiten von Sourcecode empfehle ich den Einsatz einer IDE anstelle von Texteditoren. Zwar kann man für kleinere Änderungen auch mal einen Texteditor nutzen, aber dieser bietet nicht die Annehmlichkeiten einer IDE: Dort können Kompiliervorgänge und Sourcecodeanalysen automatisch und im Hintergrund stattfinden, wodurch gewisse Softwaredefekte direkt noch während des Editierens erkannt und angezeigt werden können, etwa in einer To-do-/Task-Liste. IDEs bereiten zudem vielfältige Informationen auf. Das erlaubt unter anderem die Anzeige von Ableitungshierarchien und das Auffinden von Klassen über deren Namen. Auch das Verknüpfen der JDK-Klassen mit deren Sourcecode und das Anzeigen zugehöriger Dokumentation sind Vorteile von IDEs. Weiterhin werden automatische Transformationen und Änderungen von Sourcecode, sogenannte ***Refactorings***, unterstützt.

Für Java bieten sich verschiedene IDEs an. Sowohl Eclipse als auch NetBeans sind kostenlos. IntelliJ IDEA gibt es als kostenlose Community Edition sowie als kostenpflichtige Ultimate Edition. Alle IDEs haben ihre speziellen Vorzüge, aber auch Nachteile. Entscheiden Sie selbst und besuchen Sie dazu folgende Internetadressen:

- `www.eclipse.org`
- `www.jetbrains.com/idea`
- `www.netbeans.org`

Die genannten IDEs lassen sich durch die Integration von Tools, etwa Sourcecode-Checkern, Versionsverwaltungen, XML-Editoren, Datenbank-Tools, Profiling-Tools usw., erweitern. Im folgenden Text werde ich speziell auf Eclipse und entsprechende Tools eingehen. Nach der Lektüre dieses Kapitels haben Sie dann bereits eine Arbeitsumgebung, die professionelles Entwickeln ermöglicht. Im Verlauf des Buchs werden thematisch passende, arbeitserleichternde Erweiterungen vorgestellt.

Basiskonfiguration für Eclipse

Zur Fehlervermeidung und Qualitätssicherung empfiehlt es sich, den Sourcecode regelmäßig zu analysieren und dabei die Einhaltung gewisser Regeln und Standards zu forcieren. In einem ersten Schritt kann man dazu auf die in Eclipse integrierte Sourcecode-Prüfung zurückgreifen. Umfangreichere Tests bieten Tools wie Checkstyle, FindBugs und PMD (vgl. Abschnitt 19.4).

Die in Eclipse integrierten Sourcecode-Prüfungen lassen sich im Einstellungsdialog WINDOW –> PREFERENCES konfigurieren. Dort wählen wir im Baum den Eintrag JAVA –> COMPILER –> ERRORS/WARNINGS. Im zugehörigen Dialog nehmen wir Anpassungen in den Bereichen CODE STYLE, POTENTIAL PROGRAMMING PROBLEMS, UNNECESSARY CODE sowie NULL ANALYSIS vor. Abbildung 2-1 zeigt eine mögliche, sinnvolle Einstellung der Werte im Abschnitt CODE STYLE. Auch in den anderen Sektionen können Sie bei Interesse strengere Auswertungen wählen. Experimentieren Sie ruhig ein wenig mit den Einstellungen.

Abbildung 2-1 *Konfiguration von Java –> Compiler –> Errors/Warnings –> Code style*

Code style Methoden, die genauso heißen wie die Klasse selbst, also dadurch sehr leicht mit einem Konstruktor verwechselt werden können, werten wir als `Error`. Ein unqualifizierter Zugriff auf Attribute wird ignoriert, genauso wie nicht-externalisierte Strings und die Möglichkeit, Methoden `static` zu machen. Alle anderen Werte setzen wir auf `Warning`.

Potential programming problems Konvertierungen mithilfe von Auto-Boxing und -Unboxing stellen wir auf `Warning`. Eine versehentliche Zuweisung in einer booleschen Bedingung betrachten wir als `Error`. Alles andere stellen wir auf `Warning`.

Unnecessary code Unnötige `else`-Anweisungen sollen ignoriert werden, da es lediglich eine Stilfrage ist, ob man bei Auswertung einer Bedingung durch das `else` die Behandlung des anderen Zweiges darstellen möchte oder nicht. Einen ungenutzten Exception-Parameter setzen wir auf `ignore`. Die restlichen Prüfungen stellen wir auf `Warning`.

Null analysis Um uns vor Problemen mit unerwarteten `null`-Werten und Zugriffen darauf zu bewahren, stellen wir in dieser Sektion alles mindestens auf `Warning`.

2.2 Projektorganisation

Im Sourcecode erleichtert eine einheitliche Formatierung und eine sinnvolle Namensgebung die Verständlichkeit und gute Lesbarkeit. Wenn man das Ganze auf die Dateien eines Projekts überträgt, erleichtert eine einheitliche Projektstruktur die Orientierung in fremden (und auch in eigenen) Projekten. Daher stelle ich nun zwei bewährte Varianten vor, die Sie als Vorschlag für die Verzeichnisstruktur eigener Projekte nutzen können.

Als Erstes gehe ich auf die Projektstruktur ein, die entsteht, wenn man mit Eclipse ein Java-Projekt anlegt. Als Zweites beschreibe ich eine Projektstruktur, die sich durch die Verbreitung von Maven als Build-Tool als De-facto-Standard etabliert hat. Die Beispiele dieses Buchs folgen dieser zweiten Konvention.

2.2.1 Projektstruktur in Eclipse

Wenn man Projekte mit Eclipse anlegt und deren Inhalt verwaltet, so verwendet man ein Hauptverzeichnis mit dem Namen oder Synonym des Projekts. Nahezu alle anderen Daten werden in Unterordnern abgelegt. Lediglich einige (in Eclipse zunächst ausgeblendete) Metadateien (`.project` und `.classpath`) findet man im Hauptverzeichnis. Die Projektstruktur wird zunächst grafisch dargestellt und danach kurz beschrieben:

```
+ <project-root>
|
+---src                           - Sourcecode unserer Applikation
|   +---ui                        - Grafische Oberfläche
|       +---FileDialog.java       - Die Klasse FileDialog
|
+---test                          - Sourcecode der Unit Tests
|   +---ui                        - Tests für die grafische Oberfläche
|       +---FileDialogTest.java   - Tests für die Klasse FileDialog
|
+---lib                           - Externe Klassenbibliotheken (JARs)
|   +---junit_4.11                - Ordner zur Strukturierung
|       +---junit.jar             - Wichtig zum Übersetzen der Unit Tests
|       +---hamcrest-core.jar     - Von JUnit benötigt
|
+---config                        - Konfigurationsdateien
|   +---images                    - Bilder
|   +---texts                     - Sprachressourcen
|
+---docs                          - Dokumentation
+---generated-reports             - Erzeugte Berichte (JUnit, Checkstyle usw.)
+---bin                           - Kompilierte Klassen und JAR der Applikation
```

Sourcecode und Tests Den Sourcecode legen wir im Verzeichnis `src` ab. Darunter erfolgt die Aufteilung in Form von Packages. Parallel dazu werden Tests in einem Ordner `test` abgelegt. Dabei verwenden wir hier eine Spiegelung der Package-Struktur des `src`-Ordners. Das bietet zweierlei Vorteile: Erstens trennt man so Tests und Applikationscode im Dateisystem, wodurch sich ungewünschte Abhängigkeiten leicht erkennen lassen und später bei Auslieferungen die Testklassen nicht Bestandteil des Programms sein müssen. Zweitens liegen die Tests dadurch logisch in gleichen

Packages wie der korrespondierende Sourcecode,[1] was den Zugriff auf alle Elemente des Packages (außer den privaten) möglich macht.

Weitere Dateien In der Regel nutzt man zur Realisierung von Projekten verschiedene Klassenbibliotheken. Selbst in diesem einfachen Beispiel besitzen wir durch die Testklasse eine Abhängigkeit zur JUnit-Bibliothek. JUnit wird durch Eclipse bereits mitgeliefert und automatisch verwaltet. Andere Bibliotheken (oder eine aktuellere Version von JUnit) werden in einem Ordner `lib` gesammelt. Dabei erleichtert eine gut gewählte Verzeichnishierarchie die Übersicht über die verwendeten Bibliotheken und deren Versionen. Häufig sind für ein Projekt auch verschiedene Konfigurationsdateien zu verwalten. Dazu bietet sich ein Ordner namens `config` an. Dort können z. B. Textressourcen für Sprachvarianten und Konfigurationen für Logging usw. abgelegt werden. Verschiedene Arten von Dokumentation speichert man im Verzeichnis `docs`.

Tipp: Bibliotheken (JARs) in Eclipse einbinden

Wenn Sie Klassen aus Bibliotheken nutzen wollen, so liegen diese in Form von JARs (Java Archive) vor. Um diese in einem Eclipse-Projekt zu nutzen, gehen Sie wie folgt vor: Klicken Sie auf Ihr Projekt und wählen Sie im Kontextmenü BUILD PATH –> CONFIGURE BUILD PATH und in dem erscheinenden Dialog wählen Sie links den Eintrag JAVA BUILD PATH und dort den Tab LIBRARIES. Mithilfe der Buttons ADD JAR… und ADD EXTERNAL JAR… können Sie die gewünschten Bibliotheken einbinden. Ersteres wählen Sie, wenn sich die JAR-Dateien innerhalb Ihres Projekts (z. B. im Verzeichnis `lib`) befinden. Mithilfe von ADD EXTERNAL JAR… kann man JARs auch aus externen Verzeichnissen einbinden, etwa einem dedizierten Installationsverzeichnis einer Datenbank o. Ä.

Beim professionellen Programmieren sollte die Verwaltung von Abhängigkeiten und das Einbinden von Fremdbibliotheken über ein Build-Tool wie Maven oder Gradle erfolgen. Details dazu beschreibt Abschnitt 2.7.

Generierte Dateien Gemäß der Faustregel »***Trenne generierte Dateien von regulärem Sourcecode***« werden generierte Dateien in separaten Verzeichnissen abgelegt. Vom Compiler generierte `.class`-Dateien liegen im Ordner `bin`. Durch Tests und andere Sourcecode-Prüfungen entstehende Berichte werden in einem Verzeichnis `generated-reports` gesammelt. Diese Aufteilung erleichtert die Übersicht und Trennung, was wiederum die später beschriebene Versionsverwaltung sowie die Automatisierung von Build-Läufen vereinfacht.

[1] Wenn man in der IDE den Ordner `test` als zusätzliches Sourcecode-Verzeichnis wählt.

Auslieferungen und Releases

Normalerweise wird eine Applikation nicht nur innerhalb der IDE laufen, sondern vor allem als eigenständige Applikation. Häufig sollen davon auch verschiedene Versionen, sogenannte ***Releases***, erzeugt werden. Einige davon stellen stabile Softwarestände dar und können als offizielle Version oder ***Auslieferung*** an Kunden übergeben werden.

Für die aktuelle Programmversion bietet sich die Speicherung in einem Ordner `release` an. Damit die Applikation tatsächlich unabhängig von der IDE ausgeführt werden kann, müssen im Ordner `release` alle benötigten externen Bibliotheken, Konfigurationsdateien usw. bereitgestellt werden (z. B. durch Kopie) oder zugreifbar sein. Deren Pfade müssen in den sogenannten `CLASSPATH` aufgenommen werden. Dies ist die Menge von Verzeichnissen und Dateien, in der die JVM nach Klassen und anderen Ressourcen, etwa Bildern, sucht und die als Startparameter der JVM gesetzt werden kann. Auf das Classloading gehe ich kurz in Anhang A ein.

```
+ <project-root>
|
+---release                - Release-Ordner
    +---lib                - Kopie des lib-Verzeichnisses
    +---config             - Kopie des config-Verzeichnisses
    +---app.jar            - Applikation als JAR
```

Schwachpunkte der gezeigten Projektstruktur

Die dargestellte Verzeichnisstruktur eignet sich für viele Projekte recht gut, besitzt aber Schwachstellen. Zunächst einmal muss die Verzeichnisstruktur (bzw. Teile davon) für jedes Projekt erneut von Hand angelegt und auch gepflegt werden. Dabei besteht die Gefahr, dass sich Inkonsistenzen einschleichen: Heißt der Ordner mit den entstehenden Klassen *bin* oder *build*? Und wie derjenige mit den Fremdbibliotheken? In einem Projekt etwa *lib*, im anderen *libs*. Das setzt sich bei der Vergabe der Namen von Unterordnern fort: Wird ein solcher *junit4* genannt oder *junit4.11* oder aber *junit_4.11*?

2.2.2 Projektstruktur für Maven und Gradle

Nachfolgend schauen wir uns die von Maven und Gradle genutzte und weitverbreitete Verzeichnisstruktur als Alternative zu der von Eclipse standardmäßig verwendeten an.

Einheitliches Projektlayout

Das Problem möglicher Inkonsistenzen bezüglich der genutzten Verzeichnisse für kompilierte Klassen, Fremdbibliotheken oder Reports usw. wird von den beiden Build-Tools Maven und Gradle adressiert, indem diese eine einheitliche Verzeichnisstruktur für alle Projekte fordern. Dadurch ist immer klar, wo sich Dateien gewünschten Inhalts befinden, etwa für Sourcen und Tests in den Verzeichnissen `src/main/java` bzw. `src/test/java`. Darüber hinaus benötigte Ressourcendateien werden wiederum in getrennten Verzeichnissen hinterlegt, nämlich wie folgt:

```
+ <project-root>
|
+---src
    +---main
    ¦   +---java            - Java-Klassen
    ¦   +---resources       - Konfigurationsdateien
    ¦
    +---test
        +---java            - Testklassen
        +---resources       - Konfigurationsdateien für Testklassen
```

Projektlayout am Beispiel Nutzen wir die obige Projektstruktur für eine einfache Hello-World-Applikation, so ergibt sich folgendes Verzeichnislayout, unter der Annahme, dass wir die Applikation mit einer Klasse `App.java` und die dazugehörige Testklasse `AppTest.java` im Package `de.javaprofi.helloworld` realisieren:

```
+---src
    +---main
    ¦   +---java
    ¦       +---de
    ¦           +---javaprofi
    ¦               +---helloworld
    ¦                       App.java
    ¦
    +---test
        +---java
            +---de
                +---javaprofi
                    +---helloworld
                            AppTest.java
```

Wenn man das Projekt mit Maven kompiliert und paketiert (`mvn package`), so entsteht ein Verzeichnis `target` als Ziel für kompilierte Klassen und weitere erzeugte Dateien wie z. B. Testresultate. Diese Struktur ist nachfolgend zum besseren Verständnis auszugsweise und gekürzt abgebildet:

```
+---target
    ¦   helloworld-1.0-SNAPSHOT.jar
    ¦
    +---classes
    ¦   +---de
    ¦       +---javaprofi
    ¦           +---helloworld
    ¦                   App.class
    ¦
    +---surefire-reports
    ¦       de.javaprofi.helloworld.AppTest.txt
    ¦       TEST-de.javaprofi.helloworld.AppTest.xml
    ¦
    +---test-classes
        +---de
            +---javaprofi
                +---helloworld
                        AppTest.class
```

Abweichungen im Projektlayout mit Gradle Gradle verwendet für Sourcen, Tests und Ressourcen die Maven-Konventionen zur Strukturierung der Verzeichnisse. Allerdings besitzen die erzeugten Verzeichnisse einen leicht abweichenden Standard. Es entsteht ein Verzeichnis `build` mit verschiedenen Unterverzeichnissen wie folgt:

```
+ <project-root>
¦
+---build
¦   +---classes
¦   ¦   +---main
¦   ¦   ¦   +---de
¦   ¦   ¦       +---javaprofi
¦   ¦   ¦           +---helloworld
¦   ¦   ¦                   App.class
¦   ¦   ¦
¦   ¦   +---test
¦   ¦       +---de
¦   ¦           +---javaprofi
¦   ¦               +---helloworld
¦   ¦                       AppTest.class
¦   ¦
¦   +---dependency-cache
¦   +---libs
¦   ¦       helloworld.jar
¦   ¦
¦   +---reports
¦   ¦   +---tests
¦   ¦       ¦   index.html
...
```

Verwaltung externer Abhängigkeiten

Eine Sache könnte uns noch wundern: Wo und wie werden denn die Abhängigkeiten zu externen Bibliotheken beschrieben? Wir finden in der Verzeichnisstruktur keinen Ordner mit Bibliotheken. Exakt! Während wir für das Eclipse-Projekt bzw. bei Builds mit Ant diese Verwaltung noch selbst erledigen mussten, helfen uns hier die Build-Tools Maven und Gradle, indem sie sich um die Auflösung und Bereitstellung externer Bibliotheken kümmern. Details dazu lernen wir später in Abschnitt 2.7 kennen.

2.3 Einsatz von Versionsverwaltungen

Nachdem wir zwei Alternativen zur Strukturierung von Projekten kennengelernt haben, widmen wir uns nun dem Thema Versionsverwaltungen. Diese ermöglichen die Speicherung und Verwaltung aller relevanten Dateien eines Softwareprojekts inklusive deren Historie (vgl. dazu den folgenden Praxistipp). Dazu werden diese Dateien an einem zentralen Ort, dem sogenannten ***Repository***, abgelegt. Der Vorteil davon ist, dass sich hier nicht nur die aktuelle Version einer Datei befindet, sondern deren gespeicherte Historie zur Verfügung steht. Im Repository sind also frühere Stände enthalten, einsehbar und bei Bedarf wiederherstellbar. Das ist insofern von Bedeutung, weil man so

auch einmal Experimente vornehmen und nötigenfalls zu einem älteren, funktionierenden Stand zurückkehren kann.

Darüber hinaus lassen sich verschiedene, voneinander unabhängige Entwicklungslinien, sogenannte ***Branches***, verwalten. Diese unterstützen z. B. Umbaumaßnahmen, spezielle Features oder Experimente. Die Hauptentwicklungslinie wird ***Stamm***, ***Hauptast*** oder auch ***master*** bzw. ***default*** genannt. Der aktuellste Stand einer Datei auf einer Entwicklungslinie heißt ***HEAD***. Änderungen und Erweiterungen, die auf Branches entwickelt wurden, können über einen ***Merge***-Vorgang in den aktuellen Stand integriert werden. Diese Abläufe deutet Abbildung 2-2 an.

Abbildung 2-2 *Dateiversionsbaum*

Hilfreich ist, dass sich bei Änderungen an Dateien die Unterschiede zu beliebigen anderen Versionen ermitteln lassen. Außerdem können Versionen bei der Speicherung im Repository mit Kommentaren versehen werden, was den Überblick über Änderungen in den jeweiligen Versionen erleichtert.

Tipp: Inhalt des Repository

Anhand der in Abschnitt 2.2 beschriebenen Projektstruktur könnte man auf die Idee kommen, einfach alle Dateien eines Projekts in das Repository aufzunehmen. Dies ist aber nur begrenzt sinnvoll. Abgesehen von wenigen Ausnahmen sollten all diejenigen Dateien, die während des Build-Prozesses entstehen, *nicht* gesichert werden, eben weil sie sich immer wieder problemlos erneut erzeugen lassen. Dazu gehören vor allem die generierten `.class`-Dateien oder Berichte bzw. Dokumentationen. Ausnahmen können spezielle Versionen davon sein, die man historisieren möchte.

Was sollte also gespeichert werden? Die Frage ist einfach zu beantworten: Alle Dateien, die sich nicht automatisch generieren lassen. Dazu gehören der Sourcecode, die Unit Tests, verschiedene Dokumentationen (UML-Diagramme, Textdokumente usw.), Konfigurationsdateien und benötigte Build-Skripte.

Motivation für Versionsverwaltungen

Jede Bank verwendet einen Tresor, um wertvolle Dinge zuverlässig zu schützen. Softwareentwicklung ohne Sicherung ist ähnlich unsicher wie eine Geldanlage in Omas Sparstrumpf. Also sichern Sie Ihren größten Schatz – Ihren Sourcecode – in einem virtuellen Safe! Falls Sie bis jetzt ohne Versionsverwaltung leben, so sollten Sie sich durch die folgenden zwei Situationsbeschreibungen davon überzeugen lassen, schnellstmöglich ein solches System zu nutzen. Ansonsten könnte jeder kleine Fehler fatale Folgen haben, was ich mit zwei Anekdoten verdeutlichen möchte:

- Ich erinnere mich noch gut an die Tage mit Sicherungsdisketten. Es war eine mühselige und fehlerträchtige Arbeit, die aktuellen Daten auf die entsprechende Diskette zu überspielen. Als ich einmal die Arbeit eines Tages sichern wollte, habe ich beim Kopieren Quelle und Ziel vertauscht und damit hatte ich den gesamten Tag umsonst gearbeitet. Außerdem hatte ich danach zwei ältere Softwarestände, von denen ich nicht wusste, welche Änderungen diese enthielten. Um solche Situationen zu vermeiden, sollte häufig ins Repository gesichert werden. Somit bleibt selbst bei Problemen der maximale Datenverlust, d. h. die nicht gesicherten Änderungen, immer überschaubar.
  Diese Anekdote adressiert einen anderen wichtigen Aspekt: ***Datensicherung***. Ein Repository ist zwar ein guter Platz, um Daten zentral abzulegen und gezielt wiederherstellen zu können. Man ist so aber lediglich gegen die »eigene Dummheit« beim Ändern des Sourcecodes geschützt. Für eine weiter gehende Sicherheit der Daten ist es jedoch elementar wichtig, regelmäßig ein Backup des Repository zu erstellen, um das Repository an sich zu sichern, etwa gegen Datenträgerfehler.
- Stellen Sie sich vor, es ist Freitagmorgen und Ihr Chef betritt aufgeregt den Raum und verlangt für den späten Nachmittag ein Release, basierend auf dem letzten Auslieferungsstand des Programms von vor vier Wochen mit genau zwei dedizierten Bugfixes. Ohne Versionsverwaltungssystem haben Sie jetzt ganz schlechte Karten, denn in der Zwischenzeit hat die gesamte Entwicklertruppe kräftig weitergearbeitet. Dies alles ohne Fehler wieder rückgängig zu machen ist nahezu unmöglich. Eine Sicherung des Sourcecodes gibt es zwar irgendwo, aber in der Hektik ist diese momentan nicht auffindbar oder sie ist auf der mobilen Festplatte des Kollegen, der natürlich gerade gestern krank geworden ist.

Die Moral von der Geschichte ist: Lassen Sie den Computer die Arbeit erledigen, die er viel besser kann als Sie: das Verwalten vieler Versionen.

Wie schon eingangs erwähnt, spricht neben den genannten Vorteilen noch ein weiteres wesentliches Argument für den Einsatz einer Versionsverwaltung: Man kann mithilfe von Branches sehr leicht experimentelle Versionen der Software entwickeln, ohne dabei Angst haben zu müssen, Auslieferungen oder die gesamte Weiterentwicklung zu stören. Verläuft ein solches Experiment erfolgreich, so kann man die Änderungen übernehmen. Ansonsten verwirft man sie einfach.

Varianten der Versionsverwaltung

Vor dem Aufkommen von dedizierten Programmen zur Versionsverwaltung wurden häufig die Dateien eines Projekts lediglich in ein anderes Verzeichnis kopiert – zweckmäßigerweise in ein Verzeichnis mit einem Zeitstempel im Namen. Diese händische Versionsverwaltung scheint ganz natürlich und ist auch recht einfach, aber doch ziemlich fehleranfällig. Darüber hinaus besteht ein entscheidender Nachteil: Wenn man herausfinden möchte, was sich zwischen Versionen geändert hat, so ist dies oftmals ziemlich mühselig festzustellen. Erschwerend kommt hinzu, dass mitunter versehentlich die zur Versionierung eigentlich notwendigen Kopien ausbleiben und somit gewisse Zwischenstände nicht gesichert sind. Dann besitzt man nur eine unvollständige Historie.

Um diesen Nachteilen zu begegnen und Änderungen an den Dateien eines Projekts feingranular speichern und später nachvollziehen zu können, wurden Versionsverwaltungssysteme erfunden. Man unterscheidet zwischen ***zentralen*** und ***dezentralen Versionsverwaltungen***. Schon vor einigen Jahrzehnten entstanden zentrale Versionsverwaltungen (Version Control System, VCS). Dort erfolgt die Versionsverwaltung auf einem zentralen Server und erfordert von den Nutzern (Clients) somit (Netzwerk-)Zugriff darauf. Dezentrale Versionsverwaltungen (Distributed Version Control System, DVCS) kombinieren die Vorteile der beiden gerade beschriebenen Varianten: Sie sind nahezu so einfach in der Handhabung wie des Erstellen lokaler Kopien und bieten darüber hinaus leistungsstarke Versionsverwaltungsfunktionalität.

2.3.1 Arbeiten mit zentralen Versionsverwaltungen

Während die lokale Versionsverwaltung durch Kopieren per Hand selbst schon für eine Person recht schnell an Grenzen stößt, so gilt dies umso mehr bei der Zusammenarbeit und Verwaltung von Änderungen im Team, weil hier deutlich mehr (eventuell auch konkurrierende) Änderungen und damit auch Abstimmungsbedarf entstehen. Als Abhilfe wurden zentrale Versionsverwaltungen entwickelt. Bekannte zentrale Versionsverwaltungen sind das Concurrent Versions System (CVS) und Subversion (SVN). CVS ist älter und besitzt einige Einschränkungen, etwa bei Namensänderungen. SVN wurde als Nachfolger neu entwickelt und unterstützt diverse Dinge besser als CVS.

Repository

Wie bereits erwähnt, werden die Dateien eines Projekts mit ihrer gesamten Historie an einer zentralen Stelle, dem ***Repository***, gespeichert. Gewöhnlich liegt dieses Repository auf einem dedizierten Server und dort wird auch der Serveranteil zur Versionsverwaltung ausgeführt. Zum Zugriff darauf benötigt man als Nutzer eine spezielle Clientsoftware, mit der alle Aktionen zur Versionsverwaltung erfolgen, z. B. das Hinzufügen, Ändern oder Aufbereiten der Versionshistorie. Diese Aktionen erfordern jeweils Zugriffe auf das Repository. Der dezentrale Client-Server-Ansatz erlaubt es, an verschiedenen Orten verteilt an einem Projekt zu arbeiten, setzt aber Netzwerkzugriff voraus.

Arbeitsablauf

Zum Bearbeiten eines Versionsstands überträgt ein Entwickler den gewünschten Stand aller Dateien eines Projekts aus dem Repository in ein Arbeitsverzeichnis auf seinem Rechner. Diesen Vorgang bezeichnet man als »***Auschecken***« (Check-out). Anschließend arbeitet man auf lokalen Kopien der Dateien, der sogenannten ***Working Copy***, und nimmt dort Änderungen vor. Sind diese abgeschlossen oder haben diese einen stabilen Zwischenstand erreicht, so sollte eine Integration in das Repository erfolgen. Dadurch werden die Änderungen allen anderen Kollegen zugänglich gemacht. Dieser Vorgang wird »***Einchecken***« (Check-in) oder auch ***Commit*** genannt. Damit die neu eingespielten Änderungen allerdings für die Kollegen in ihrem Arbeitsbereich tatsächlich sichtbar werden, müssen diese zunächst einen Abgleich mit dem Repository durchführen. Diesen Abgleich nennt man ***Update*** oder ***Synchronize***. Die beschriebenen Arbeitsabläufe visualisiert Abbildung 2-3.

Abbildung 2-3 *Arbeitsablauf mit zentralen Versionsverwaltungen*

Besonderheiten beim Abgleich Da die eigenen Änderungen nur auf lokalen Kopien der Originale erfolgen, kann es zu Problemen beim Abgleich kommen, wenn mehrere Entwickler die gleiche Datei verändert haben. In CVS werden bei einem ***Update*** die lokalen Änderungen mit der aktuellen Version aus dem Repository überschrieben. Ein ***Synchronize*** versucht die Änderungen der lokalen Version mit denen aus dem Repository abzugleichen. Dies funktioniert gut, wenn unterschiedliche Teile einer Datei von Änderungen betroffen sind. SVN integriert Änderungen bei einem Update automatisch, wenn aufgrund der Differenzanalyse keine Probleme festgestellt wurden. Widersprechen sich Änderungen an der lokalen Datei und deren Version im Repository, so muss man steuernd eingreifen. Eine solche Situation wird ***Konflikt*** genannt. Eine betroffene Datei lässt sich erst ins Repository integrieren, wenn alle dort gefundenen Konflikte behoben sind. Die Zusammenführung verschiedener Änderungen wird als ***Merge*** bezeichnet.

Tagging und Branching

Gehen wir noch einmal zu dem Beispiel zurück, in dem eine Auslieferung des Systems, basierend auf dem letzten Release sowie ein paar dringend benötigten Fehlerbehebungen, gewünscht wird. Sie brauchen nun Zugriff auf den Stand des Projekts genau so, wie es ausgeliefert wurde. Beispielsweise lassen sich Dateien anhand ihrer Versionsnummer zurückholen. Dieser Vorgang ist für CVS aufwendig, da die Versionsnummer von Datei zu Datei variiert.[2] SVN bietet mehr Komfort, da die Versionsnummer für alle Dateien des Repository einheitlich behandelt wird (siehe folgenden Praxistipp).

Alternativ können Dateien anhand eines speziellen Datums zurückgeholt werden. Dies ist nützlich, wenn wenige Dateien betroffen sind oder nur der zeitliche Verlauf von Interesse ist. Normalerweise soll ein Projekt jedoch wieder in einen Zustand gebracht werden, in dem es zu einem bestimmten Zeitpunkt oder Ereignis war, beispielsweise einer Auslieferung oder des Abschlusses größerer Änderungen. Das führt uns zum sogenannten Tagging, was wir uns nun anschauen.

Tipp: Versionsnummern in SVN

Die Versionsnummer einer Datei und von Verzeichnissen entspricht immer der Versionsnummer des Projekts zum Zeitpunkt der Änderung. Für Verzeichnisse ist dies die höchste Versionsnummer der enthaltenen Dateien und Verzeichnisse. Dadurch kann die Folge der Versionsnummern einzelner Dateien oder Verzeichnisse auch Lücken haben. Beim Auschecken wird jeweils die größte Versionsnummer aus dem Repository geholt, die kleiner oder gleich der angeforderten Versionsnummer beim Auschecken ist.

[2] Es müssen dann die gesamten Dateien eines Projekts überprüft werden, um herauszufinden, welche Version jede Datei zum Zeitpunkt des gewünschten Stands hatte. Anschließend muss dann jede Datei einzeln anhand der ermittelten Versionsnummer wiederhergestellt werden.

Tagging Wenn man für Änderungen einen ganz speziellen Stand eines Projekts bearbeiten möchte, so wäre es umständlich und fehleranfällig, eine Menge von Dateien über deren Versionsnummer oder ein Datum zusammensuchen zu müssen. Als Abhilfe kann man zu einem beliebigen Zeitpunkt eine sogenannte ***Markierung*** (engl. ***Tag***) setzen. Vor größeren Änderungen und Auslieferungen sollte man dies in jedem Fall tun. Mithilfe von Markierungen lässt sich später ein Zwischenstand exakt wiederherstellen, um basierend darauf Fehlerkorrekturen durchführen zu können.

Man kann sich Markierungen wie eine Verbindungsschnur bzw. einen Zusammenschluss zwischen Versionen verschiedener Dateien vorstellen (vgl. Abbildung 2-4).

Abbildung 2-4 *Zwei Tags (miteinander verbundene Versionen von Dateien)*

Branching Unter einer Verzweigung oder einem ***Branch*** versteht man eine eigene Entwicklungslinie. Ein solcher Abzweig lässt sich von jeder beliebigen anderen Entwicklungslinie abspalten, oft geschieht dies aber vom Stamm, auch ***Main-Branch*** oder Trunk genannt. In einer solchen Entwicklungslinie entsteht eine eigene Historie, sodass parallel zu anderen Zweigen weitergearbeitet werden kann: Änderungen in einem Zweig haben keinen Einfluss auf andere Verzweigungen. Dadurch sind Branches hilfreich, um potenziell gefährliche oder umfangreiche Änderungen zu isolieren, bis sich diese stabilisiert haben.

Zur Motivation blicken wir auch hier wieder auf das vorherige Beispiel zurück. Der ältere Sourcecode-Stand konnte aufgrund einer gesetzten Markierung leicht wiederhergestellt werden. Auf dieser lokalen Kopie können Fehlerbehebungen erfolgen. Natürlich wäre es wünschenswert, die durchgeführten Änderungen im Repository zu sichern, damit man später auf diesem speziellen Stand weiterarbeiten kann. Dies ist aber nicht sinnvoll möglich, wenn kein Branch erzeugt wurde, weil man ansonsten neuere Versionen des Hauptasts mit einem älteren Stand überspielen würde. Das ist jedoch in der Regel nicht gewünscht. Als Abhilfe dienen Branches, die eine eigene, separate Historie bieten. Dadurch lassen sich Änderungen wieder sinnvoll ins Repository reintegrieren.

Tipps für die Arbeit mit Branches

Beim Erstellen von Branches sollte man etwas Vorsicht walten lassen. Unbedacht eingesetzt können zu viele Branches ein Projekt ins Chaos stürzen, da man schnell den Überblick verliert, was in welchem Branch erweitert oder korrigiert wurde. Als Folge divergiert die Sourcecode-Basis, sodass es mit der Zeit immer schwieriger wird, die Änderungen auf den Branches wieder korrekt zusammenzuführen.

Mit CVS bzw. SVN lassen sich Verbesserungen und Fehlerbehebungen eines Branches oftmals nur mit Mühe auf andere Branches übertragen. Allerdings ist nicht unbedingt die absolute Anzahl an Branches klein zu halten, sondern lediglich die Anzahl »aktiver« Branches, d. h., an denen gleichzeitig gearbeitet wird. Darüber hinaus sollten die Verzweigungstiefe und Komplexität der Branches überschaubar sein. Nur in wenigen Fällen ist es sinnvoll, von einer Verzweigung wiederum Verzweigungen zu erzeugen. Ein möglicher Grund dafür sind Wartungsaufgaben auf älteren Ästen.

Hält man die Anzahl parallel laufender Entwicklungen gering, so sinkt die Wahrscheinlichkeit für Konflikte beim Integrieren. Weniger Konflikte erreicht man auch dadurch, ***dass man Branches möglichst häufig (z. B. täglich oder sobald die Arbeiten einen stabilen Zwischenstand erreicht haben) mit dem Ursprungsbranch synchronisiert***, also die dortigen Änderungen in den eigenen Branch reintegriert.

Tipp: Namensgebung für Markierungen und Verzweigungen

Die Namen von Branches und Markierungen sollten einem einheitlichen Schema folgen und deren Bedeutung sollte offensichtlich sein. Der Name sollte alle wichtigen Informationen über den Softwarestand enthalten und einer Konvention, etwa `ZWECK_KUNDE_VERSION_DATUM`, folgen. Versionsnummern für Releases sollten die Versionsangaben `Major.Minor.Patchlevel` enthalten. Damit ergibt sich als Beispiel folgender Name: `auslieferung_Meyer_V2.7.13_20080612`. Für den Zweck sind zumindest folgende Kürzel sinnvoll:

- `entwicklung` für Branches für Entwicklungen
- `auslieferung` für Auslieferungen an Kunden
- `release` für funktionsfähige Zwischenstände der Software

Bitte beachten Sie, dass die kritischen Aussagen zum Branching insbesondere für VCS mit zentralem Repository gelten und in wesentlich geringerem Maße für DVCS, da diese speziell dafür ausgelegt sind, mit vielen Branches und Versionsständen umgehen zu können.

2.3.2 Dezentrale Versionsverwaltungen

Während die zuvor vorgestellten Systeme CVS und SVN beide eine zentrale Versionsverwaltung repräsentieren, sind heutzutage dezentrale Versionsverwaltungen, etwa Git[3] und Mercurial[4], immer beliebter. Git ist derzeit wohl die populärste dezentrale Versionsverwaltung und wurde von Linus Torvalds kurzerhand selbst geschrieben, da keine der herkömmlichen Versionsverwaltungen ausreichend Flexibilität und Funktionalität für die verteilte Entwicklung des Linux-Kernels bot. Insbesondere fehlte es an gutem Support für viele unabhängige Branches und das offline Arbeiten sowie für das Mergen oder Rücknehmen von verschiedenen Änderungen. Ein zum Einstieg lehrreiches interaktives Tutorial zu Git steht unter `https://try.github.io/` bereit. Dort lernt man das Anlegen eines Repository und das Arbeiten damit in einem geführten Ablauf.

Mercurial ist eine valide und von mir bevorzugte Alternative zu Git. Wieso? Für mich lässt sich Mercurial teilweise einfacher handhaben, wie es nachfolgend im Meinungskasten dargestellt wird.

Meinung: Git vs. Mercurial

Im Prinzip sind Git und Mercurial funktional in etwa gleich mächtig: Bei der Integration in IDEs gibt es aber Unterschiede: Oft wird Git direkt unterstützt und Mercurial mitunter nur durch Plugins. Auf der Kommandozeile hinkt Git aber meiner Meinung nach in der Benutzbarkeit etwas hinterher, weil hier zu viele Implementierungsdetails durchscheinen. Unter Windows empfinde ich auch die grafische Integration von Mercurial (TortoiseHG[a]) in den Explorer ein wenig übersichtlicher als die von TortoiseGit[b]. Das zeigt sich in Kleinigkeiten: Commit-Kommentare werden standardmäßig bei Git über den vi eingegeben. Mercurial öffnet dagegen den vom System voreingestellten Texteditor. Entscheidend ist für mich aber die Art und Weise der Verwaltung und Angabe von Revisionsnummern. In Mercurial sind für den Benutzer Revisionsnummern natürliche Zahlen und es werden nicht die internen hexadezimalen Keys wie bei Git nach außen exponiert. Ich referenziere einfach lieber Revision 4711 als 00eda2a680893a20ea11d48ddd884812dc97a718. Basierend auf den Ausführungen wird klar, dass ich persönlich Mercurial bevorzuge, da es sich für mich natürlicher in der Handhabung anfühlt. Ich empfehle Ihnen, einfach mal beide Versionsverwaltungen zu installieren und damit ein wenig herumzuspielen, um die eigene Präferenz zu finden.

[a] *https://tortoisehg.bitbucket.io/*
[b] *https://tortoisegit.org/*

[3] *http://git-scm.com/downloads*
[4] *https://www.mercurial-scm.org/*

Repositories

Für die Arbeitsweise mit CVS und SVN haben wir nach der Lektüre der vorangegangenen Abschnitte ein erstes Verständnis aufgebaut. Dezentrale Versionsverwaltungen arbeiten gar nicht so verschieden dazu. Insbesondere fügen sie jedoch optional eine weitere Hierarchieebene ein. Schauen wir uns das im Folgenden genauer an.

Während bei zentralen Versionsverwaltungen jeweils Zwischenstände (Arbeitskopien) auf die Rechner der jeweiligen Entwickler übertragen werden, wird bei dezentralen Versionsverwaltungen lokal auf dem Rechner des Entwicklers im Arbeitsverzeichnis das gesamte Repository gespeichert.[5] Innerhalb des ***Arbeitsverzeichnisses*** existiert demnach ein vollständiger Stand des Projekts mitsamt seiner Historie. Mit diesem ***lokalen Repository*** kann man nahezu wie mit CVS oder SVN gewohnt arbeiten, allerdings mit dem großen Vorteil, dass alle Aktionen nicht über das Netzwerk, sondern lokal im Dateisystem und damit extrem performant ausgeführt werden können. ***Git und Mercurial ermöglichen also das Arbeiten mit lokalen Repositories unabhängig von einem zentralen Repository und Netzwerkzugang***.[6] Zunächst einmal erfolgt die Versionskontrolle dezentral und jedes Repository existiert für sich eigenständig.

Es besteht aber die Möglichkeit, ein Repository im Netz zu veröffentlichen und so auch mit anderen teilen zu können. Dadurch gibt es bei den Vertretern von DVCS im Unterschied zu den zentralen Versionsverwaltungen konzeptionell noch die Ebene über dem lokalen Repository, nämlich alle möglichen Repositories im Netz (***Remote Repository***). Für die Zusammenarbeit an einem größeren Projekt empfiehlt es sich, neben vielen verteilten Repositories auch ein ausgezeichnetes Projekt-Repository auf einem zentralen Server zu nutzen, ähnlich zu dem zentralen Server bei CVS/SVN. Dadurch gestaltet sich der Umstieg von CVS/SVN leichter, da man sich die lokalen Repositories als praktische lokale Zwischenspeicher vorstellen kann: Ein Feature, das man sich häufig für CVS/SVN gewünscht hat – vor allem dann, wenn die Zugriffe auf das zentrale Repository langsam oder aufgrund von Netzwerkproblemen nicht bzw. nur eingeschränkt möglich waren.

Tipp: Sicherungskopien

Weil jeder Benutzer ein vollständiges Repository auf seinem Rechner besitzt, ist es ganz einfach durch Kopie des Arbeitsverzeichnisses möglich, ein Repository neu zu erzeugen. Somit können auf einfache Art und Weise Sicherungskopien erstellt werden. Selbst bei Datenträgerdefekten auf einem Rechner kann man somit einen (nahezu) vollständigen Stand einfach durch Kopie eines anderen Repository wiederherstellen.

[5] Mercurial und Git legen dazu versteckte Ordner und Dateien an.

[6] Insbesondere muss kein zentraler Server mit Repository existieren – für die Arbeit in (verteilten) Teams ist dies aber hilfreich.

Begriffe und Arbeitsablauf im Überblick

Die Ideen und Abläufe sind bei der Arbeit mit einem lokalen Repository recht ähnlich zu der von zentralen VCS gewohnten Arbeitsweise. In der Abbildung sehen wir einige Arbeitsschritte, die einen ersten Eindruck von der Arbeit mit einem DVCS vermitteln.

Abbildung 2-5 *Repository-Workflow für SVN (links) und DVCS (rechts)*

Für CVS und SVN haben wir bereits den typischen Arbeitsablauf und die entsprechenden Begriffe wie Check-out und Commit kennengelernt. Nachfolgend betrachten wir ein paar typische und grundlegende Arbeitsschritte beim Einsatz von DVCS, die aufgrund ihrer abweichenden Arbeitsweise auch eine leicht andere Begriffswelt besitzen.

Die folgenden Ausführungen zeigen exemplarisch immer die Befehle sowohl für Mercurial als auch für Git. Das Kommando für Mercurial ist `hg`[7] und für Git `git`.

Repository erstellen Sollen die Dateien in einem Verzeichnis versioniert werden, so kann man sehr einfach dort ein neues Repository erstellen: Man wechselt in das gewünschte Verzeichnis und gibt dafür Folgendes ein:

```
git init     // für Git
// ---------------------------------------------------------------
hg init      // für Mercurial
```

Das ist alles, um ein neues Repository anzulegen. Nun können Sie die zu versionierenden Dateien dem Repository hinzufügen. Bevor ich darauf eingehe, beschreibe ich den Fall, dass das Repository als Kopie eines anderen Repository erstellt wird.

[7] Mercurial bedeutet Quecksilber und hg ist das chemische Kürzel dafür.

Repository clonen Zum Erstellen einer Kopie eines Repository nutzt man den `clone`-Befehl, der folgende Syntax besitzt – als Abkürzung verwende ich nachfolgend mitunter `git/hg`, was ausdrückt, dass die Syntax identisch ist:

```
git/hg clone /pfad/zum/repository ziel                // lokales Repository
git/hg clone benutzername@host:/pfad/zum/repository   // Remote Repository
```

Nachfolgend zeige ich ein Beispiel, wie man auf lokale Repositories zugreifen kann, und deute für Mercurial den Zugriff auf ein Remote Repository unter Nutzung von HTTP, hier vereinfachend des eigenen Rechners (`localhost`), an:

```
git clone file:///C:/Users/Micha/Desktop/DVCS DVCS-GIT
// ---------------------------------------------------------------
hg clone file:///C:/Users/Micha/Desktop/DVCS DVCS-CLONE
hg clone http:/localhost:8000/DVCS DVCS-CLONE-BY-HTTP
```

Das erstmalige Initialisieren eines lokalen Repository kann durch eine Kopie eines anderen lokalen oder Remote Repository erfolgen. Dabei wird eine vollständige Kopie erzeugt, d. h. inklusive sämtlicher Branches und der gesamten Versionshistorie. Zudem erfolgt ein Check-out des Hauptasts, den man für Git ***master*** bzw. für Mercurial ***default*** nennt, statt ***trunk*** bei SVN.

Check-out Nach dem Klonen eines Repository befindet man sich zunächst auf dem *master* bzw. *default* und dem aktuellen Stand (HEAD). Um andere Versionsstände (eventuell aus anderen Branches) bearbeiten zu können, dazu dient ein `checkout`, der den gewünschten Versionsstand einer speziellen Revision oder eines Branches in das Arbeitsverzeichnis überträgt. Wollten wir die ältere Revision 123 auschecken, so schreibt man Folgendes, wobei für `git` ein frei erfundener Hex-Key dargestellt ist:

```
git checkout 22a16bb5cdd8a8d72bda4276b720e0a86a2bad72
// ---------------------------------------------------------------
hg checkout -r 123
```

Um wieder auf die aktuellste Revision HEAD zurückzuwechseln, schreibt man einfach:

```
git checkout master
// ---------------------------------------------------------------
hg checkout default
```

Änderungen und Staging Area Wenn Sie Änderungen an den lokalen Dateien vornehmen, werden diese nicht direkt ins lokale Repository übertragen. Git und Mercurial bieten eine weitere Zwischenebene: die sogenannte ***Staging Area***. Alle veränderten Dateien, die (später) ins Repository committet werden sollen, müssen zunächst der Staging Area hinzugefügt werden. Dazu dient der `add`-Befehl:

```
git/hg add <dateiname>
git/hg add *
```

Dies kann man beliebig für verschiedene Dateien wiederholen oder zur Vereinfachung die Wildcard `*` nutzen. Falls man eine oder mehrere Dateien einmal versehentlich der Staging Area hinzugefügt hat, so kann man diese bei Bedarf wieder daraus entfernen. Meistens wird man sich zunächst einen Überblick verschaffen wollen. Dazu dient der `status`-Befehl:[8]

```
git/hg status
```

Bei der Ausgabe symbolisiert ein `?`, dass die Datei nicht unter Versionskontrolle steht. Das `A` (Added) bedeutet, dass die Datei in der Staging Area hinzugefügt ist:

```
A ToBeRemovedFromStaging.txt
? ToBeAdded.txt
```

Anhand der Konsolenausgabe kann man diejenigen Dateien ermitteln, die man wieder entfernen möchte, etwa die Datei `ToBeRemovedFromStaging.txt`. Der dazu benötigte `remove`-Befehl besitzt folgende Syntax:

```
git/hg remove <dateiname>
git/hg remove *
```

Änderungen ins lokale Repository übertragen Durch Aufruf des `commit`-Befehls werden Änderungen aus der Staging Area ins lokale Repository übertragen:

```
git/hg commit -m "Commit-Nachricht"
```

Der Parameter `-m` erlaubt es, beim Commit eine (möglichst) aussagekräftige Nachricht zu übergeben. Die Änderungen sind dann im lokalen Repository historisiert. Bei der Arbeit im Team wird man diese jedoch immer mal wieder mit dem Projekt-Repository abgleichen wollen. Dazu dient der später beschriebene `push`-Befehl.

Branching Bekanntermaßen lassen sich durch Einsatz von Branches unabhängige Entwicklungen separat und isoliert voneinander mit unterschiedlichen Historien entwickeln. Wenn ein Repository neu erstellt wird, so existiert dort standardmäßig immer der Branch *master* (Git) bzw. *default* (Mercurial). Um Merge-Probleme möglichst zu vermeiden, war es bei CVS/SVN nicht ungewöhnlich, auf dem dazu korrespondierenden *trunk* zu entwickeln. Ein analoges Vorgehen ist für Git und Mercurial zwar auch möglich, aufgrund ihrer deutlich besseren Unterstützung von Branching und Merging bietet sich aber ein anderes Vorgehen an: Entwicklungen sollten auf eigenständigen Branches durchgeführt werden, wobei hier – wie schon für CVS/SVN erwähnt – eine

[8] Bitte beachten Sie, dass dieser Befehl auch alle Unterverzeichnisse durchsucht, wodurch die Liste recht umfangreich werden kann. Möchte man die Treffermenge etwa auf PDF-Dateien begrenzen, so kann man Folgendes eingeben: `git/hg status *.pdf`. Es gibt weitere Möglichkeiten zur Einschränkung. Ergänzende Hinweise gibt `git/hg status --help`.

Beschränkung auf eine überschaubare Anzahl an Branches aus purem Selbstschutz und zur besseren Übersicht eingehalten werden sollte.

Möchte man eine Funktionalität auf einem Branch entwickeln, so führt man Folgendes zu dessen Erstellung aus:

```
git branch mybranch        // Branch erstellen
git checkout mybranch      // Stand mybranch in Working Copy holen
git checkout -b mybranch   // Kurzform
// ------------------------------------------------------------
hg branch mybranch
```

Nun kann man Änderungen an den Dateien der Arbeitskopie vornehmen, die den Branch repräsentiert.

Merge Irgendwann sind die Arbeiten an dem besonderen Feature auf dem Branch *mybranch* erfolgreich fertiggestellt und sollen wieder mit dem *master* bzw. *default* abgeglichen werden. Gerade im Zusammenführen von Änderungen und Branches liegt eine Stärke von DVCS.

Um einen Merge-Vorgang zu starten, muss man sich in dem Branch befinden, in den die Änderungen eingespielt werden sollen. Da wir bislang auf dem Branch `mybranch` gearbeitet haben, müssen wir mit dem Arbeitsverzeichnis zunächst wieder auf den *master* bzw. *default* wechseln. Dazu existieren folgende Kommandos:

```
git checkout master
// ------------------------------------------------------------
hg checkout default
```

Um dann einen Abgleich zwischen *mybranch* und *master* bzw. *default* vorzunehmen, gibt man folgendes Kommando ein:

```
git merge mybranch   // Merge vom Branch mybranch
// ------------------------------------------------------------
hg merge mybranch    // Merge vom Branch mybranch
hg commit -m "Merged Branch mybranch"
```

Im Vergleich zu zentralen Versionsverwaltungen benötigen obige Aktionen wenig Zeit, da keine aufwendigen Zugriffe über das Netzwerk erfolgen müssen. Darüber hinaus kann in der Regel deutlich mehr an konkurrierenden Änderungen sinnvoll miteinander kombiniert werden, ohne dass dies Eingriffe des Entwicklers erfordert. Beim Mergen können natürlich dennoch ab und zu Konflikte auftreten, die manuell gelöst werden müssen.

Push und Pull Nachdem nun die Arbeiten abgeschlossen sind oder einen sinnvollen Zwischenstand erreicht haben, können und sollten diese mit dem Projekt-Repository abgeglichen werden.[9] Dazu dient der `push`-Befehl. Diesen kann man jedoch nicht ausführen, wenn bereits Änderungen vorliegen.

Das ist bei der Arbeit in Teams aber eher die Regel: Andere Teammitglieder haben oftmals ebenfalls bereits Änderungen durchgeführt und diese schon an das Projekt-Repository übertragen. Um das eigene, lokale Repository mit demjenigen auf dem Projektserver abzugleichen, dient der `pull`-Befehl.

Wie bei einem Update in SVN können bei einem `pull` mehrere Änderungen innerhalb einer Datei existieren und zu behandeln sein. Oft lassen sich diese automatisch mergen. Bei Widersprüchen müssen diese allerdings von Hand aufgelöst werden. Danach ist dann schließlich ein `push` möglich und erlaubt die Veröffentlichung der eigenen Änderungen.

2.3.3 VCS und DVCS im Vergleich

In diesem Abschnitt stelle ich nochmals zusammenfassend die Eigenschaften zentraler Versionsverwaltungen, insbesondere deren Nachteile, und die Vorteile dezentraler Versionsverwaltungen gegenüber und schließe mit einem Fazit.

Nachteile zentraler Versionsverwaltungen

Zentrale Versionsverwaltungen erleichtern zwar unbestritten die Verwaltung der Historie von Dateien enorm, jedoch besitzen sie auch ganz entscheidende Nachteile:

- **Single Point of Failure** – Durch ihre Ausrichtung auf das zentrale Repository besitzen zentrale Versionsverwaltungen einen ***Single Point of Failure***: Sobald das Repository einmal nicht mehr per Netz erreichbar ist, können nahezu keine Interaktionen mehr ausgeführt werden, die im Zusammenhang mit der Versionsverwaltung stehen. Schnell werden dann ganze Teams einige Stunden ausgebremst.
- **Zeitpunkt des Commits** – Es ist nicht so leicht, den richtigen Zeitpunkt für einen Commit zu finden: Mitunter ist es sogar recht schwierig zu entscheiden, wann und vor allem auch welche Änderungen eingecheckt und welche Teile besser noch nicht im Repository veröffentlicht werden sollten.

Wahl des richtigen Zeitpunkts zum Commit bei VCS Intuitiv scheint klar, dass man mit einem Commit so lange warten sollte, bis die gemachten Änderungen einen sinnvollen Zwischenstand erreicht haben, und dass auch nur lauffähiger, möglichst gut getesteter Sourcecode ins Repository integriert werden sollte. Das Ganze hat

[9] Damit ein Abgleich mit Remote Repositories erfolgen kann, müssen diese entsprechend konfiguriert werden. Ich setze voraus, dass dies initial geschehen ist.

jedoch folgenden Haken: Wenn man an Erweiterungen arbeitet, so erfüllt der neu erstellte Sourcecode manchmal die genannten Anforderungen (noch) nicht. Dafür gibt es unterschiedliche Gründe, etwa weil nicht alles zu 100 % durchdacht ist oder einige Anforderungen unbekannt oder unvollständig sind oder einfach falsch verstanden wurden. Für bestehenden Sourcecode gilt Ähnliches. Möglicherweise sind Erweiterungen auch nur halbherzig umgesetzt und zudem beispielsweise aufgrund vieler Abhängigkeiten kaum testbar. Wird derartiger Sourcecode eingecheckt, so ist er danach auch für alle Teammitglieder sichtbar. Das wiederum hat folgende Konsequenzen:

- Wenn man diesen Stand des Sourcecodes eincheckt, bevor er wirklich fertig ist, verärgert man durch die Halbfertigkeit eventuell andere oder sorgt dafür, dass keine korrekte Programmversion erstellt werden kann.
- Zögert man stattdessen das Einchecken eine Weile hinaus, so besitzt man für einige Dateien keine Versionshistorie, da lediglich lokal gearbeitet wird. Beim Commit entstehen dann oftmals große und schwer nachvollziehbare Änderungen.
- Wenn man als Ausweg auf einem Branch arbeitet, bis sich die Änderungen stabilisiert haben, so hilft dies. Es gibt aber trotzdem mitunter weitere Probleme beim späteren Abgleich der Branches. Deswegen empfiehlt es sich, den eigenen Branch mit dem aktuellen Stand abzugleichen, damit man die Änderungen von Kollegen schon im eigenen Branch hat, um so die Reintegration des eigenen Branches in den Hauptstand zu erleichtern.

Manchmal gibt es Situationen, in denen einige Entwickler ihren Sourcecode einige Tage oder gar Wochen nicht ins Repository integrieren oder aber längere Zeit auf einem Branch arbeiten, einfach daher, weil es nicht leicht ist, den Zeitpunkt zu bestimmen, wann Sourcecode »reif« genug für das Repository ist. Je länger der Integrationsvorgang allerdings verzögert wird, desto höher ist die Wahrscheinlichkeit, dass auch Teamkollegen in der Zwischenzeit an denselben Dateien arbeiten und Konflikte entstehen. Dadurch gestaltet sich das Zusammenführen der Änderungen immer schwieriger.

Während also – wie auch schon erwähnt – viele Branches in zentralen VCS eher Probleme bereiten, als diese zu lösen, adressieren die dezentralen Versionsverwaltungen diese und andere Schwierigkeiten.

Vorteile dezentraler Versionsverwaltungen

Mit DVCS wird dieses Szenario deutlich entschärft, weil jeder Entwickler lokal sein eigenes Repository besitzt. Dort kann er Änderungen ausführen, diese committen, sie zurückrollen usw., alles ohne Einfluss auf die Teamkollegen. Wenn dann eine gute Qualität vorliegt, können die Modifikationen in ein (zentrales) Remote Repository überspielt werden. Indikatoren für den richtigen Moment sind, dass ein Entwickler mit seinen Änderungen zufrieden ist, die Coding Conventions (vgl. Kapitel 19) eingehalten wurden, die Tests erfolgreich laufen und bestenfalls sogar ein Codereview (Begutachtung des Sourcecodes durch verschiedene Entwickler, vgl. Kapitel 21) stattgefunden hat.

Die beschriebene Arbeitsweise für DVCS entspricht eigentlich genau derjenigen, die man sich als SVN-Benutzer schon immer gewünscht hat, nämlich immer dann, wenn a) keine Netzwerkverbindung bestand und b) man umfangreichere Änderungen durchführen musste und deren Zwischenstände im Repository sichern, aber nicht allen direkt zugänglich machen wollte. Als Lösung hat man dann in SVN mit sogenannten Feature-Branches gearbeitet. Das läuft so lange gut, wie man nur auf den Branches entwickelt. Allerdings wird das schnell kompliziert, wenn man die Branches wieder zusammenführen muss. Weil DVCS mehr Informationen zu den Dateien speichern, fällt dort das Arbeiten mit Branches und vor allem das Vereinigen von Branches um einiges leichter als mit SVN.

Fazit

Verteilte Versionsverwaltungen erleichtern die Versionierung von Dateien gegenüber zentralen Versionsverwaltungen. Das gilt selbst dann, wenn man lediglich als Einzelperson lokal arbeiten möchte. Einfacher als mit Git oder Mercurial kann man dies kaum tun. In den letzten Jahren hat sich auch die Unterstützung durch grafische Tools deutlich verbessert. Zur Integration der Versionsverwaltung in den Windows-Explorer sind die Tools TortoiseGit und TortoiseHg praktisch. Beide bieten auch eine grafische Oberfläche. Für Mac OS empfiehlt sich etwa SourceTree[10], das beide DVCS unterstützt.

Kaum ein Unternehmen zögert heutzutage noch, ein DVCS zu nutzen. Verständlich, denn für dessen Einsatz sprechen mittlerweile vielfältige gute Gründe. Diese liste ich nachfolgend nochmals auf, um Sie für deren Einsatz zu begeistern und darüber hinaus eine Argumentationshilfe zu bieten, wenn Sie oder Ihre Firma den Wechsel zu einem DVCS noch nicht vollzogen haben.

- **Geschwindigkeit** – Das lokale Repository ist vollständig im Dateisystem gespeichert und die meisten Aktionen und Arbeitsschritte erfordern keine Netzwerkzugriffe wie bei CVS oder SVN. Einzig der Abgleich mit Remote Repositories kann ohne Netz nicht erfolgen.
- **Einfache Installation und Nutzung** – Git und Mercurial sind im Internet frei verfügbar. Nach ihrer Installation (und einer minimalen Konfiguration) sind die beiden DVCS sofort einsatzbereit. Man kann bereits mit der Arbeit beginnen, selbst wenn eventuell noch kein zentrales Repository eingerichtet ist.
- **Offline Arbeiten** – Wünschenswert ist es häufig, auch offline arbeiten zu können. Insbesondere gilt dies etwa unterwegs, wenn nur eine eingeschränkte Verbindung zum Netz besteht. Zentrale Versionsverwaltungen setzen in der Regel eine funktionierende sowie ständige Verbindung zum Versionsverwaltungsserver voraus, um Arbeitsschritte durchführen zu können. DVCS erlauben das lokale Arbeiten und auch ein nachträglicher Abgleich mit einem anderen Repository ist (vielfach) problemlos möglich.

[10] Frei verfügbar unter `https://www.sourcetreeapp.com/`.

- **Einfaches Tagging und Branching** – Tagging und Branching sind zwei wichtige Hilfsmittel beim Verwalten von Versionen. Mit CVS oder SVN ist die Arbeit mit Tags und Branches teilweise aufwendig oder umständlich. Für DVCS ist das Ganze deutlich einfacher.
- **Einfaches Committen** – Commits erfolgen ins lokale Repository und stehen damit nicht im Konflikt mit den Änderungen anderer. Zudem kann man Commits mithilfe der Staging Area vorbereiten. Bei umfangreichen Änderungen ist es dadurch möglich, diese dann in einem Rutsch zu committen.
- **Datensicherheit** – Jeder Clone eines Repository ist eine vollständige Kopie. Falls tatsächlich mal ein Datenträgerdefekt o. Ä. auftreten sollte, kann man einen Stand aus einem anderen Repository wiederherstellen (sofern es einen Clone gibt).

2.4 Einsatz eines Unit-Test-Frameworks

Um eine ausreichende Softwarequalität sicherzustellen, müssen Programme nach verschiedenen Kriterien geprüft werden. Eine mögliche Qualitätssicherungsmaßnahme auf Sourcecode-Ebene sind sogenannte ***Unit Tests***, die hier kurz vorgestellt werden. Ausführlich wird das Thema Unit-Testen in Kapitel 20 besprochen. Ziel beim Unit-Testen ist es, das korrekte Arbeiten einer kleinen Einheit (Unit) – normalerweise einer Java-Klasse – für sich zu untersuchen: Das Zusammenspiel mit anderen Softwarekomponenten wird bei dieser Art von Test gewöhnlich nicht betrachtet. Das erleichtert den Fokus auf einen klar abgegrenzten Bereich der Funktionalität, wodurch sich berechnete Ergebnisse besser mit gewünschten Resultaten vergleichen lassen.

Die erwarteten Ergebnisse und die korrespondierenden Testfälle werden in Form von Java-Klassen realisiert. Tests werden also nicht nur durchgeführt, sondern zunächst programmiert. So versucht man, auch die Entwickler zum Testen zu bewegen, die sich sonst eher davor drücken. Gar keine Tests durchzuführen, stellt allerdings eine unangebrachte Art von Bequemlichkeit dar: Als Entwickler sollen Sie zumindest dafür sorgen, dass für Tester eine umfassende funktionale Prüfung der Applikation möglich ist, ohne dass diese dabei ständig über einfache Programmierfehler stolpern.

2.4.1 Das JUnit-Framework

JUnit ist ein in Java geschriebenes Framework, das beim Erstellen und Automatisieren von Testfällen auf Klassenebene unterstützt.[11] Es ist durch seinen einfachen Aufbau leicht erlernbar und nimmt viel Arbeit beim Schreiben und Verwalten von Testfällen ab: Praktischerweise kümmert sich das Framework bereits um Dinge wie das Zählen und Berichten von Fehlern, sodass nur die Logik für die Testfälle selbst zu implementieren ist. Dabei unterstützt das Framework durch eine Vielzahl an Methoden, mit denen Testbehauptungen aufgestellt und ausgewertet werden können.

[11] Damit lassen auch ohne Weiteres Integrations- und Systemtests schreiben.

Beispiel: Ein erster Unit Test mit JUnit 5

Zum Testen einer Applikationsklasse wird normalerweise eine korrespondierende Testklasse geschrieben. Häufig beginnt man zur Absicherung wichtiger Funktionalität eigener Klassen damit, einige zentrale Methoden durch Tests zu überprüfen. Dies sollte anschließend schrittweise ausgeweitet werden. Dazu werden Testfälle in Form spezieller Testmethoden erstellt, die mit der Annotation[12] `@Test` markiert sein müssen und keinen Rückgabetyp definieren dürfen. Ansonsten werden sie von JUnit nicht als Testfall betrachtet und bei der Testausführung ignoriert.

Betrachten wir ein einfaches erstes Beispiel, das lediglich das Gesagte verdeutlicht, aber noch keine Funktionalität testet, sondern nur ein Grundgerüst für weitere Experimente bildet:

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class FirstTestWithJunit5
{
    @Test
    void test()
    {
        fail("Not yet implemented");
    }
}
```

Im Listing werden die Annotation `@Test` aus dem Package `org.junit.jupiter.api` sowie die Methoden aus der Klasse `org.junit.jupiter.api.Assertions` importiert. Letztere werden statisch importiert, um eine kürzere Notation und bessere Lesbarkeit beim Aufruf der Testmethoden zu erreichen.

Abweichungen zu JUnit 4 Praktischerweise unterstützt JUnit 5 auch die Vorgängerversion JUnit 4. Um das zu ermöglichen, liegen die Definitionen der Klassen, Interfaces und Annotations in unterschiedlichen Packages. Statt im Package `org.junit.jupiter.api` bei JUnit 5 finden sich bei JUnit 4 die Annotation `@Test` im Package `org.junit` und die Klasse `Assert` im Package `org.junit.Assert`:

```
import static org.junit.Assert.*;

import org.junit.Test;

public class FirstTestWithJUnit4
{
    @Test
    public void test()
    {
        fail("Not yet implemented");
    }
}
```

[12] Annotations sind mit dem Zeichen '@' beginnende Markierungen im Sourcecode, die Metainformationen zu einem damit gekennzeichneten Element (Klasse, Methode usw.) beschreiben.

Wenn man noch ein wenig genauer schaut, sieht man, dass Testmethoden in JUnit 4 `public` sein mussten. Zudem dürfen diese Methoden in JUnit 4 keine Parameter besitzen. Beides ist mit JUnit 5 nicht mehr nötig.

Unit Tests mit Eclipse erstellen Praktischerweise lässt sich ein Grundgerüst für einen Unit Test von Eclipse erzeugen. Dazu wählt man im Kontextmenü des jeweiligen Projekts NEW –> JUNIT TEST CASE. Daraufhin erscheint ein Dialog, in dem man verschiedene Einstellungen vornehmen kann. Das ist exemplarisch in Abbildung 2-6 gezeigt.

Abbildung 2-6 *Erstellen eines JUnit-Tests in Eclipse*

Man kann den Namen des Tests festlegen und angeben, auf welcher Version von JUnit er basieren soll – statt einer Versionsnummer besitzt JUnit 5 dort den Namen Jupiter. Dies nutzen wir und verwenden als Namen `FirstTestWithJUnit5`. Zudem sollten wir darauf achten, die Tests in einem speziellen Verzeichnis für Tests abzulegen, also je nach Verzeichnislayout etwa im Verzeichnis `test` oder `src/test/java`. Bestätigen wir den Dialog, so wird die im obigen Listing gezeigte Testklasse erzeugt, in der wir nun Testmethoden ergänzen können. Des Weiteren wird man im Normalfall im Textfeld `Class under test` die zu testende Klasse anwählen. Weil wir hier zur Einführung keinen Test einer Applikationsklasse, sondern nur einen minimalen Test als Ausgangspunkt erstellen, bleibt dieses Eingabefeld leer.

Tipp: Benennung von Testklassen und von Testmethoden

Die Klassen von Unit Tests sollten den Namen der zu testenden Source-Datei enthalten, um diese leicht miteinander zu assoziieren. ***Sinnvoll ist es, `Test` als Postfix für die Testklasse zu verwenden***: Man würde etwa dem Test für die Klasse `MyClass` den Namen `MyClass`**`Test`** geben. Basierend auf dem Klassennamen findet man sehr schnell den entsprechenden Test. Würde man dagegen das Präfix `Test` für die Testklasse nutzen, so lassen sich Tests nur mit viel Aufwand finden: Alle Namen von Testklassen beginnen dann mit dem Präfix `Test` und unterscheiden sich erst nach dem gemeinsamen Präfix.

Benennung von Testmethoden Mit JUnit 3 mussten alle Testmethoden mit dem Präfix `test` beginnen, um vom Framework erkannt und ausgeführt zu werden. Seit JUnit 4 ist das nicht mehr erforderlich, da dessen Architektur auf Annotations umgestellt wurde. Ich verwende mitunter zusätzlich zur Annotation auch weiterhin das Präfix `test`. Das ist nicht notwendig, erleichtert mir aber die Arbeit. So erreiche ich bei umfangreicheren Testklassen eine bessere Trennung zwischen Hilfsmethoden in der Testklasse und solchen, die Testfälle darstellen.

Schreiben und Ausführen von Tests

Nachdem wir nun wissen, wie man einfache Unit Tests erstellt, möchte ich das Ganze ausbauen. Wir lernen dazu u. a. folgende Funktionalitäten aus dem JUnit-Framework zum Ausführen von Tests und zur vorherigen Definition von Testfällen inklusive der Auswertung von Bedingungen in Testfallmethoden kennen:

- **Auswertung von Bedingungen** – Die Klasse `Assertions` stellt eine Menge von Prüfmethoden bereit, mit denen Bedingungen formuliert und dadurch Zusicherungen über den zu testenden Sourcecode geprüft werden können:
  - Durch Aufruf der überladenen Methoden `assertTrue()` und `assertFalse()` lassen sich boolesche Bedingungen prüfen. Erstere Methode geht davon aus, dass eine Bedingung erfüllt ist, und meldet ansonsten einen Fehler. Für `assertFalse()` gilt das Gegenteil.
  - Mit den überladenen Methoden `assertNull()` bzw. `assertNotNull()` können Objektreferenzen auf `null` bzw. ungleich `null` geprüft werden.
  - Die überladene Methode `assertEquals()` ermöglicht es, sowohl zwei Objekte auf inhaltliche Gleichheit (Aufruf von `equals(Object)`) als auch zwei Variablen primitiven Typs auf Gleichheit zu prüfen. Aufgrund möglicher Rundungsungenauigkeiten bei Berechnungen für die Typen `float` und `double` lässt sich bei deren Prüfung eine maximale Abweichung vom erwarteten Wert angegeben.
  - Die überladenen Methoden `assertSame()` bzw. `assertNotSame()` prüfen Objektreferenzen auf Gleichheit bzw. Ungleichheit gemäß `==`.

- Mithilfe von `fail()` kann man einen Testfall bewusst fehlschlagen lassen, beispielsweise um auf eine unerwartete Situation reagieren zu können.
- Mit JUnit 5 lässt sich mithilfe der Methode `assertThrows()` sehr elegant prüfen, ob eine erwartete Exception ausgelöst wird.

Das initial erzeugte Unit-Test-Grundgerüst erweitern wir nun: Folgendes Listing zeigt einige der gerade vorgestellten Methoden im Einsatz, wobei verschiedene Testmethoden in diesem Beispiel bewusst Fehler provozieren:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class JUnit5ExampleTest
{
    @Test
    public void testAssertTrue()
    {
        final List<String> names = new ArrayList<>();
        names.add("Max");
        names.add("Moritz");
        names.clear();
        assertTrue(names.isEmpty());
    }

    @Test
    public void testAssertFalse()
    {
        final List<Integer> primes = List.of(2, 3, 5, 7);
        // Hier wird bewusst ein Fehler provoziert
        assertFalse(primes.contains(7));
    }

    @Test
    public void testAssertNull()
    {
        assertNull(null);
    }

    @Test
    public void testAssertNotNull()
    {
        // Hier wird bewusst ein Fehler provoziert
        assertNotNull(null, "Unexpected null value");
    }

    @Test
    public void testAssertEquals()
    {
        assertEquals("EXPECTED", "expected".toUpperCase());
    }

    @Test
    public void testAssertEqualsWithPrecision()
    {
        assertEquals(2.75, 2.74999, 0.1);
    }
```

```
    @Test
    public void testFailWithExceptionJUnit5()
    {
        assertThrows(java.lang.NumberFormatException.class, () ->
        {
            // Hier wird bewusst ein Fehler provoziert
            final int value = Integer.parseInt("Fehler simulieren!");
        });
    }
}
```

Listing 2.1 *Ausführbar als* **'JUNIT5EXAMPLETEST'**

- **Testausführung** – JUnit ist in gängige IDEs integriert und erlaubt die Ausführung von Tests direkt aus der IDE. Tests kann man entweder über ein Kontextmenü oder über Buttons im GUI ausführen. Dabei kommt es zu einer Ausgabe ähnlich zu der in Abbildung 2-7. Ein roter Balken zeigt Fehler an. Im Idealfall sieht man ein »beruhigendes« Grün, das den erfolgreichen Abschluss aller Testfälle meldet.

Abbildung 2-7 *Testausführung aus dem GUI der IDE*

Gleitkommawerte mit Abweichung testen Bei Berechnungen mit Gleitkommazahlen kann es zu Rundungsfehlern kommen. Gerade im Bereich von Unit Tests ist es daher wünschenswert, eine kleine Abweichung vom erwarteten Ergebnis noch als valide akzeptieren zu können. Im obigen Beispiel haben wir dies schon für fixe Zahlenwerte gesehen. Für Berechnungen ist das mit JUnit wie folgt möglich:

```
@Test
public void testDoubleWithDeviation()
{
    final double EPSILON = 0.0001;

    // Hier nur exemplarisch für eine komplexe Berechnung
    final double result = 10.0 / 3.0 * 100;

    assertEquals(333.3333, result, EPSILON);
}
```

2.4.2 Parametrierte Tests mit JUnit 5

Teilweise muss man eine Vielzahl an Wertebelegungen testen. Würde man für jede eine eigene Testmethode erstellen, so würde dies die Testklasse ziemlich aufblähen und unübersichtlich machen. Um das eleganter zu lösen, gibt es verschiedene Varianten. Alle haben ihre spezifischen Stärken und Schwächen.

Nachfolgend wollen wir Berechnungen für festdefinierte Wertebereiche oder für eine ausgesuchte Menge an Eingaben prüfen. Allerdings ist es bei einer (sehr) großen Anzahl an Werten keine gute Idee, für alle diese eine Prüfung vorzunehmen, da dies die Ausführungsdauer der Unit Tests oftmals deutlich verlängert – ohne jedoch einen (größeren) Mehrwert zu bringen. Insbesondere hier empfiehlt sich der Einsatz von Repräsentanten aus Äquivalenzklassen, wodurch sich die Zahl der Testfälle drastisch reduzieren sollte. Für Details verweise ich auf das dedizierte Kapitel zum Unit-Testen.

Einführung in parametrierte Tests mit JUnit 5

Mit JUnit 5 ist die Definition von parametrierten Tests ziemlich einfach. Beginnen wir mit dem Fall, dass wir lediglich Parameter für unsere Testmethoden vorgeben und das Ergebnis nicht mit übergeben wollen. Das ist immer dann praktisch, wenn nur eine Bedingung zu prüfen ist, etwa ob ein String nicht leer ist oder eine Zahl eine Primzahl darstellt. Beides könnte man für eine kleine Menge an vorgegebenen Eingaben mithilfe der Annotations `@ParameterizedTest` sowie `@ValueSource` wie folgt realisieren:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

// Es werden zur Demonstration ein paar Fehler produziert
public class FirstParameterizedTest
{
    @ParameterizedTest(name = "run {index}: ''{0}'' is not empty")
    @ValueSource(strings = { "Tim", "Tom", "", "Mike" })
    void isNotEmpty(String value)
    {
        assertFalse(value.isEmpty());
    }

    @ParameterizedTest(name = "run {index}: {0} is a prime")
    @ValueSource(ints = { 1, 2, 3, 4, 5, 6, 7 })
    void ensureIsPrime(int value)
    {
        assertTrue(MathUtils.isPrime(value));
    }
}
```

Im Listing sehen wir, dass jeder parametrierte Test mit `@ParameterizedTest` annotiert werden muss. Als Parameter kann man im Attribut `name` einen String mit Platzhaltern verwenden. Dabei bedeuten die Platzhalter Folgendes: `{index}` entspricht dem Index in den Testdaten, und `{0}`, `{1}`, `{2}` usw. referenzieren die Parameter bzw. die entsprechenden Datenelemente – häufiger wird es auch mehrere Eingaben geben, wie wir es im Anschluss sehen werden. Zudem muss der Testgenerator wissen, welche Eingaben zu prüfen sind. Das kann unter anderem per `@ValueSource` spezifiziert werden,

im Beispiel mit den Spezialisierungen für Strings bzw. `int`-Werte. Zudem gibt es vordefinierte Varianten für `long` und `double`.

Schauen wir uns kurz an, wie sich das Ganze auswirkt: Für jeden angegebenen Parameter wird ein eigener Testfall erstellt und ausgeführt. Wie das dann in Eclipse aussieht, zeigt Abbildung 2-8. Bedenken Sie bitte, dass zu Demonstrationszwecken ein paar Testfehler eingebaut wurden.

Abbildung 2-8 *Parametrierte Testfälle des Programms* FIRSTPARAMETERIZEDTEST

Praxisnähere parametrierte Tests

Bei vielen Tests benötigt man in der Praxis eine Menge von Eingaben und Ergebnissen. Dabei hilft die Annotation `@CsvSource`. Hier kann man jeweils in sich geschlossene kommaseparierte Angaben für die gewünschten Kombinationen der jeweiligen Eingaben bzw. Parameter für die Testmethode machen. Sinnvollerweise repräsentiert der erste oder bevorzugt der letzte das erwartete Ergebnis – nutzt man den letzten Parameter, so folgt dies besser der (europäischen) Denkweise von links nach rechts.

Nachfolgend zeige ich eine mögliche Parametrierung für die Addition zweier Werte mitsamt des erwarteten Ergebnisses:

```
@ParameterizedTest(name = "{index}: {0} + {1} = {2}")
@CsvSource({ "1, 1, 2", "2, -2, 0", "3, 4, 7" })
void testAdd(int first, int second, int expected)
{
    int sum = first + second;

    assertEquals(expected, sum);
}
```

Hier erkennt man, dass die textuellen Werte automatisch in die von den Parametern genutzten Typen konvertiert werden. Genialerweise gibt es darüber hinaus vordefinierte

Konvertierungen, etwa für die Typen aus dem Date and Time API, und man kann damit verständliche Tests schreiben. Hier ist dies für die Ermittlung des ersten Tags im Quartal gezeigt:

```
@ParameterizedTest
@CsvSource({ "2014-03-15, 2014-01-01", "2014-06-16, 2014-04-01",
             "2014-09-15, 2014-07-01", "2014-11-15, 2014-10-01"})
void adjustToFirstDayOfQuarter(LocalDate startDate, LocalDate expected)
{
    final Temporal result = new FirstDayOfQuarter().adjustInto(startDate);

    assertEquals(expected, result);
}
```

2.4.3 Vorteile von Unit Tests

Nachdem wir einen ersten, aber durchaus vielschichtigen Einblick in das Unit-Testen gewonnen haben, möchte ich kurz mögliche positive Auswirkungen auf die Entwicklung sowie die Qualität nennen, um Sie für das Erstellen von Unit Tests zu ermuntern.

Regressionstest

Die Integration von Unit-Test-Frameworks in IDEs ermöglicht es, Unit Tests parallel zur eigentlichen Anwendung zu entwickeln und durchzuführen, ohne dafür die Arbeitsumgebung verlassen zu müssen.

Einmal programmierte Unit Tests können beliebig oft wiederholt werden. Diese Form des Testens wird ***Regressionstest*** genannt. Nach Änderungen an der Software lässt sich durch die erneute Ausführung der Unit Tests sicherstellen, dass auch weiterhin die erwarteten Resultate geliefert werden. Das sollte natürlich nicht von Hand geschehen, sondern es bietet sich eine Automatisierung und Integration als ein Schritt im später vorgestellten Build-Prozess an.

Positive Effekte von Unit Tests

Je komplexer Programme werden, desto wahrscheinlicher enthalten diese auch Fehler. Durch die entwicklungsbegleitende Erstellung von Unit Tests kann man bereits viele davon noch während der Implementierung finden und beseitigen. Das vermeidet Ärger und Kosten, weil die Analysen viel einfacher sind, als wenn Fehler erst kurz vor Auslieferungen oder gar erst vom Kunden gefunden werden. Neben der Verärgerung und Behinderung in der täglichen Arbeit des Kunden erfordert die Fehlerbehebung oft aufwendige Analysen und Nachtests, bis die Ursachen aufgedeckt werden können.

Findet und korrigiert man Fehler dagegen noch während der Entwicklung, so spart dies in der Regel einiges an Aufwand, weil die Funktionalität meistens erst vor Kurzem entwickelt wurde und man sich daher normalerweise besser an Details erinnert sowie mehr anderweitige Informationen verfügbar sind. Zudem hilft das nachfolgend vorgestellte Debugging, Programmabläufe im Detail nachzuvollziehen.

2.5 Debugging

Selbst wenn parallel zur Entwicklung gewissenhaft getestet wird und viele Tests existieren, die im besten Fall die Funktionalität des Programms gründlich prüfen, so verbleiben normalerweise immer noch einige Fehler im Programm. Um auch hartnäckige Fehler aufspüren zu können, ist es wünschenswert, die Programmabläufe im Detail, also am besten Schritt für Schritt, während der Ausführung beobachten zu können. Dazu ist der Einsatz eines sogenannten Debuggers (zu deutsch »Entwanzer«[13]) sinnvoll. Im JDK ist ein einfacher, kommandozeilenbasierter Debugger, der `jdb`, integriert. Allerdings ist dessen Einsatz kryptisch und mühselig und daher nur in Ausnahmefällen angebracht, zumal ein Qualitätsmerkmal aktueller IDEs genau darin besteht, leistungsfähige Debugger mit komfortabler, grafischer Benutzeroberfläche mitzuliefern. Dies gilt gleichermaßen für Eclipse, IntelliJ IDEA und NetBeans. Bevor ich konkret auf das Thema Debugging eingehe, möchte ich noch ein paar Details zur Fehlersuche darstellen.

Notwendige Informationen für eine sinnvolle Fehlersuche

Trotz aller Anstrengungen zum Erreichen einer guten Qualität werden irgendwann bis dahin unentdeckte Programmfehler auftreten. Für deren Behebung und die dazu notwendige vorangehende Analyse ist es meistens sehr hilfreich, wenn man möglichst umfangreiche Informationen zum Fehlverhalten besitzt.

Solange Programme einwandfrei funktionieren, wird man sich seltener auf Fehlersuche begeben als bei einem akut vorliegenden Problem. Nichtsdestotrotz sollte man immer auch einen Teil seiner Arbeitszeit in eine kontinuierliche Verbesserung bzw. Prüfung der bestehenden Implementierung stecken, um präventiv möglichen Fehlern vorzubeugen. Dazu kann man beispielsweise Abfragen und Prüfungen in das Programm integrieren, um die Validität von Eingabeparametern öffentlicher Methoden zu überprüfen und sicherzustellen. Zudem empfiehlt sich eine gründliche Prüfung durch die bereits kurz vorgestellten Unit Tests.

Eine möglicherweise notwendige Fehleranalyse wird deutlich erleichtert, wenn die Fehlersituation anhand der Historie der Methodenaufrufe (Stacktrace) eingegrenzt werden kann. Diese Information besitzt man immer dann, wenn eine Exception ausgelöst wurde. Daher empfehle ich, in eigenen Programmen (schwerwiegende) Fehlersituationen durch Exceptions zu kommunizieren. Diese dienen dazu, den Programmablauf in einer geeigneten Behandlung wieder auf Kurs zu bekommen sowie aussagekräftige Fehlermeldungen für den Kunden oder für Entwickler zu generieren.

Hilfreich für eine Problemanalyse ist zudem, Exceptions mit möglichst viel Kontextinformationen anzureichern. Dort sollten etwa Angaben zu Wertebelegungen von Attributen, lokalen Variablen und Übergabeparametern usw. gemacht werden – aber na-

[13] Der merkwürdige Name »Entwanzer« geht historisch auf die großen wohnraumfüllenden ersten Computer zurück, in denen Insekten zu Fehlern in den Schaltungen geführt haben. Eine kurze Anekdote findet man im Buch »Der Pragmatische Programmierer« von Andrew Hunt und David Thomas [34] in Abschnitt 18 zum Thema Fehlersuche.

türlich beschränkt auf den relevanten Zustand! Mithilfe eines Stacktrace kann man die verursachende Stelle vielfach recht gut eingrenzen, und die bereitgestellten Kontextinformationen helfen bei der weiteren Analyse. Auf diese Weise lassen sich dann in der Regel die Ursachen für Fehler leichter finden.

Leider ist die Fehlersuche in der Praxis dann doch häufig noch etwas komplizierter. Nach der Korrektur offensichtlicher oder durch ausgelöste Exceptions auffindbare Programmfehler wird es zunehmend schwieriger, verbliebene Fehler zu entdecken. Beispielsweise äußern sich diese hartnäckigen Fehler dadurch, dass Berechnungen falsche Ergebnisse liefern – jedoch nur unter ganz speziellen Rahmenbedingungen. Alle diese Situationen und Randfälle im Voraus zu bedenken, wird kein Mensch schaffen. Was kann man also tun? Neben Unit Tests ist für solche Fälle der Einsatz von Debugging-Tools sinnvoll. Daher wenden wir uns nun diesem Thema zu.

Meinung: Fehlerbehandlung und Exceptions

Manchmal hört man, dass Exceptions etwas Schlechtes und zu Vermeidendes sind. Aber oftmals ist es zur Fehlerbehebung hilfreich, wenn bei schwerwiegenden Fehlern oder Problemen aussagekräftige Exceptions ausgelöst werden. Allerdings gilt diese derart positiv formulierte Aussage nur während der Entwicklungsphase und des Tests. Nach Auslieferung an Kunden sind Exceptions immer ärgerlich – auch wenn die Fehlersituation mit Stacktrace deutlich besser analysierbar ist als ohne.

2.5.1 Fehlersuche mit einem Debugger

Ein in die IDE integrierter Debugger ist ein komfortables Hilfsmittel, das die Fehlersuche zum Teil enorm vereinfachen kann. Dafür ist es besonders praktisch, dass die Ausführung eines Programms an nahezu beliebiger Stelle angehalten, die Belegung von Variablen angeschaut, überprüft (und sogar verändert) und das Programm anschließend fortgesetzt werden kann. Eine Programmstelle, an der die Ausführung unterbrochen werden soll, nennt man ***Breakpoint***. Einen solchen kann man im Editor der IDE für die gewünschte Programmzeile setzen (in Eclipse erscheint durch einen Doppelklick ein kleiner blauer Punkt), einen bestehenden Breakpoint kann man auf die gleiche Weise wieder entfernen.[14] Abbildung 2-9 zeigt einen Programmausschnitt einer Methode `doActionForAll()` einer Klasse `GameModel` mit vier gesetzten Breakpoints.

Um ein Programm in dieser speziellen Debugging-Ausführungsart ablaufen zu lassen und untersuchen zu können, muss es im Debug-Modus (F11) und nicht wie üblich per Run (CTRL+F11) gestartet werden. Im Debug-Modus wird die Ausführung des Programms an den zuvor im Editor gesetzten Breakpoints unterbrochen. In der folgenden Abbildung 2-10 ist dies für die Methode `doActionForAll()` gezeigt.

Im linken oberen Teil des Bildschirms ist die Aufrufhierarchie dargestellt. Rechts daneben sind zwei Views (als Karteikartenreiter) platziert, die sowohl die aktuellen

[14] Die folgenden Beschreibungen beziehen sich auf Eclipse. Die Abläufe sind aber in ähnlicher Weise auch mit anderen IDEs möglich.

```
GameModel.java

    public void doActionForAll()
    {
        tickCounter++;
        Line breakpoint:GameModel [line: 116] - doActionForAll()
            stars.tick();
        for (final UFO currentUfo : ufos)
        {
            currentUfo.tick();
        }
        for (final UfoBullet currentBullet : ufoBullets)
        {
            currentBullet.tick();
        }

        if (InputState.INSTANCE.moveLeft)
            player.moveDx(-playerSpeed);
        if (InputState.INSTANCE.moveRight)
            player.moveDx(playerSpeed);

        if (InputState.INSTANCE.fire)
```

Breakpoint

Abbildung 2-9 *Programmausschnitt mit vier gesetzten Breakpoints*

Debug

Playfield (3) [Java Application]
ch1_e_bullets.Playfield at localhost:2027
Thread [main] (Suspended (breakpoint at line 116 in GameModel))
GameModel.doActionForAll() line: 116
Playfield.runGameLoop() line: 40
Playfield.main(String[]) line: 238
Thread [AWT-Shutdown] (Running)
Daemon Thread [AWT-Windows] (Running)
Thread [AWT-EventQueue-0] (Running)
C:\Programme\Java\jdk1.7.0\bin\javaw.exe (21.05.2011 13:57:04)

Aufrufhierarchie

Variables | Breakpoints

GameModel [line: 116] - doActionForAll()
GameModel [line: 121] - doActionForAll()
GameModel [line: 125] - doActionForAll()
GameModel [line: 130] - doActionForAll()

Variablen- bzw. Breakpoint-Ansicht

GameModel.java

Sourcecode und Breakpoints

```
public void doActionForAll()
{
    tickCounter++;
    if (tickCounter % 5 == 0)
        stars.tick();

    for (final UFO currentUfo : ufos)
    {
        currentUfo.tick();
    }
    for (final UfoBullet currentBullet : ufoBul
    {
        currentBullet.tick();
    }
```

Outline

Outline-View

ch1_e_bullets
import declarations
GameModel
INSTANCE : GameModel
UFO_DISTANCE_X : int
UFO_DISTANCE_Y : int
ufos : List<UFO>
ufoBullets : List<UfoBullet>
player : Player
playerBullets : List<PlayerBullet>
lives : int
playerSpeed : int
stars : StarsModel
tickCounter : int
lastFireTickCounter : int
GameModel()
createUfos(Rectangle)

Abbildung 2-10 *Beispiel einer Debug-Session*

Belegungen der Variablen als auch eine Übersicht über definierte Breakpoints bieten. Im linken unteren Teil sehen wir den Programmausschnitt, in dem die Zeile markiert ist, in der das Programm unterbrochen wurde. Rechts daneben befindet sich die sogenannte Outline-View, die Informationen zu der hier untersuchten Klasse `GameModel` zeigt. Diese etwas ungewöhnliche Anordnung wurde hier lediglich zu Demonstrationszwecken gewählt, um die einzelnen Views in kompakter Form beschreiben zu können. Die Anordnung der Views lässt sich je nach Bedarf beliebig ändern.

Aktionen beim Debugging

Wenn das Programm gerade angehalten ist (oder genauer: die JVM die Ausführung unterbrochen hat), können wir nun mit dem Debugging bzw. der Fehleranalyse beginnen. Uns stehen unter anderem folgende Optionen zur Verfügung:

- **RESUME (F8)** – Das Programm wird bis zum nächsten Breakpoint ausgeführt, falls es einen solchen im Ausführungspfad gibt. Im Beispiel würden wir dann auf dem Breakpoint der Zeile `currentUfo.tick();` landen.
- **STEP INTO (F5)** – Befinden wir uns in einer Zeile mit einem Methodenaufruf, so wird zur ersten Zeile der Methode – im Beispiel der aufgerufenen Methode `stars.tick()` – gesprungen und dort die Ausführung in der ersten Zeile angehalten.[15] Enthält die Programmzeile keinen Methodenaufruf, dann stoppt der Debugger an der nächsten auszuführenden Zeile der aktuellen Methode oder dem letzten Befehl einer Methode.
- **STEP RETURN (F7)** – Wurde mit STEP INTO in eine Methode gesprungen, so kann man mit dieser Funktion das Debugging einer derart angesprungenen Methode aussetzen, d. h., die folgenden Zeilen werden ausgeführt, und der Debugger läuft bis direkt hinter die vorherige Aufrufstelle.
- **STEP OVER (F6)** – Es wird die aktuelle Programmzeile ausgeführt. Handelt es sich dabei um einen Methodenaufruf, so wird die entsprechende Methode nicht im Debugger angesprungen, sondern vollständig abgearbeitet. Die Programmausführung wird an der nächsten auszuführenden Zeile nach der aktuellen Zeile gestoppt oder an einem anderen Breakpoint, falls es einen im Aufrufpfad gibt.
- **RUN TO LINE (CTRL+R)** – Soll die Ausführung bis zu einer speziellen Zeile fortgesetzt werden, so können wir den Cursor auf die gewünschte Zeile bewegen und dann diese Funktion auswählen.
- **TERMINATE (CTRL+F2)** – Die Ausführung des Programms wird beendet.

Alternativ zu den in der Aufzählung angegebenen Tastaturkürzeln können wir die Aktionen mithilfe korrespondierender Buttons der Toolbar ausführen. In Abbildung 2-11 ist dies für die Aktion STEP INTO gezeigt. Im unteren rechten Teil ist die Darstellung

[15] Wenn Programmzeilen mehrere Methodenaufrufe hintereinander enthalten, etwa `calcSum(x, y) + calcFactorial(y)`, oder als verschachtelten Aufruf, z. B. `calcSum(x, calcFactorial(y))`, werden diese der Reihe nach abgearbeitet.

der Variablenbelegung zum Ausführungszeitpunkt zu sehen. Wir können beobachten, ob sich alle Werte gemäß unseren Erwartungen verändern, und sogar bei Bedarf einzelne Werte während der Programmausführung modifizieren, sodass entweder Fehler provoziert oder kleinere Fehlbelegungen korrigiert werden können.

Abbildung 2-11 *Beispiel einer Debug-Session mit Auswahl von* STEP INTO

2.5.2 Remote Debugging

Die vorangegangenen Abschnitte haben einen Einstieg in die Thematik Debugging gegeben. Bisher haben wir aber nur solche Programme betrachtet, die direkt in unserer IDE laufen. Oftmals ist es jedoch wünschenswert, auch unabhängig von der IDE gestartete Programme – etwa solche, die bereits an die Testabteilung oder sogar an Kunden ausgeliefert wurden – mit einem Debugger zu untersuchen. Wenn ein Programm in einem Testsystem oder beim Kunden im Einsatz ist, kann man Fehlern nicht mehr nachspüren, indem man das Programm direkt aus der IDE im Debug-Modus startet. Das ist schade, da eine Fehlersuche durch eine schrittweise Analyse des Programmablaufs leichter möglich ist. Durch einen Trick wird jedoch praktischerweise das Debugging separat gestarteter Java-Programme mit dem Eclipse-Debugger möglich. Voraussetzung dazu ist lediglich eine TCP/IP-Verbindung zum Zielsystem und eine entsprechende Parametrierung der JVM beim Start auf diesem System. Bedenken Sie bitte, dass das potenziell ein ***Sicherheitsrisiko*** darstellt und dass dies mit ihrer IT abgeklärt werden sollte.

Parametrierung der JVM zum Remote Debugging

Die JVM bietet spezielle Parameter, die es erlauben, das in einer JVM ausgeführte Programm mithilfe eines externen Debuggers zu untersuchen. Dieser wird in der Regel in einer IDE ausgeführt und verbindet sich über ein Netzwerk mit der programmausführenden JVM. Dazu muss für diese JVM unter anderem ein Port für das Remote Debugging festgelegt werden, mit dem sich die IDE später verbinden kann. Zum Start einer programmausführenden JVM im Debug-Modus müssen folgende Parameter übergeben werden, wobei die Werte gegebenenfalls anzupassen sind:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777
```

Als Hilfestellung für möglicherweise erforderliche Anpassungen sind in der nachfolgenden Aufzählung die genutzten Optionen von `-agentlib:jdwp` beschrieben:[16]

- `transport=dt_socket` – Aktiviert die Kommunikation des Debuggers mit der JVM über Sockets. Normalerweise wird die IDE lokal ausgeführt und verbindet sich über ein Netzwerk mit einer programmausführenden JVM.
- `server=y/n` – Bei `y` wird auf das Einklinken eines Debuggers auf dem durch `address` spezifizierten Port gehorcht. Bei der Option `n` wird versucht, ein Debugging-Gegenstück an der unter `address` angegebenen Adresse zu finden.
- `suspend=y/n` – Bestimmt, ob die JVM mit der Ausführung wartet, bis sich ein Debugger mit ihr verbunden hat oder nicht. Die Einstellung `n` ist sinnvoll, wenn man sich in einer Testphase befindet oder nicht direkt beim Applikationsstart ein Remote Debugging durchführen möchte, sondern nur bei Bedarf.
- `address=7777` – Legt den Port fest, über den sich der Debugger und die JVM verbinden. Es ist ratsam, hier einen leicht zu merkenden, immer gleichen Port zu nutzen (abgesehen vielleicht von Security-Bedenken).

Remote Debugging in Eclipse an einem Beispiel

An einem einfachen Beispiel wollen wir nachvollziehen, dass Remote Debugging recht leicht zu bewerkstelligen ist. Hier geht es wirklich nur darum, zu demonstrieren, wie man Remote Debugging ausführt. Es hat ansonsten keinen praktischen Nutzen!

Wir beginnen mit der zu debuggenden Klasse `RemoteDebuggingExample`. In deren simpler `main()`-Methode wird lediglich von 0 bis 49 gezählt, der aktuelle Wert auf der Konsole ausgegeben und danach jeweils 10 Sekunden Pause gemacht:

[16] JDWP steht für Java Debug Wire Protocol. Das ist ein spezielles Protokoll zur Kommunikation zwischen einem Debugger und einer JVM. Eine Übersicht über alle Optionen bekommt man durch die Aufrufe von `java -agentlib:jdwp=help` oder `java -Xrunjdwp:help`.

```
import java.util.concurrent.TimeUnit;

public class RemoteDebuggingExample
{
    public static void main(final String[] args) throws InterruptedException
    {
        for (int i = 0; i < 50; i++)
        {
            print(i);
        }
    }

    public static void print(final int i) throws InterruptedException
    {
        System.out.println("i=" + i);
        TimeUnit.SECONDS.sleep(10);
    }
}
```

Start des Programms mit JVM-Debug-Einstellungen Der Start des Programms erfolgt auf dem Zielsystem (was für unseren einfachen Test identisch mit dem eigenen Rechner sein kann) mit den bekannten JVM-Debug-Parametern:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777
     RemoteDebuggingExample
```

Komfortabler als der gerade gezeigte Start von der Kommandozeile ist es, sich in Eclipse eine Ausführungskonfiguration zu erstellen (vgl. Abbildung 2-12).

Abbildung 2-12 *Beispiel einer Einstellung zum Starten der Applikation*

Nachdem das Programm im Debug-Modus per Kommandozeile oder aus Eclipse heraus gestartet wurde, ermöglichen es uns die jeweiligen Pausen von 10 Sekunden, in Ruhe mithilfe des Eclipse-Debuggers eine Verbindung zu derjenigen JVM aufzubauen, die unser Beispielprogramm im Debug-Modus ausführt. Damit das Andocken des Debuggers an eine JVM möglich wird, müssen wir eine Ausführungskonfiguration »Remote Java Application« in Eclipse anlegen, wie dies im folgenden Abschnitt beschrieben ist.

Einstellungen zum Remote Debugging in Eclipse vornehmen Um ein Programm remote zu debuggen, muss eine Ausführungskonfiguration erstellt werden. Anschließend sind einige Parametrierungen vorzunehmen. In Eclipse wählt man dazu das Menü RUN –> DEBUG CONFIGURATIONS…. Im Konfigurationsdialog müssen die zuvor für die Applikation verwendeten Angaben zu Port und Rechner eingegeben werden. Das ist beispielhaft in Abbildung 2-13 dargestellt.

Abbildung 2-13 *Beispiel einer Einstellung zum Remote Debugging*

Hinweis: Korrekte Sourcecode-Konfiguration

Damit das Debugging sinnvoll durchgeführt werden kann, müssen im Tab SOURCE die Pfade zum Sourcecode des zu debuggenden Programms passend gewählt werden. Das heißt aber auch, dass im Standardfall für das Remote Debugging immer der korrekt passende Sourcecode vorliegen sollte.

Remote Debugging des Programms Nach diesen Vorbereitungen kann wie beim Debugging eines aus Eclipse heraus gestarteten Programms normal gearbeitet werden, d. h., es können im Sourcecode Breakpoints gesetzt und während der Programmausführung Variablen inspiziert werden. In diesem Beispiel lässt sich recht unspektakulär die Variable `i` beobachten. Zudem können wir mit den Kommandos STEP INTO, STEP OVER usw. experimentieren, um zu erkennen, dass Remote Debugging abgesehen von den Konfigurationsaufwänden tatsächlich so leicht ist wie lokales Debugging. Abbildung 2-14 zeigt eine Unterbrechung des obigen Programms nach dem 20. Durchlauf.

Abbildung 2-14 *Beispiel einer Remote Debugging Session*

Fazit

Dieser Abschnitt hat Ihnen einen einführenden Überblick über die Möglichkeiten zur Fehlersuche mithilfe eines Debuggers verschafft. Insbesondere der Eclipse-Debugger bietet noch weitreichendere Optionen – dort sind z. B. Breakpoints möglich, die nur bei Eintreten einer bestimmten Bedingung oder beim Auftreten einer bestimmten Exception aktiv werden, also im Normalfall den Programmablauf nicht unterbrechen.

Um das Werkzeug Debugger in der Praxis gewinnbringend einsetzen zu können, sollten Sie ein wenig damit experimentieren. Dadurch sind Sie in Zukunft besser gerüstet, wenn es einen hartnäckigen Fehler aufzuspüren gilt.

2.6 Deployment von Java-Applikationen

Nachdem wir nun wissen, wie wir unsere Projekte strukturieren und eine Historie der enthaltenen Dateien verwalten sowie auch durch Unit Tests und Debugging funktionierende Stände produzieren können, werfen wir einen kurzen Blick auf die Auslieferung bzw. die Installation, neudeutsch: das ***Deployment***, von Applikationen.

Größere Programme bestehen aus einer Vielzahl an Klassen und anderen Dateien, wie Textressourcen oder Bildern. Zur Auslieferung eines solchen Programms an Kunden wäre es unpraktisch und vor allem auch fehleranfällig, alle diese Dateien einzeln übertragen zu müssen. Deutlich angenehmer und weniger mühsam ist dagegen die Bereitstellung einer Java-Anwendung in Form einer einzigen Datei. Dazu kann man mehrere Dateien zu einem Java ARchive (***JAR***) zusammenfassen. Vorteilhaft ist dies auch für die Übertragung des Programms über das Netzwerk.

Nach dieser kurzen Motivation und Einführung möchte ich einen Blick auf einige Eigenschaften und Vorzüge von JAR-Dateien werfen:

- **Plattformunabhängigkeit** – Das JAR-Format ist plattformunabhängig und somit problemlos zwischen verschiedenen Plattformen austauschbar. Das stellt einen großen Vorteil im Vergleich zu nativen Bibliotheken dar.
- **Komfort** – JARs vereinfachen die Verteilung und Auslieferung von Software, weil nur eine Datei anstatt einer Menge von Dateien in unterschiedlichen Verzeichnissen bereitgestellt wird. Mithilfe spezieller Metainformationen, die in der sogenannten Manifest-Datei abgelegt sind, kann man das Programm auch direkt über die Kommandozeile starten, und je nach Konfiguration im Betriebssystem ist es sogar möglich, ein enthaltenes Hauptprogramm per Doppelklick zu starten.
- **Klassenbibliotheken** – Neben dem Einsatzgebiet, eine lauffähige Java-Applikation in Form eines JARs bereitzustellen, lassen sich auch mehrere Klassen als Klassenbibliothek in Form eines JARs anderen Programmen zur Verfügung stellen. Eine nutzende Applikation muss dann beim Kompilieren und bei der Ausführung die JAR-Datei in ihren `CLASSPATH` aufnehmen.
- **Performance** – Sowohl im Netzwerk als auch im Dateisystem ist das Kopieren bzw. Übertragen einer Vielzahl kleinerer Dateien aufwendiger und langsamer als das Verarbeiten einer größeren Datei. JARs erlauben somit eine optimierte Übertragung über das Netzwerk oder im Dateisystem.
- **Komprimierung** – Ein JAR ist standardmäßig im ZIP-Format komprimiert. Das macht Netzwerkübertragungen effizienter als ohne Komprimierung.
- **Sicherheit** – Durch Signierung mit einem digitalen Zertifikat kann man unerwünschte Manipulationen an JAR-Dateien aufdecken. In solchen Fällen kann man verhindern, das JAR auszuführen (Executable) bzw. zu laden (Bibliothek).

2.6.1 Das JAR-Tool im Kurzüberblick

Im JDK gibt es zum Verarbeiten, Erstellen, Inspizieren, Ausführen, Entpacken usw. von Java-Archiven ein Tool namens `jar`. Der Aufruf des Tools besitzt folgende Struktur:

```
jar [Aktionsoptionen] [Manifest] jarArchiveName input1 [input2]
```

Achten Sie unbedingt darauf, keine Leerzeichen zwischen den Aktionsoptionen anzugeben, da das `jar`-Tool die Angaben ansonsten falsch interpretiert. Nachfolgend liste ich die wichtigsten Aktionsoptionen tabellarisch auf:

Tabelle 2-1 *Kommandozeilenoptionen des* `jar`*-Tools*

Kürzel und Aktion	Erklärung
`c` – create	Es wird aus den angegebenen Quellen (`input1`, `input2` usw.) ein neues JAR erzeugt. Quellen können nicht nur Dateien, sondern auch Verzeichnisse sein.
`t` – table of contents	Gibt den Inhalt des Archivs auf der Konsole aus.
`x` – extract	Extrahiert die angegebenen Dateien aus dem Archiv in das aktuelle Verzeichnis.
`u` – update	Aktualisiert (überschreibt oder erzeugt) die angegebene Datei in einem bestehenden Archiv.
`f` – file	Gibt an, dass der nächste Parameter die Archivdatei ist. Dies wird nahezu immer gewählt. Sonderbarerweise ist die Ausgabe auf die Konsole der Standard.
`v` – verbose	Man erhält eine ausführliche Protokollierung der durchgeführten Aktionen.
`m` – manifest	Erzeugt eine Manifest-Datei mit Metainformationen zu dem JAR – in Form einer Textdatei mit Schlüssel-Wert-Angaben.

Die Aktionen `c`, `t`, `x` und `u` können nicht miteinander kombiniert werden. Die Option `f` muss man dagegen nahezu immer angeben, weil fast nie eine Interaktion mit der Konsole gewünscht ist. Optional kann eine ausführliche Ausgabe mit der Option `v` erzielt werden.

Falls Sie Unterstützung benötigen, bietet ein Aufruf ohne Argumente eine recht aussagekräftige Hilfe. Nach diesem Kurzüberblick schauen wir uns nun einzelne Kommandos etwas genauer an.

Generierung einer JAR-Datei

Zum Erzeugen einer JAR-Datei nutzt man die Option `c` mit folgender Syntax:

```
jar cvf jarArchiveName inputFileOrDir1 inputFileOrDir2 ...
```

Die Optionen `cvf` stehen für create, verbose und file. Die Quellen können Dateien oder Verzeichnisse sein. Die Angabe kann mit exaktem Namen erfolgen oder aber auch Wildcards (*) enthalten. Im Falle von Verzeichnissen wird deren Inhalt rekursiv dem JAR hinzugefügt.

Machen wir es konkret und verpacken ein kleines Spiel. Dieses enthält neben den Klassen auch Bilder und Töne aus den Verzeichnissen `images` und `sounds`:

```
> jar cvf MyGame.jar *.class images sounds
added manifest
adding: Breakout.class(in = 893) (out= 520)(deflated 41%)
adding: MainGame.class(in = 3616) (out= 2028)(deflated 43%)
adding: images/(in = 0) (out= 0)(stored 0%)
adding: images/explosion.png(in = 978) (out= 983)(deflated 0%)
adding: images/bonus.png(in = 839) (out= 844)(deflated 0%)
adding: sounds/(in = 0) (out= 0)(stored 0%)
adding: sounds/explosion.wav(in = 3912) (out= 1983)(deflated 50%)
adding: sounds/bonus.wav(in = 2839) (out= 1413)(deflated 50%)
```

Anhand der Ausgabe sehen wir neben der Protokollierung des erwarteten Hinzufügens von Dateien auch die Ausgabe von »`added manifest`« sowie die Kompressionsrate für die einzelnen Dateien. Basierend auf den unterschiedlichen Werten für `in` und `out` sowie der Angabe von `deflated` wird protokolliert, wie stark die Dateien komprimiert werden – tatsächlich sind die PNG-Bilder bereits von Hause aus komprimiert und werden durch die Kompression potenziell sogar minimal größer. Verbleibt noch das Manifest, das eine Sammlung von Metainformationen zum Archiv enthält. Später dazu mehr.

2.6.2 JAR inspizieren und ändern, Inhalt extrahieren

Man kann das `jar`-Tool auch dazu nutzen, sich den Inhalt einer JAR-Datei anzeigen zu lassen, daran Modifikationen vorzunehmen oder daraus Dateien zu extrahieren. Statt dies über die Kommandozeile zu machen, bietet es sich an, dafür ein gängiges ZIP-Programm zu nutzen, etwa WinZIP, 7-ZIP oder WinRAR für Windows oder The Unarchiver für Mac. In jedem Fall profitiert man davon, dass JAR-Dateien das ZIP-Format einsetzen.

Anzeigen des Inhalts

Zum Inspizieren des Inhalts einer JAR-Datei existiert das folgende Kommando `t`:

```
jar tf jarArchiveName
```

Damit wird der Inhalt des JARs auf der Konsole ausgegeben. Detailliertere Informationen (etwa zu Dateigrößen usw.) erhält man durch die Angabe der Option `v`.

Ändern des Inhalts

Wenn einzelne Dateien nachträglich im JAR geändert werden sollen, so möchte man dazu natürlich nicht das Archiv vollständig neu erzeugen müssen. Der Inhalt einer JAR-Datei lässt sich mithilfe des Kommandos `u` modifizieren:

```
jar uf jarArchiveName input-file(s)
```

Die übergebenen Dateien werden im JAR aktualisiert. Falls diese im JAR noch nicht existierten, werden sie hinzugefügt.

Nehmen wir an, wir hätten eine Änderung an der Datei `ReadMe.txt` vorgenommen und wollten diese ins JAR integrieren. Dazu schreiben wir:

```
jar uf MyGame.jar ReadMe.txt
```

Extrahieren des Inhalts

Zum Extrahieren des Inhalts einer JAR-Datei existiert das Kommando `x`:

```
jar xf jar-file [archived-file(s)]
```

Damit extrahiert man alle angegebenen Dateien aus dem JAR in das aktuelle Verzeichnis. Werden keine Dateien angegeben, so wird das JAR vollständig entpackt.

Als Anwendungsfall könnten wir uns vorstellen, dass man nochmals an der Textdatei `ReadMe.txt` eine Änderung vornehmen möchte. Dazu extrahieren wir die Datei aus dem JAR, ändern anschließend deren Inhalt und fügen sie wieder per Update in das Archiv ein.

```
jar xf MyGame.jar ReadMe.txt
...
jar uf MyGame.jar ReadMe.txt
```

2.6.3 Metainformationen und das Manifest

Wenn Sie bei der Erzeugung der JAR-Datei in Abschnitt 2.6.1 genau hingeschaut haben, dann wurde dabei auf der Konsole der Text »`added manifest`« ausgegeben. Bei der Generierung eines JARs wird durch das `jar`-Tool automatisch eine sogenannte ***Manifest-Datei*** namens `MANIFEST.MF` im Unterverzeichnis `META-INF` erstellt und dem JAR hinzugefügt. Dieses Manifest ist eine spezielle Datei und dient dazu, Metainformationen zu den Dateien, die im JAR zusammengefasst sind, bereitzustellen.

Zum besseren Verständnis werfen wir nun einen Blick auf die Default-Manifest-Datei, dann werden wir Klassen aus einem JAR starten und dazu eine Main-Klasse als Einstiegspunkt angeben.

Metainformationen in der Default-Manifest-Datei

Wir nutzen unser bisheriges Wissen und entpacken die Datei mit folgendem Aufruf:

```
jar xvf MyGame.jar META-INF/MANIFEST.MF
```

Beachten Sie unbedingt, dass das `jar`-Tool penibel zwischen Groß- und Kleinschreibung unterscheidet, obwohl beispielsweise das Windows-Betriebssystem dies nicht tut. Durch das erfolgreiche Entpacken wird das Unterverzeichnis `META-INF` und darin die Datei `MANIFEST.MF` erzeugt. Öffnen wir die gerade entpackte Datei, so enthält sie folgenden Inhalt mit (recht spärlichen) Metainformationen – wobei die nach `Created-By` angegebene Java-Version gegebenenfalls bei Ihnen abweicht:

```
Manifest-Version: 1.0
Created-By: 1.8.0_121 (Oracle Corporation)
```

Wie man leicht sieht, handelt es sich hier um eine einfache zeilenorientierte Textdatei mit der Angabe von Schlüssel-Wert-Paaren, die durch »`:`« getrennt sind.

Klassen aus JAR starten

Wie schon zuvor angedeutet, können wir eine Java-Anwendung als JAR bereitstellen. Wenn wir diese nun starten wollen, so kann man folgenden Aufruf nutzen:

```
java -cp jarArchiveName mypackage.MainClassName
```

Dabei dient das JAR als Klassenbibliothek und wird explizit im `CLASSPATH` aufgeführt. Zudem wird die zu startende Klasse angegeben. Alternativ ist auch folgender Aufruf erlaubt, um eine Klasse aus einem JAR zu starten:

```
java -jar jarArchiveName
```

Probieren wir es für unser Beispiel `MyGame.jar` folgendermaßen aus:

```
java -jar MyGame.jar
```

Statt des Programmstarts kommt es jedoch zu einem Problem und folgender Fehlermeldung »`kein Hauptmanifestattribut in MyGame.jar`«. Durch kurzes Nachdenken wird klar: Woher soll die JVM wissen, welche `main()`-Methode welcher Klasse gestartet werden soll? Diese Information kann in einem JAR über die Manifest-Datei beschrieben werden. Dazu lassen sich neben den Standardangaben, die beim Vorgang der Archivierung automatisch vom `jar`-Tool erzeugt werden, weitere Informationen, etwa zur `main`-Klasse und benötigten Bibliotheken, im Manifest hinterlegen.

Informationen im Manifest bereitstellen Wir haben gesehen, dass die Manifest-Datei `MANIFEST.MF` automatisch vom `jar`-Tool erzeugt wird. Um dort weitere, über die Standardangaben hinausgehende Informationen bereitzustellen, kann man eine Textdatei beliebigen Namens, meistens `manifest.txt` genannt, nutzen. Darin lassen sich Schlüssel-Wert-Paare angeben, etwa wie folgt:

```
Main-Class: de.javaprofi.helloworld.App
```

Diese Angaben werden vom `jar`-Tool bei der Erstellung der Manifest-Datei berücksichtigt, wenn man die Option `m` angibt:

```
jar cvfm MyGame.jar manifest.txt classes/
```

Das so erzeugte JAR enthält nun die zuvor fehlenden Informationen zur `main`-Klasse. Rufen wir also Folgendes auf:

```
java -jar MyGame.jar
```

Doch das Programm startet immer noch nicht! Stattdessen kommt es zu folgendem Fehler: »`Hauptklasse de.javaprofi.helloworld.App konnte nicht gefunden oder geladen werden`«.

Das ist zunächst etwas merkwürdig. Wenn wir jedoch das JAR mit dem Kommando `jar tf Mygame.jar` inspizieren, so stellen wir fest, dass die `.class`-Dateien aus dem Unterordner `classes` exakt so (also auch inklusive des Verzeichnisnamens) in das Archiv übernommen wurden:

```
Manifest wurde hinzugefügt
classes/ wird hinzugefügt(ein = 0) (aus = 0)(0 % gespeichert)
classes/de/ wird hinzugefügt(ein = 0) (aus = 0)(0 % gespeichert)
classes/de/javaprofi/ wird hinzugefügt(ein = 0) (aus = 0)(0 % gespeichert)
classes/de/javaprofi/helloworld/ wird hinzugefügt(ein = 0) (aus = 0)(0 %
      gespeichert)
classes/de/javaprofi/helloworld/App.class wird hinzugefügt(ein = 561) (aus =
      349)(37 % verkleinert)
```

Dadurch stimmen die Package-Angaben im JAR und in den Java-Klassen nicht überein. Um dies zu verhindern, muss man dem `jar`-Tool beim Erzeugen eines JARs mitteilen, dass das Verzeichnis, in dem sich die Klassen befinden, nicht Bestandteil des archivierten Pfads werden soll. Dazu dient die Option `-C` zur Festlegung des Pfads zu den Klassen mit einer etwas ungewöhnlichen Angabe wie folgt:

```
jar cvfm MyGame.jar manifest.txt -C classes .
```

Nach dieser Modifikation lässt sich das Programm dann aus dem JAR starten.

2.6.4 Inspizieren einer JAR-Datei

Nachdem wir einiges zu JAR-Dateien und deren Manifest kennengelernt haben, wollen wir uns nun ansehen, wie wir diese Informationen mithilfe eines Java-Programms aus-

lesen können. Neben der Klasse `java.io.File`, die hier die JAR-Datei im Dateisystem repräsentiert, kommen noch die Klassen `JarFile`, `Manifest` und `Attributes` aus dem Package `java.util.jar` zum Einsatz, die die programmatische Verarbeitung von Informationen aus JAR-Dateien erleichtern.

```
public static void main(final String[] args) throws IOException
{
   final String jarFileName = "src/main/resources/ch02_profbuild/" +
                              "gradle-example.jar";
   final File file = new File(jarFileName);
   System.out.println(file.getAbsolutePath());
   System.out.println("jarFileName='" + jarFileName + "'");

   // Informationsobjekte ermitteln
   try (final JarFile jarfile = new JarFile(jarFileName))
   {
      final Manifest manifest = jarfile.getManifest();
      final Attributes attributes = manifest.getMainAttributes();

      // Informationen auslesen
      final String mainClassName = attributes.getValue(Attributes.Name.
            MAIN_CLASS);
      System.out.println("mainClassName='" + mainClassName + "'");

      final String createdBy = attributes.getValue("Created-By");
      System.out.println("createdBy='" + createdBy + "'");

      final String manifestVersion =
                  attributes.getValue(Attributes.Name.MANIFEST_VERSION);
      System.out.println("manifestVersion='" + manifestVersion + "'");
   }
}
```

Listing 2.2 *Ausführbar als* '**INSPECTINGJAREXAMPLE**'

Im Listing sehen wir verschiedene Methoden, wie etwa `getManifest()` und `getMainAttributes()`, zum Zugriff auf die Attribute des Manifests. Zum Auslesen der Werte dient die Methode `getValue()`, der ein Schlüsselname übergeben wird. Verschiedene gebräuchliche Namen sind in `Attributes.Name` definiert. Starten wir das Programm INSPECTINGJAREXAMPLE, so kommt es zu folgenden Ausgaben:

```
/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-quelltext/gradle-example
      .jar
jarFileName='gradle-example.jar'
mainClassName='ch02_profbuild.ResourceAccessExample'
createdBy='null'
manifestVersion='1.0'
```

Auf Daten bzw. Ressourcen aus JAR-Dateien zugreifen

Wir wissen nun, wie man Klassen und andere Ressourcen zu JARs zusammenfasst, Informationen dazu in einer Manifest-Datei hinterlegt und diese programmatisch wieder ausliest. Wenn wir eine Applikation in Form eines JARs generiert haben und nun deren

`main`-Klasse starten, so stellt sich die Frage, wie man innerhalb der Applikation auf Daten zugreifen kann, die im JAR hinterlegt sind.

Nehmen wir an, die Spieleapplikation des Beispiels wäre als JAR verpackt und es soll auf die Bilder und Töne in den Unterverzeichnissen `images` bzw. `sounds` zugegriffen werden. Da diese Dateien aber innerhalb des JARs und nicht als Dateien im Dateisystem existieren, ist ein Zugriff nicht mithilfe einer `File`-Instanz möglich: Denn diese erlaubt keinen Zugriff auf die internen Strukturen des JARs. Für diesen Anwendungsfall gibt es in der Klasse `java.lang.ClassLoader` zwei spezielle Methoden, nämlich `getResource()` und `getResourceAsStream()`, wobei erstere ein Objekt vom Typ `java.net.URL` liefert. Die Methode `getResourceAsStream()` gibt ein `java.io.InputStream`-Objekt zurück. Als Alternative existieren beide Methoden auch in der Klasse `java.lang.Class`. Neben einem Zugriff auf Ressourcen innerhalb eines JARs kann man mit den Methoden auch auf Ressourcen zugreifen, die innerhalb des Dateisystems liegen.

```
public static void main(final String[] args) throws IOException
{
   final String filename = "images/Bad-Question-Message.png";

   // Variante mit ClassLoader (1) und Class (2)
   final URL imageUrl1 = ResourceAccessExample.class.getClassLoader().
                                                 getResource(filename); // 1
   final URL imageUrl2 = ResourceAccessExample.class.getResource(filename); // 2
   System.out.println("imageUrl1: " + imageUrl1);
   System.out.println("imageUrl2: " + imageUrl2);

   // Variante mit InputStream
   final InputStream is1 = ResourceAccessExample.class.
                           getClassLoader().getResourceAsStream(filename);
   final InputStream is2 = ResourceAccessExample.class.
                           getResourceAsStream(filename);
   // ...
}
```

Listing 2.3 *Ausführbar als* **'RESOURCEACCESSEXAMPLE'**

Der Unterschied zwischen den im Listing gezeigten Varianten besteht darin, dass die Variante 1 basierend auf `ClassLoader` relativ zum Projektverzeichnis zugreift und Variante 2 relativ zu der `.class`-Datei der entsprechenden Klasse.

Um den Zugriff aus einem JAR zu demonstrieren, starten Sie bitte das Programm RESOURCEACCESSEXAMPLE. Dann kommt es in etwa zu folgenden Ausgaben:

```
imageUrl1: jar:file:/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-
      quelltext/gradle-example.jar!/images/Bad-Question-Message.png
imageUrl2: jar:file:/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-
      quelltext/gradle-example.jar!/ch02_profbuild/images/Bad-Question-Message.
      png
is1: sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@6f79caec
is2: sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@5d3411d
```

2.7 Einsatz eines IDE-unabhängigen Build-Prozesses

In Abschnitt 2.3 habe ich motiviert, eine Versionsverwaltung einzusetzen, um gewünschte Versionsstände des Sourcecodes und anderer Dateien jederzeit zuverlässig gemäß einem älteren Stand wiederherstellen zu können. Dieser Abschnitt beschreibt die Vorteile eines von der IDE unabhängigen Build-Prozesses. Schauen wir kurz, warum dies so wichtig ist. Programme bestehen gewöhnlich aus einer Vielzahl an Klassen, die oftmals auch Abhängigkeiten zu externen Bibliotheken besitzen. Innerhalb der IDE lassen sich derartige Abhängigkeiten und zugehörige Pfade verwalten – jedoch können hier lokale Einstellungen größeren Einfluss darauf haben, wie das Programm erstellt wird: Insbesondere variieren Einstellungen erfahrungsgemäß (zumindest ein wenig) von Entwickler-PC zu Entwickler-PC. Dadurch sind die Resultate beim Kompilieren nicht immer 100 % reproduzierbar. Das ist für Auslieferungen oder spezielle Zwischenstände bzw. Versionen des Programms (auch ***Release*** genannt) nicht akzeptabel. Vielmehr sollte der Herstellungsprozess (***Build***) auf eine definierte Art und Weise mit einer eindeutigen Konfiguration und gemäß einem definierten Ablauf erfolgen. Dazu dient ein von der IDE unabhängiger Build-Prozess. Darüber hinaus ist es wünschenswert, sowohl Unit Tests als auch andere Prüfungen als festen Bestandteil des Build-Laufs auszuführen, um ein Mindestmaß an Qualität sicherzustellen. Im Speziellen gilt dies, wenn man den Vorgang des Kompilierens und der Releaseerzeugung automatisieren möchte. Dafür nutzt man in der Regel ein Build-Tool.

Für Java stehen mit ***Ant***, ***Maven*** und ***Gradle*** populäre Tools bereit. Jedes davon besitzt eine unterschiedliche Art, wie es den Entwickler bei der Gestaltung des Build-Vorgangs unterstützt. Während mit Ant jeder (kleine) Arbeitsschritt einzeln beschrieben werden muss, vereinfachen Maven und Gradle den Build-Vorgang durch die Bereitstellung von Automatiken, basierend auf Konventionen und einer fixen Reihenfolge der Abläufe im Build-Prozess. Dadurch verliert man zwar etwas Flexibilität, jedoch wird andererseits auch viel Arbeit abgenommen, da die manuelle, mühsame und oft fehlerträchtige Beschreibung des Build-Laufs entfällt.[17]

Während für die ersten beiden Auflagen dieses Buchs noch Ant als Build-Tool genutzt wurde, kommt seit der dritten Auflage das Build-Tool Gradle zum Einsatz. In dieser 5. Auflage behandle ich Ant nicht mehr, da es in der Praxis nur noch eine unbedeutende Rolle spielt. Allerdings ist es für den Berufsalltag hilfreich, wenn man ein wenig Hintergrundwissen zu Maven besitzt. Deswegen wird in den folgenden Abschnitten Maven kurz beleuchtet, um dann die Vorzüge von Gradle entsprechend schätzen zu können.

[17] Tatsächlich kann man den Build auch mit Gradle feingranular beeinflussen – als normaler Anwender wird man dies eher selten nutzen.

Motivation für ein IDE-unabhängiges Build-System

Stellen Sie sich noch einmal einen dieser hektischen Freitage vor. Es ist wieder für den späten Nachmittag eine Version des Programms, basierend auf einem älteren Stand, auszuliefern. Glücklicherweise haben Sie – spätestens nach der Lektüre des Abschnitts über Versionsverwaltungen – eine solche Versionsverwaltung installiert. Die Tatsache, dass an der Applikation bereits weiterentwickelt wurde, stört Sie nicht, da Sie den benötigten älteren, mit einer Markierung versehenen Auslieferungsstand separat auschecken. Nach ein paar Stunden harter Arbeit sind die geforderten Bugfixes in das Programm integriert. Die Kompilierung in Ihrer Entwicklungsumgebung verläuft ohne Probleme. Auch die Ausführung der Unit Tests zeigt ein beruhigendes Grün. Bis jetzt sieht alles so aus, als ob die Auslieferung bald vorgenommen werden könnte. Um eine Auslieferung zu erzeugen, müssen die Sourcen allerdings mithilfe eines Build-Tools übersetzt werden. Nur wenn ein solcher Build-Vorgang erfolgreich war, erhält man ein neues Release. Eigentlich sollte jetzt nichts mehr schiefgehen, da der IDE-Build und die Unit Tests erfolgreich waren! Doch der Build schlägt fehl. Wie kann das sein?

Eine wahrscheinliche Ursache sind nicht vollständig eingecheckte Stände der eigenen Änderungen. Das stellt man jedoch nicht fest, wenn man das Build-Skript im eigenen Arbeitsverzeichnis ausführt. Für ein Release-Build sollte daher der aktuelle Repository-Stand in ein separates Verzeichnis ausgecheckt werden. In diesem wird dann das Build-Skript gestartet. Treten dabei Probleme auf, so wurden tatsächlich einige Änderungen nicht vollständig ins Repository eingespielt. Ansonsten erhält man eine positive Rückmeldung und man kann eine Markierung setzen.

Ein weiteres mögliches Problem besteht darin, dass von der IDE und dem Build-Skript verschiedene Bibliotheken referenziert werden, etwa wenn eine neue Bibliothek in der IDE bekannt ist, aber noch nicht dem Build-Skript. Zu solchen Problemen kommt es schnell, wenn man die Abhängigkeiten in der IDE manuell verwaltet.[18]

Tipp: Startschwierigkeiten beim Einführen eines Build-Prozesses

Beim Einführen eines Build-Prozesses gibt es zum Teil einige Startschwierigkeiten: Möglicherweise verwendet die IDE einen anderen Java-Compiler (oder eine andere Version davon) als das Build-Skript. Dies kann immer dann passieren, wenn verschiedene JDKs und JREs installiert sind. Teilweise werden durch die IDE auch andere Pfade verwendet und so andere Versionen von Bibliotheken oder Klassen referenziert. Zudem können absolute statt relativer Pfade ein Problem darstellen. ***Deshalb sollten Sie gleich bei Projektstart auf einen automatischen Build-Prozess Wert legen, um nachträglich Probleme zu vermeiden***.

[18] Praktischerweise erlauben es einige Build-Tools, wie das später vorgestellte Gradle, Projektdateien für die IDE zu generieren. Wenn man diese Funktion nutzt, kann man Build-Skript und IDE leichter synchron halten. Gerade bei vielen externen Abhängigkeiten ist das hilfreich.

Grundlage für Nightly Builds bis hin zu Continuous Integration (CI)

Ein von der IDE-unabhängiger Build-Prozess befreit von Besonderheiten und Konfigurationseinstellungen der genutzten IDE und bietet vor allem auch die Möglichkeit, den Build-Vorgang auf einen speziellen Build-Server auszulagern. Auf dem Build-Server wird vor dem Start des Build-Vorgangs zunächst der gewünschte bzw. aktuelle Stand aus der Versionsverwaltung in ein separates Verzeichnis ausgecheckt und dort dann ein Build-Lauf gestartet.

Erfolgt dieser Vorgang jede Nacht, so spricht man von einem ***Nightly Build***. Dadurch erhält man täglich eine Rückmeldung über den momentanen Stand der Entwicklung. Wünschenswert ist es, diesen Prozess mehrmals am Tag auszuführen. Idealerweise wird bei jeder Integration in das Repository nach kurzer Zeit automatisch ein Build-Vorgang angestoßen. Dadurch erhält man umgehend eine Rückmeldung über den Zustand des Systems und ob die letzten Änderungen möglicherweise Fehler enthalten. Ein bekanntes Tool, das eine solche ***kontinuierliche Integration*** (***Continuous Integration***) erlaubt, ist Jenkins[19].

2.7.1 Maven im Überblick

Erfahrungsgemäß besitzen viele Projekte doch recht ähnliche Anforderungen an einen Build-Lauf, nämlich Java-Klassen kompilieren, Tests ausführen und daraus ein Produkt (auch ***Artefakt*** genannt), z. B. in Form eines JARs, erzeugen. Natürlich ist es wünschenswert, diesen Ablauf nicht für jedes Projekt erneut beschreiben zu müssen. Genau dabei hilft Maven[20], indem es einen Standardablauf vorgibt, der für sehr viele Projekte geeignet ist. Das vereinfacht das Build-Management von Softwareprojekten, erfordert es aber, gewisse Konventionen einzuhalten: Maven setzt dazu stark auf ***Convention over Configuration***, d. h., es sind sinnvolle Standards definiert, und man muss nur selten steuernd eingreifen. Das basiert auf drei Eckpfeilern:

1. **Verzeichnislayout** – Es wird ein einheitliches Layout des Projektverzeichnisses erwartet, sodass immer klar ist, wo sich welche Dateien befinden und welche Verzeichnisse beim Build-Lauf angelegt werden.
2. **Abfolge im Build-Lauf** – Der Build-Lauf ist durch eine standardisierte, wohldefinierte Abfolge verschiedener Schritte im Build-Prozess festgelegt.
3. **Dependency Management** – Nahezu immer benötigt ein Projekt auch externe Bibliotheken oder Artefakte. Ein sogenanntes ***Repository*** dient dazu, versionierte Artefakte bereitzustellen, also Artefakte zu speichern und zu verwalten. Mithilfe von Repositories und von dort bereitgestellten Artefakten erleichtert Maven das Dependency Management enorm.

[19] `https://jenkins.io/`

[20] Maven kann unter `http://maven.apache.org/index.html` frei bezogen werden.

Aufbau des Projektverzeichnisses

Maven nutzt die bereits in Abschnitt 2.2 vorgestellte einheitliche Projektstruktur, wodurch direkt klar ist, wo sich die Sourcen der Applikation und von Testklassen sowie zugehörige Ressourcen befinden.

Build-Lauf und Project Object Model (POM)

Die Build-Ausführung ist bei Maven in verschiedene voneinander abhängige Phasen unterteilt. Für Java-Projekte existieren unter anderem folgende:

- `clean` – Führt Aufräumarbeiten aus, löscht temporäre oder im vorherigen Build-Lauf erzeugte Dateien und Verzeichnisse.
- `compile` – Kompiliert die Sourcen (aber nicht die Testklassen).
- `test` – Kompiliert die Testklassen und führt die enthaltenen Testfälle aus.
- `package` – Dient zum Erstellen eines Artefakts, z. B. eines JARs. Dazu müssen zuvor die Sourcen kompiliert und die Tests ausgeführt werden.

Damit Maven weiß, was in jedem der aufgelisteten Schritte zu tun ist, lassen sich gewisse Vorgaben definieren. Dazu dient die Datei `pom.xml`. Dort wird das sogenannte ***Project Object Model*** (***POM***) eines Projekts definiert, das in XML codiert ist und festlegt, wie das zu erstellende Artefakt heißt, welche Versionsnummer es trägt, welche Abhängigkeiten von externen Bibliotheken existieren usw.

Eine Hello-World-Applikation könnte etwa folgende `pom.xml`-Datei besitzen:

```
<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/
        maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>de.javaprofi.helloworld</groupId>
  <artifactId>helloworld</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>helloworld</name>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.1</version>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <dependencies>
     <!-- hier können Abhängigkeiten angegeben werden -->
  </dependencies>
</project>
```

Die Build-Datei ist recht kurz und dabei sehr mächtig, weil durch die obigen Zeilen ein vollständiger Build-Lauf inklusive Packaging ausgedrückt wird. Der Einsatz von XML erschwert es aber, den Nutzinhalt aus den visuell dominierenden Tags zu extrahieren. Darüber hinaus sind einige nicht sofort verständliche Begriffe, wie `groupId`, `artifactId` usw., enthalten, auf die ich nun eingehe.

Wichtige Begrifflichkeiten bei Maven

Um mit Maven produktiv zu arbeiten und die nachfolgenden Beispiele leichter nachvollziehen zu können, müssen wir zunächst ein paar elementare Begriffe kennenlernen.

Basierend auf der Standardverzeichnisstruktur wird durch einen Maven-Build in der Regel genau ein Artefakt erzeugt, etwa ein JAR, ZIP, WAR usw. Um Artefakte unterscheiden und versionieren zu können, werden drei Identifikationen genutzt:

1. `groupId` – Eine Gruppenkennung, ähnlich einer Package-Hierarchie oder eines Namespaces, z. B. `de.javaprofi.helloworld`
2. `artifactId` – Die Kennung eines Artefakts, z. B. `helloworld`
3. `version` – Eine Versionskennung, z. B. `3.8.1`[21]

Mithilfe dieser Informationen wird beim Build-Lauf der Name des Artefakts konstruiert. Darüber hinaus können die Angaben auch dazu genutzt werden, Abhängigkeiten des zu erstellenden Artefakts von anderen Artefakten, etwa den JARs einer externen Bibliothek, zu beschreiben.

Abhängigkeiten und Dependency Management

Wir wollen nun einen Blick darauf werfen, wie man mit Maven Abhängigkeiten des eigenen Projekts von anderen Bibliotheken (Artefakten) beschreiben kann. Dazu wird von Maven eine leistungsfähige Automatik bereitgestellt, die darauf basiert, dass Abhängigkeiten deklarativ beschrieben werden. Dazu hinterlegt man im POM einen `dependency`-Eintrag wie folgt (nachfolgend für eine externe Abhängigkeit zu JUnit in Version 5.6.2):

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Die Angabe von `<scope>test</scope>` erlaubt es, die Art der Abhängigkeit genauer festzulegen. Hier gilt, dass die Abhängigkeit nur in der Phase des Testens benötigt wird.

[21] Maven unterscheidet zwischen Release- und (in der Entwicklung befindlichen) Snapshot-Versionen, wobei Letztere explizit durch das Postfix `-SNAPSHOT` gekennzeichnet werden.

Für das Auflösen und das Bereitstellen der benötigten Bibliotheken ist Maven zuständig und bietet dafür eine leistungsfähige Unterstützung: Neben direkten Abhängigkeiten machen einem bei einer händischen Verwaltung vor allem transitive Abhängigkeiten (also die Abhängigkeiten der direkten Abhängigkeiten) das Leben schwer. Weil jedes Artefakt seine Abhängigkeitsinformationen bei sich trägt, kann Maven transitive Abhängigkeiten automatisch auflösen, sodass diese nicht explizit aufgeführt werden müssen. Um Abhängigkeiten zu visualisieren, kann das Kommando `mvn dependency:tree` nützlich sein.

Abhängigkeiten mit Repositories auflösen Wie eingangs erwähnt, werden in Repositories Artefakte gespeichert und verwaltet. Auch Maven nutzt solch ein Repository. Praktischerweise existiert im Internet das Repository ***Maven Central***, in dem eine Vielzahl von Bibliotheken frei zugänglich sind.

Beim ersten Build-Lauf werden von Maven automatisch alle benötigten Abhängigkeiten (und gegebenenfalls auch benötigte Plugins) aus dem Internet nachgeladen. Um nachfolgende Build-Läufe zu beschleunigen und für folgende Build-Läufe eine Unabhängigkeit vom Internet zu schaffen, werden die heruntergeladenen Artefakte in einem lokalen Repository auf dem eigenen Rechner zwischengespeichert.

Maven am Beispiel

Mittlerweile haben wir alles an Informationen zusammen, um mit einem Beispiel beginnen zu können. Dabei werden Sie die Vorzüge von Maven erfahren und schätzen lernen. Es beginnt damit, dass Sie bereits für das Anlegen neuer Projekte inklusive Verzeichnisstruktur eine gute Unterstützung bekommen. Dazu dienen sogenannte Archetypen – der für Java generiert jedoch veraltete Verweise auf JUnit 3.8.1.

Schauen wir uns an, wie der Verzeichnisaufbau für eine einfach Applikation aussehen könnte: Zunächst finden wir im Verzeichnis `helloworld` eine Datei `pom.xml` und ein Verzeichnis `src` mit den Unterverzeichnissen `main/java` und `test/java`. Das entspricht der bereits vorgestellten Standardverzeichnisstruktur:

```
helloworld
¦
¦   pom.xml
¦
+---src
    +---main
    ¦   +---java
    ¦       +---de
    ¦           +---javaprofi
    ¦               +---helloworld
    ¦                       App.java
    ¦
    +---test
        +---java
            +---de
                +---javaprofi
                    +---helloworld
                            AppTest.java
```

Man erkennt sehr schön die Separierung von Sourcen (`src/main/java`) und Testcode (`src/test/java`). Auch ist hier eine Applikationsklasse `App.java` inklusive der zugehörigen Testklasse `AppTest.java` vorhanden. Darüber hinaus liegt die Projektbeschreibung in Form der Datei `pom.xml` vor. Das bietet einen guten Ausgangspunkt, um mit unserem Beispielprojekt zu starten.

Erzeugen eines Artefakts Wir wollen nun ein Artefakt mit Maven erzeugen, wechseln dazu in das gerade angelegte Verzeichnis `helloworld` und geben dann Folgendes ein:

```
mvn package
```

Auch hier werden Sie – wie beim Erzeugen des Projekts – bei der ersten Ausführung einige Zeit warten müssen, da Maven nochmals diverse benötigte Dateien aus dem Internet nachlädt. Wenn der Build-Lauf abgeschlossen ist, wurde ein Verzeichnis `target` mit folgendem Inhalt angelegt:

```
helloworld
¦
+---target
    ¦   helloworld-1.0-SNAPSHOT.jar
    ¦
    +---classes
    ¦   +---de
    ¦       +---javaprofi
    ¦           +---helloworld
    ¦                   App.class
    ¦
    +---maven-archiver
    ¦       pom.properties
    ¦
    +---surefire
    +---surefire-reports
    ¦       de.javaprofi.helloworld.AppTest.txt
    ¦       TEST-de.javaprofi.helloworld.AppTest.xml
    ¦
    +---test-classes
        +---de
            +---javaprofi
                +---helloworld
                        AppTest.class
```

Wie man daran sieht, kompiliert Maven die Klassen in das Zielverzeichnis `classes`. Außerdem führt Maven Unit Tests aus und protokolliert deren Ergebnisse im Verzeichnis `surefire-reports`. Verläuft all dies ohne Fehler, so werden die Projektdateien in Form eines JARs bereitgestellt. Maven erzeugt ein lauffähiges ausführbares Programm, was wir nun wie folgt starten:

```
java -cp target/helloworld-1.0-SNAPSHOT.jar de.javaprofi.helloworld.App
```

Damit wollen wir unsere Kurzeinführung in Maven beenden und ziehen ein Fazit.

Fazit

Wir haben einen ersten Eindruck davon gewinnen können, wie Maven mithilfe von Convention over Configuration und durch die Vorgabe sinnvoller Voreinstellungen und Abläufe den Build vereinfacht.

Allerdings ist die Beschreibung eines Projekts mithilfe einer `pom.xml` etwas geschwätzig und teilweise leicht unübersichtlich. Das gilt vor allem bei zunehmender Komplexität eines Projekts und vielen Abhängigkeiten von externen Bibliotheken. Oftmals erfordert Maven für den einen oder anderen Entwickler einen etwas höheren Einarbeitungsaufwand. Als Alternative schauen wir uns nun das Build-Tool Gradle an.

2.7.2 Builds mit Gradle

In diesem Abschnitt stelle ich einige Grundlagen zum Build-Tool Gradle vor. Wir werden dieses Wissen nutzen, um die Beispielapplikationen dieses Buchs unabhängig von der eingesetzten IDE kompilieren und starten zu können.

Installation

Gradle kann frei unter `http://www.gradle.org/downloads`[22] bezogen werden, wobei im September 2020 Version 6.6.1 aktuell ist. Nachdem Sie den Download abgeschlossen haben, entpacken Sie das Archiv in einen beliebigen Ordner, z. B. `C:\tools\gradle` oder `/usr/local/gradle` für Mac OS.[23] Danach ergänzen Sie die `PATH`-Variable um einen Eintrag auf dieses Installationsverzeichnis und das Unterverzeichnis `bin`. Öffnen Sie eine Konsole und tippen Sie dort das Kommando `gradle` ein. Daraufhin sollte Gradle starten und in etwa folgende Ausgaben produzieren:

```
Welcome to Gradle 6.6.1.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>

For troubleshooting, visit https://help.gradle.org

BUILD SUCCESSFUL in 2s
1 actionable task: 1 executed
```

Erscheint diese Ausgabe, so haben Sie das Build-Tool Gradle erfolgreich installiert und wir können unsere Entdeckungsreise zu Gradle mit einem konkreten Beispiel beginnen. Falls Sie Gradle jedoch nicht starten können, so prüfen Sie nochmals die korrekten Angaben in den Umgebungsvariablen und konsultieren Sie die Gradle-Homepage.

[22] Dort finden Sie diverse weitere Informationen rund um Gradle – insbesondere gibt es dort auch verschiedene Tutorials, die als Ergänzung zu diesem Text dienen können.

[23] Für Mac OS und Linux kann man auch die Tools `brew` oder `SDKMAN` nutzen.

Beispielprojekt

Wir werden gleich sehen, wie man ein Projekt mit Gradle per Hand aufsetzt, um verschiedene Dinge zu lernen.

Bevor wir aber Builds mit Gradle starten, werfen wir zunächst einen Blick auf das zu verwaltende Beispielprojekt `SimpleExampleApplication`. Weil dieses unabhängig von den sonstigen Programmen und Sourcen dieses Buchs existiert, legen Sie das Projekt bitte selbst an und folgen für die Verzeichnisstruktur den Maven-Richtlinien. Erstellen Sie bitte auch die den Packages entsprechenden Unterverzeichnisse, etwa mit `mkdir -p de/javaprofi/helloworld`. Schließlich benötigen wir noch zwei Java-Klassen, für die es anfangs noch keine Tests gibt. Daher ist das Unterverzeichnis `src/test/java` zunächst leer:

```
SimpleExampleApplication
`-- src
    |-- main
    |   `-- java
    |       `-- de
    |           `-- javaprofi
    |               `-- helloworld
    |                   |-- HelloWorld.java
    |                   `-- HelloWorld2.java
    `-- test
...
```

Beide Java-Klassen besitzen eine extrem einfache Implementierung und führen lediglich eine Ausgabe auf der Konsole aus:

```
package de.javaprofi.helloworld;

public class HelloWorld
{
    public static void main(final String[] args)
    {
        System.out.println(getMessage());
    }

    public static String getMessage()
    {
        return "Hello World";
    }
}
```

Die zweite Java-Klasse wird hier nicht gezeigt, da sie nahezu identisch realisiert wird und nur einen leicht abweichenden Text ausgeben soll.

Das erste Build-Skript

Wir wollen nun die Java-Klassen mit Gradle kompilieren und daraus ein JAR bauen. Ähnlich wie bei Maven werden die Abläufe mithilfe einer speziellen Build-Datei (auch Build-Skript genannt) festgelegt, die standardmäßig `build.gradle` heißt. Sie muss im Hauptverzeichnis des Projekts liegen, also für das Beispielprojekt im Verzeichnis `SimpleExampleApplication`. Im Build-Skript wird der Ablauf des Builds

als Groovy-Skript (oder mittlerweile auch Kotlin) durch verschiedene Tasks mithilfe einer eigenen DSL (Domain Specific Language) beschrieben. Ein solcher Task ist eine Aufgabenbeschreibung, wie etwa das Kompilieren oder das Ausführen von Tests.

Praktischerweise liefert Gradle bereits eine Vielzahl an Erweiterungen (Plugins) mit, die viele Tasks zur Verfügung stellen. Basierend auf dem Plugin für Java erzeugen wir die Build-Datei namens `build.gradle` folgendermaßen:

```
apply plugin: 'java'
```

Auch wenn es kaum zu glauben ist, diese eine Zeile stellt eine erste, aber vollständige Build-Datei dar, die neben dem Kompilieren und Erstellen eines JARs auch diverse weitere Funktionalität in Form vordefinierter Tasks bietet. Schauen wir uns die Möglichkeiten an.

Anzeige aller verfügbaren Tasks

Zur Prüfung der Installation hatten wir das Kommando `gradle` ausgeführt. In der Konsolenausgabe wurde das Kommando

```
gradle tasks
```

angepriesen, um die verfügbaren Tasks anzuzeigen. Führen wir dieses Kommando aus, so wird folgende beeindruckende Liste verfügbarer Tasks auf der Konsole ausgegeben (hier zur Lesbarkeit etwas mit ... gekürzt und leicht umformatiert):

```
> Task :tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that ...
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles classes 'main'.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles classes 'test'.
...

Documentation tasks
-------------------
javadoc - Generates Javadoc API documentation for the main source code.

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root...
dependencies - Displays all dependencies declared in root project 'Simple....'.
...
tasks - Displays the tasks runnable from root project 'SimpleExampleApplication'
      .
...
```

```
Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.
...
```

Anhand der Ausgaben erkennt man, dass Gradle das aktuelle Verzeichnis als Projektverzeichnis ansieht und deswegen entsprechend dem Verzeichnisnamen von einem gleichnamigen Projekt `SimpleExampleApplication` ausgeht.

Sourcen kompilieren

Probieren wir einen Build-Lauf aus und tippen Folgendes ein:

```
gradle build
```

Gradle startet seine Arbeit und kurz darauf erscheint unter anderem folgende Zusammenfassung als Konsolenausgabe:

```
BUILD SUCCESSFUL in 0s
4 actionable tasks: 4 executed
```

Während frühere Versionen von Gradle umfangreiche Ausgaben erzeugt haben, ist die Konsolenausgabe seit Gradle 4 eher minimalistisch. Allerdings lässt sich eine ausführlichere Ausgabe per Kommandozeilenparameter aktivieren. Das ist zur leichteren Auswertbarkeit und Nachvollziehbarkeit und vor allem auch für Continuous Integration Builds sinnvoll, da die einzelnen Aktionen besser verdeutlicht werden. Dafür startet man einen Build folgendermaßen:

```
gradle --console plain clean build
```

Das produziert folgende Ausgaben:

```
:clean
:compileJava
:processResources NO-SOURCE
:classes
:jar
:assemble
:compileTestJava
:processTestResources NO-SOURCE
:testClasses
:test
:check
:build

BUILD SUCCESSFUL in 0s
5 actionable tasks: 5 executed
```

Die Ausgaben lassen die im Build-Lauf abgearbeiteten Schritte erahnen: Zunächst werden die Java-Dateien kompiliert (Task `compileJava`) und später zu einem JAR (Task `jar`) zusammengefasst. Während des Builds entsteht ein Verzeichnis `build`,

in dem erzeugte Dateien landen. Im Unterverzeichnis `libs` entsteht ein JAR namens `SimpleExampleApplication.jar`, das die beiden kompilierten Klassen enthält:

```
SimpleExampleApplication
|-- build
|   |-- classes
|   |   `-- java
|   |       `-- main
|   |           `-- de
|   |               `-- javaprofi
|   |                   `-- helloworld
|   |                       |-- HelloWorld.class
|   |                       `-- HelloWorld2.class
|   |-- libs
|   |   `-- SimpleExampleApplication.jar
```

Neben dem Kompilieren ist es auch wünschenswert, Dokumentation zu generieren und Unit Tests auszuführen. Experimentieren wir noch ein wenig herum und lernen die dazu notwendigen Gradle-Kommandos kennen.

Hinweis: Inkrementelle Builds

Als Besonderheit ermittelt Gradle, ob die Ausführung eines Tasks überhaupt notwendig ist oder ob dieser zuvor schon erfolgreich ausgeführt wurde. Dann protokolliert Gradle dies mit der Ausgabe von `UP-TO-DATE` und überspringt die Ausführung, wodurch sich Build-Läufe häufig zeitlich recht kurz halten lassen.

Ausführen von Unit Tests

Die Ausführung vorhandener Unit Tests sollte ein fester Bestandteil des Build-Laufs sein. Werfen wir deshalb einen Blick auf das Testen und prüfen die Funktionalität der Klasse `HelloWorld` durch folgenden Unit Test:

```
package de.javaprofi.helloworld;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class HelloWorldTest
{
    @Test
    public void testGetMessage()
    {
        assertEquals("Hello World", HelloWorld.getMessage());
    }
}
```

Diese Testklasse wird im Verzeichnis `src/test/java/de/javaprofi/helloworld` gespeichert. Zum Ausführen des Tests geben wir Folgendes ein:

```
gradle test
```

Statt der Ausführung unseres Tests kommt es jedoch zu Kompilierfehlern. Die Ursache liegt darin, dass die benötigte JUnit-Bibliothek (noch) nicht im Build-Lauf verfügbar ist. Dazu müssen wir eine externe Abhängigkeit angeben, bei der wir uns praktischerweise auf das zentrale Maven-Repository bzw. dessen Obermenge, referenziert durch `jcenter()`, beziehen können.

Externe Abhängigkeit spezifizieren Die Definition von Abhängigkeiten ist mit Gradle kurz, klar und selbsterklärend. In diesem Beispiel wird die Abhängigkeit zu JUnit während des Kompilierens und zur Ausführung der Tests benötigt. Dies lässt sich wie folgt spezifizieren – auch der Artefaktname wird direkt sichtbar:

```
apply plugin: 'java'

repositories
{
    jcenter()
}

dependencies
{
    testCompile('org.junit.jupiter:junit-jupiter:5.7.0')
    testRuntime('org.junit.vintage:junit-vintage-engine:5.7.0')
}

test
{
    // JUnit 5 nutzen!!
    useJUnitPlatform()
}
```

Führen wir nun erneut das Kommando `gradle test` aus, so werden die Abhängigkeiten aufgelöst und dazu die JUnit-Bibliotheken aus dem Internet nachgeladen. Zudem werden die vorhandenen und automatisch ermittelten Unit Tests ausgeführt. Dadurch ergibt sich folgende Verzeichnisstruktur (gekürzt):

```
SimpleExampleApplication
|-- build
|   |-- classes
|   |   `-- java
|   |       |-- main
|   |       |   `-- de
|   |       |       `-- javaprofi
|   |       |           `-- helloworld
|   |       |               |-- HelloWorld.class
|   |       |               `-- HelloWorld2.class
|   |       `-- test
|   |           `-- de
|   |               `-- javaprofi
|   |                   `-- helloworld
|   |                       `-- HelloWorldTest.class
|   |-- libs
|   |   `-- SimpleExampleApplication.jar
|   |-- reports
|   |   `-- tests
|   |       `-- test
|   |           |-- index.html
```

Man sieht dort auch einen Ordner `reports`. Dort finden sich von JUnit (und anderen Tools) generierte Test-Reports. Ein Beispiel ist in Abbildung 2-15 dargestellt.

Test Summary

1	0	0	0.001s
tests	failures	ignored	duration

100% successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
de.javaprofi.helloworld	1	0	0	0.001s	100%

Abbildung 2-15 *Von JUnit in HTML erzeugter Bericht*

Sonderbehandlung fehlschlagender Tests In der Regel sollte ein fehlgeschlagener Test zum Abbruch des Build-Prozesses führen. Eine abweichende Handhabung ist beispielsweise dann nützlich, wenn zwar normalerweise der erfolgreiche Abschluss der Unit Tests für alle Builds verbindlich ist, jedoch während einer Prototypentwicklung einige Tests temporär fehlschlagen dürfen. Das kann man über die Angabe von `ignoreFailures` steuern:

```
test
{
    ignoreFailures = true
}
```

Tipp: Abhängigkeiten definieren

Für einen Build können verschiedene Abhängigkeiten existieren. Für jede davon kann durch verschiedene Schlüsselwörter in der Sektion `dependencies` festgelegt werden, welchen Gültigkeitsbereich sie besitzt. Abhängigkeiten können beim Kompilieren (`compile`), beim Ausführen (`runtime`), beim Kompilieren von Tests (`testCompile`) und beim Ausführen der Tests (`testRuntime`) existieren und durch die zuvor in Klammern angegebenen Schlüsselwörter festgelegt werden. Standardmäßig sind alle Abhängigkeiten von `testRuntime` eine Obermenge von `testCompile`, was wiederum alle Abhängigkeiten von `compile` enthält. Demnach bildet `compile` die Basis und auch die kleinste Menge von Abhängigkeiten.

Mittlerweile sollte man aber `implementation` statt `compile` sowie `testImplementation` statt `testCompile` nutzen. Beides wurde mit der Modularisierung eingeführt, um Abhängigkeiten besser steuern zu können, insbesondere auch API-Abhängigkeiten, wofür man `api` nutzt, was zu einer transitiven Abhängigkeit führt. Details finden Sie unter `https://docs.gradle.org/current/userguide/java_library_plugin.html`.

Javadoc generieren

Schauen wir zunächst, wie einfach man eine Javadoc-Dokumentation zu den Klassen erzeugen kann. Wir geben dazu Folgendes ein:

```
gradle javadoc
```

Dieses Kommando erzeugt für alle Klassen des Projekts eine Javadoc-Dokumentation im Unterverzeichnis `build/docs/javadoc`.

JAR erstellen

Zum Erzeugen eines Artefakts kann man den `jar`-Task nutzen. Auch hier geschieht in der Regel vieles automatisch. Sollen jedoch spezielle Attribute in der Manifest-Datei gesetzt werden, so kann man dies einfach wie folgt spezifizieren:

```
jar
{
    manifest
    {
        attributes ( "Implementation-Title" : "<title>",
                     "Implementation-Version" : version )
    }
}
```

Oftmals enthält das JAR eine zu startende `main`-Klasse. Dazu schreibt man Folgendes:

```
jar
{
    manifest
    {
        attributes ( "Main-Class" : "de.javaprofi.helloworld.HelloWorld" )
    }
}
```

Teilweise enthält ein JAR aber auch mehrere Klassen mit `main()`-Methoden. Die Frage ist nun, wie man die `main()`-Methode einer beliebigen gewünschten Klasse aus einem JAR ausführen kann. Dazu lernen wir nun die Definition eines eigenen Tasks kennen.

Eigene Tasks definieren – Abhängigkeiten zwischen Tasks

Bislang haben wir diverse Funktionalität allein mit Gradle-Bordmitteln realisieren können, jedoch ist es damit nicht direkt möglich, die `main()`-Methoden verschiedener Klassen auszuführen. Diese Funktionalität benötigen wir aber beispielsweise zum Starten der diversen Beispielprogramme aus den Kapiteln dieses Buchs, in denen in der Regel eine Vielzahl startbarer Demo-Applikationen enthalten ist.

Zur Vervollständigung unseres Build-Laufs nutzen wir das Schlüsselwort `task`, um einen eigenen Task zum Ausführen von `main()`-Methoden beliebiger Klassen zu definieren. Den neuen Task bauen wir auf dem vordefinierten Typ `JavaExec` auf und erwei-

tern diesen um die benötigte Funktionalität bzw. hier um die Angaben zu `CLASSPATH` und `main`-Klasse wie folgt:

```
task HelloWorld(type: JavaExec) { // <- öffnende Klammer muss aufgrund von
                                  // Groovy-Besonderheiten hier stehen
    dependsOn jar

    // Groovy-Notation [] für Liste
    classpath = files(["build/libs/SimpleExampleApplication.jar"])
    main = "de.javaprofi.helloworld.HelloWorld" // Achtung: ohne .class
}
```

Wie man bei der Definition sieht, existieren gewisse Abhängigkeiten der Tasks untereinander. Im obigen Beispiel soll beim Aufruf von `gradle HelloWorld` zunächst der Task `jar` abgearbeitet werden, um die Applikation zu bauen. Die Reihenfolge zwischen Tasks lässt sich durch das Schlüsselwort `dependsOn` festlegen. Damit wird definiert, dass etwa das Kompilieren vor dem Testen erfolgt oder aber das Bereinigen von Verzeichnissen vor dem Kompilieren.

Während die Angabe der `main`-Klasse noch recht intuitiv ist, bedarf es bei der Angabe des zu nutzenden `CLASSPATH` noch eines kleinen Hinweises. Dort finden wir die Methode `files()`. Sie ist Bestandteil der Gradle DSL und erlaubt es, eine Menge von Dateien zu spezifizieren. In diesem Fall ist es genau eine Datei, nämlich `SimpleExampleApplication.jar` im Unterverzeichnis `build/libs`. Die Angabe in eckigen Klammern ist ein sogenanntes Collection-Literal und beschreibt in Groovy und somit auch in Gradle eine Liste.

Code-Checker-Tool einbinden

Mit recht wenig Aufwand haben wir schon einen vollständigen Build-Lauf beschreiben können. Als Schmankerl möchte ich noch ein paar Sourcecode-Prüfungen ergänzen. Dazu nutzen wir das Tool PMD, für das Gradle ein Plugin bereitstellt, sodass wir lediglich folgende Zeile in unserer Datei `build.gradle` ergänzen müssen:

```
apply plugin: 'pmd'
```

Führen wir den Build mit

```
gradle check
```

aus, entstehen im Verzeichnis `build/reports/pmd` einige Report-Dateien. Allerdings zeigen diese keine Verstöße. Das liegt schlichtweg daran, dass PMD noch keine Regeln konfiguriert hat. Wir fügen wie folgt ein paar Regeln ein:

```
pmd
{
    ruleSets = [ "category/java/design.xml" ]
}

tasks.withType(Pmd) {

    ignoreFailures = true           // execute all the other checkers too

    reports {
        xml.enabled = false
        html.enabled = true
    }
}

task showPmdReport { doLast {
    File file = file("$buildDir/reports/pmd/main.html")
    java.awt.Desktop.desktop.browse file.toURI()
} }
```

Führen wir nun erneut einen Build aus, so werden uns Verstöße gemeldet sowie das Verzeichnis, in dem wir eine detaillierte HTML-Seite dazu finden:

```
> Task :pmdMain
Removed misconfigured rule: LoosePackageCoupling  cause: No packages or classes
      specified
1 PMD rule violations were found. See the report at: file:///Users/michaeli/
      SimpleExampleApplication/build/reports/pmd/main.html

> Task :pmdTest
Removed misconfigured rule: LoosePackageCoupling  cause: No packages or classes
      specified
```

Als erster Einstieg in die Sourcecode-Prüfung soll der Aufruf von PMD hier genügen. Später in Abschnitt 19.4 behandle ich das Thema ausführlicher.

IDE-Projekte erzeugen

Betrachten wir nun die Integration von Gradle-Builds in die IDE. Praktischerweise ist es mit Gradle möglich, spezifische Projektdateien für unsere IDE generieren zu lassen.

Eclipse-Projekt erzeugen und einbinden Um das dazu benötigte Gradle-Plugin zu integrieren, ergänzen wir folgende Angabe in der Build-Datei:

```
apply plugin: 'eclipse'
```

Als Folge stehen uns mit `eclipse` und `cleanEclipse` zwei weitere Tasks für Gradle zur Verfügung. Wenn wir den Task `gradle eclipse` ausführen, dann werden Eclipse-spezifische Dateien, nämlich `.project` und `.classpath`, erzeugt. Darüber hinaus kümmert sich Gradle um das Dependency Management und lädt benötigte JAR-Dateien herunter und bindet diese als Referenced Libraries korrekt in Eclipse-Projekte ein.

Um ein wie eben erzeugtes Projekt in Eclipse zu importieren, wählen Sie das Kontextmenü IMPORT –> GENERAL –> EXISTING PROJECTS INTO WORKSPACE.

Tipp: Integration in Eclipse

Zu Eclipse kompatible Verzeichnisstruktur Bisher haben wir das Maven-Standardverzeichnislayout genutzt. Gradle ist aber diesbezüglich flexibel. Mit nur ein wenig Konfigurationsaufwand kann man auch Projekte mit Gradle bauen, die eine abweichende Verzeichnisstruktur besitzen, etwa eine, die dem Eclipse-Standardverzeichnislayout entspricht. Bevor allerdings Anpassungen erfolgen, sollte man sich überlegen, ob man nicht eher das Eclipse-Projekt so konfiguriert, dass es sich an den Maven- bzw. Gradle-Standard hält.

Zur Konfiguration nutzt man die Angabe `sourceSets`. Damit wird eine Menge von Dateien beschrieben, die kompiliert werden sollen. Zudem kann man über die Angabe `output.classesDir` das Ausgabeverzeichnis konfigurieren:

```
apply plugin: 'java'

sourceSets
{
    main
    {
        java
        {
            srcDir = ['src']
            output.classesDir = ['bin']
        }
    }
}
```

Multi-Project-Builds

Je größer Softwareprojekte werden, desto mehr bietet es sich an, diese in Subprojekte aufzuteilen. Das führt zu einer besseren Modularisierung und man kann (unerwünschte) Abhängigkeiten besser kontrollieren. Die Strukturierung geschieht in etwa wie folgt:

```
MultiProject
|
+---Subproject_1
+---Subproject_2
+---Subproject_3
```

Hier ist angedeutet, dass das Hauptprojekt in drei Subprojekte aufgegliedert ist. Eine solche Dreiteilung findet man etwa bei einer Schichtenarchitektur mit GUI, Service- bzw. Modellschicht und Datenzugriffsschicht.[24] Dabei hängt das GUI von der Serviceschicht ab und diese wiederum von der Datenzugriffsschicht.

[24] Natürlich sind andere Aufteilungen denkbar.

Wenn wir dies mit einem Gradle-Build beschreiben wollen, dann enthält das Root-Projektverzeichnis lediglich die Dateien `build.gradle` und `settings.gradle`. In Letzterer werden die Subprojekte z. B. folgendermaßen aufgelistet:

```
include 'gui', 'model', 'dao'
```

Die Hauptdatei enthält einen Abschnitt `subprojects`, in dem die Gemeinsamkeiten für alle Subprojekte aufgeführt werden:

```
subprojects
{
    apply plugin: 'java'

    repositories
    {
        jcenter()
    }
}
```

Außerdem wird für jedes Subprojekt ein eigenes Verzeichnis angelegt, und dort findet sich wiederum eine `build.gradle`-Datei. Diese ist dann so aufgebaut, wie es zuvor in diesem Kapitel beschrieben wurde.

```
MultiProject
¦
¦   build.gradle
¦   settings.gradle
¦
+---gui
¦   ¦   build.gradle
¦   ¦
¦   +---src
¦       +---main
¦           +---java
¦               +---de
¦                   +---gui
¦                           PersonGui.java
¦
+---model
¦   ¦   build.gradle
¦   ¦
¦   +---src
¦       +---main
¦           +---java
¦               +---de
¦                   +---model
¦                           Person.java
¦                   +---service
¦                           PersonService.java
¦
+---dao
    ...
```

Man kann dann im Hauptprojekt `gradle build` eingeben und die Subprojekte werden gebaut. In jedem Subprojekt entsteht als Folge – wie zuvor für normale Projekte auch – im Verzeichnis `build` eine JAR-Datei.

Zwischenfazit

Ausgehend von der einfachsten Build-Datei, bestehend aus einer Zeile, haben wir schrittweise den Build um sinnvolle Funktionalität ergänzt. Dabei wurde das Verwalten von externen Abhängigkeiten, das Ausführen von Analysetools und Dokumentationswerkzeugen vorgestellt. Auch das Zusammenspiel mit der genutzten IDE wurde kurz thematisiert.

2.7.3 Vorteile von Maven und Gradle

Sowohl Gradle als auch Maven haben ihre spezifischen Vorzüge und Stärken sowie wenige kleinere Eigenheiten oder Schwächen. Maven und Gradle bieten generell folgende Vorteile:

- Sie fördern eine Vereinheitlichung durch Convention over Configuration, wodurch Konfigurationsaufwände nur dort anfallen, wo vom Standard abgewichen wird.
- Sie erleichtern die Einarbeitung und Orientierung, da eine einheitliche Projektstruktur genutzt wird. Auch Projektneulinge finden sich so schnell zurecht, denn alles ist relativ vertraut und man muss nicht von Projekt zu Projekt umdenken.
- Sie erlauben, Abhängigkeiten von Fremdbibliotheken deklarativ zu beschreiben und automatisch durch das Build-Tool auflösen zu lassen (insbesondere auch transitiv, also die von einer Bibliothek benötigten anderen Bibliotheken).
- Sie vereinfachen die Generierung von Dokumentation und Reports zu Sourcecode-Qualität durch Aufruf von Tools wie Javadoc, Checkstyle, PMD, FindBugs usw.

Fazit

In diesem Abschnitt haben Sie das Handwerkszeug kennengelernt, um den Sourcecode und die Unit Tests der Beispielapplikationen aus den nachfolgenden Kapiteln kompilieren und ausführen zu können.

Des Weiteren wurde motiviert, dass ein Softwareprojekt stets mit einem vernünftigen Build-Management gekoppelt sein sollte. Je größer ein Projekt wird, desto mehr profitiert man von den Automatiken von Maven und Gradle. Letzteres vereinfacht den Build-Prozess auch für kleine Projekte und erfordert lediglich minimalen Einarbeitungsaufwand, sodass für dieses Buch die Wahl auf Gradle gefallen ist.

2.8 Weiterführende Literatur

Dieses Kapitel hat einen einführenden Überblick gegeben, wie Sie sich eine professionelle Arbeitsumgebung einrichten. Weitere Ideen zu deren Ausgestaltung finden Sie in den folgenden Büchern und Onlinequellen, wobei die meistens davon detaillierte Informationen zu den Themen Versionsverwaltungen und Build-Automatisierung liefern.

- »**Agile Java-Entwicklung in der Praxis**« von Michael Hüttermann [37]
- »**Pragmatisch Programmieren: Versionsverwaltung mit CVS**« von Andrew Hunt und David Thomas [36]
- »**Versionskontrolle mit Subversion**« von Ben Collins-Sussman, Brian W. Fitzpatrick und C. Michael Pilato [12]
- »**Pragmatic Version Control Using Subversion**« von Mike Mason [51]
- »**Git**« von René Preißel und Bjørn Stachmann [62]
- »**Mercurial: The Definitive Guide**« von Bryan O'Sullivan [60]
- »**Pragmatisch Programmieren: Projekt-Automatisierung**« von Mike Clark [10]
- »**Gradle In Action**« von Benjamin Muschko [54]
- »**Git**« `http://git-scm.com/book/de`
- »**Mercurial**« `http://hgbook.red-bean.com/`
- »**Maven**« `http://books.sonatype.com/mvnref-book/reference/`

3 Objektorientiertes Design

Dieses Kapitel gibt einen Einblick in den objektorientierten Entwurf von Software. Die zugrunde liegende Idee der Objektorientierung (OO) ist es, ***Zustand*** (Daten) mit ***Verhalten*** (Funktionen auf diesen Daten) zu verbinden. Prinzipiell lässt sich diese Idee in jeder Programmiersprache realisieren. Sprachen wie z. B. C++ und Java unterstützen die objektorientierten Konzepte von Hause aus. In diesem Kapitel möchte ich Ihnen die Grundgedanken der Objektorientierung und die für gutes Design verfolgten Ziele ***Kapselung***, ***Trennung von Zuständigkeiten*** und ***Wiederverwendbarkeit*** vermitteln.

Abschnitt 3.1 stellt zunächst einige Grundbegriffe und -gedanken vor und verdeutlicht diese am Entwurf eines Zählers. Das Beispiel ist bewusst einfach gewählt. Nichtsdestotrotz kann man daran bereits sehr gut den OO-Entwurf mit seinen Tücken nachvollziehen. Die gewonnenen Erkenntnisse helfen, die OO-Grundlagen besser zu verstehen. Um Objektorientierung jedoch als Konzept zu erfassen und das ***Denken in Objekten und Zuständigkeiten*** zu verinnerlichen, muss man auch Fallstricke und Grenzen kennen. Dazu zeigt Abschnitt 3.1.3 ein paar Irrwege der »funktionalen Objektierung«[1]. Darunter versteht man, dass häufig fälschlicherweise von Objektorientierung gesprochen wird, nur weil man Klassen und Objekte verwendet. Allerdings umfasst Objektorientierung viel mehr als das Programmieren mit Klassen und den intensiven Einsatz von Vererbung. Deshalb erläutere ich in Abschnitt 3.2 einige technische Grundlagen, wie Interfaces und abstrakte Klassen, bevor wir mit diesem Wissen in Abschnitt 3.3 das Thema Vererbung erneut aufgreifen und es kritisch diskutieren. Zudem betrachten wir Situationen von »explodierenden« Klassenhierarchien und Wege, die Probleme zu lösen. In Abschnitt 3.4 thematisieren wir unter anderem Read-only-Interfaces und Immutable-Klassen als fortgeschrittenere OO-Techniken. Danach lernen wir in Abschnitt 3.5 verschiedene Prinzipien guten OO-Entwurfs als Hilfestellung auf dem Weg zu einem gelungenen Design kennen. Dazu werden einige bekannte Designrichtlinien wie etwa das Law of Demeter sowie die SOLID-Prinzipien vorgestellt.

Mit JDK 5 wurden generische Typen, auch ***Generics*** genannt, eingeführt, um typsichere Klassen und insbesondere typsichere Containerklassen realisieren zu können. Eine wichtige Rolle zum Verständnis beim Überschreiben von Methoden, bei der Nutzung von Arrays und insbesondere beim Einsatz von Generics spielt der Begriff Varianz und verschiedene Formen davon. Diese werden in Abschnitt 3.6 vorgestellt. Abschließend bietet Abschnitt 3.7 einen Einstieg in das Thema Generics.

[1] Danke an meinen Freund Tim Bötzmeyer für diese schöne Wortschöpfung.

3.1 OO-Grundlagen

Die objektorientierte Softwareentwicklung setzt eine spezielle Denkweise voraus. Kerngedanke dabei ist, den Programmablauf als ein Zusammenspiel von Objekten und ihren Interaktionen aufzufassen. Dabei erfolgt eine Anlehnung an die reale Welt, in der Objekte, etwa Personen, Autos, CD-Spieler usw., und ihre Interaktionen ein wesentlicher Bestandteil sind. All diese Dinge werden durch spezielle Merkmale und Verhaltensweisen charakterisiert. Auf die Software übertragen realisieren viele einzelne Objekte durch ihr Verhalten und ihre Eigenschaften die benötigte und gewünschte Programmfunktionalität. Bevor ich diese Gedanken für den Entwurf eines Zählers wieder aufgreife, möchte ich zuvor einige Begriffe definieren, die für ein gutes Verständnis dieses und der folgenden Kapitel eine fundamentale Basis darstellen.

3.1.1 Grundbegriffe

Klassen und Objekte Sprechen wir beispielsweise über ein spezielles Auto, etwa das von Hans Mustermann, so reden wir über ein konkretes ***Objekt***. Sprechen wir dagegen abstrakter von Autos, dann beschreiben wir eine Klasse. Eine ***Klasse*** ist demnach eine Strukturbeschreibung für Objekte und umfasst eine Menge von ***Attributen*** (auch ***Membervariablen*** genannt) und ***Methoden***, die in der Regel auf diesen Daten arbeiten und damit Verhalten definieren. Ein Objekt ist eine konkrete Ausprägung (***Instanz***) einer Klasse. Die Objekte einer Klasse unterscheiden sich voneinander durch ihre Adresse im Speicher und eventuell in den Werten ihrer Attribute.

Objektzustand Der ***Objektzustand*** beschreibt die momentane Wertebelegung aller Attribute eines Objekts. Er sollte in der Regel nur durch Methoden des eigenen Objekts modifiziert werden, um Veränderungen leichter unter Kontrolle zu halten.

Schnittstelle bzw. Interface Häufig werden die Begriffe Schnittstelle und ***Interface*** analog zueinander genutzt. Mit beidem sind diejenigen Methoden einer Klasse gemeint, die von anderen Klassen aufgerufen werden können. Schnittstellen definieren ein »Angebot von Verhalten«. Java kennt das Schlüsselwort `interface` zur expliziten Definition der Schnittstelle einer Klasse in Form einer ***Menge von Methoden ohne Implementierung***.[2] Man spricht hier von ***abstrakten Methoden***. Eine Klasse, die das Interface implementiert, muss diese abstrakten Methoden mit Funktionalität füllen.

Abstrakte Klassen Ähnlich wie Interfaces erlauben ***abstrakte Klassen*** die Vorgabe einer Schnittstelle in Form von Methoden. Im Unterschied zu Interfaces können in abstrakten Klassen nicht nur Methoden implementiert werden,[3] sondern auch Attribute

[2] Auch ohne explizite Beschreibung besitzt jede Klasse eine Schnittstelle in Form der von ihr definierten Methoden, die für andere Klassen aufrufbar sind.

[3] Mit Java 8 ändert sich diese strenge Unterscheidung, da es dann erlaubt ist, in Interfaces mithilfe sogenannter Defaultmethoden bereits ein Standardverhalten vorzugeben.

definiert werden, um eine Basisfunktionalität anzubieten. Methoden, die noch keine Implementierung besitzen, sind mit dem Schlüsselwort `abstract` zu kennzeichnen. Aufgrund der unvollständigen Implementierung lassen sich keine Objekte abstrakter Klassen erzeugen und die Klasse muss ebenfalls mit dem Schlüsselwort `abstract` versehen werden. Dadurch sind abstrakte Klassen nicht mehr instanziierbar. Das gilt übrigens auch für den Spezialfall, dass keine einzige Methode, sondern nur die Klasse `abstract` definiert ist.

Typen Durch Klassen und Interfaces werden eigenständige ***Datentypen*** oder kurz Typen mit Verhalten beschrieben, etwa eine Klasse `Auto`. Im Unterschied dazu existieren ***primitive Datentypen***, die lediglich Werte ohne Verhalten darstellen. Beispiele hierfür sind die Zahlentypen `int`, `float` usw.

Realisierung Das Implementieren eines Interface durch eine Klasse wird ***Realisierung*** genannt und durch das Schlüsselwort `implements` ausgedrückt.[4] Eine Realisierung eines Interface durch eine Klasse bedeutet, dass sich ein Objekt dieser Klasse so verhalten kann, wie es das Interface vorgibt. Daher spricht man von ***Typkonformität*** oder auch von einer ***»behaves-like«-Beziehung*** bzw. ***»can-act-like«-Beziehung***. Wichtig dafür sind zum einen sprechende Methodennamen sowie zum anderen eine aussagekräftige Dokumentation zu der jeweiligen Methode. Für ein gelungenes Design reicht es also nicht aus, dass ein Interface nur aus Methodendeklarationen besteht!

Deklaration und Definition Eine ***Deklaration*** beschreibt bei Variablen deren Typ und ihren Namen, etwa `int age`. Bei Methoden entspricht eine Deklaration dem Methodennamen, den Typen der Übergabeparameter sowie angegebenen Exceptions. Diese Elemente beschreiben die ***Signatur*** einer Methode. Von einer ***Definition*** einer Variablen spricht man, wenn ihr bei der Deklaration ein Wert zugewiesen wird. Die Definition einer Methode entspricht der Methodenimplementierung.

Info: Signatur einer Methode

Unter der Signatur einer Methode versteht man die Schnittstelle, die aus dem Namen, der Parameterliste (Anzahl, Reihenfolge und Typen der Parameter) und optional angegebenen Exceptions besteht. Betrachten wir folgende Methode `open()`:

```
boolean open (final String filename, final int retries) throws IOException
{
    // ...
```

In Java gehören weder die Implementierung, der Rückgabewert noch die Namen der Parameter und die `final`-Schlüsselwörter zur Signatur, sondern nur Folgendes:

```
open (String, int) throws IOException
```

[4] Werden nicht alle Methoden des Interface implementiert, muss die Klasse `abstract` sein.

Warum ist das so? Nehmen wir dazu folgende Methodensignatur für `open()` an:

```
open(File, boolean) throws IOException
```

Erfolgt ein Aufruf an die Methode `open()`, so kann der Compiler basierend auf den Typen der Parameterliste die korrekte Methode auswählen. Der Rückgabewert trägt nicht zur Unterscheidung beim Aufruf bei, weil er nicht zwangsweise ausgewertet werden muss (in der Regel aber sollte). Somit ist er nicht Bestandteil der Signatur.

Referenzen Definiert man in Java eine Variable vom Typ einer Klasse, so stellt diese nicht das Objekt selbst dar, sondern nur eine Referenz auf das Objekt. Eine solche Variable ist ein Verweis, um das Objekt zu erreichen[5] und mit dessen Daten und Methoden zu arbeiten. Abbildung 3-1 zeigt dies für eine Referenzvariable `myBrother`, die auf ein Objekt vom Typ `Person` verweist und somit Zugriff auf dessen Attribute (hier `name` und `age`) sowie dessen (nicht dargestellte) Methoden erlaubt.

Abbildung 3-1 *Objekterzeugung und -referenzierung*

Die folgende Abbildung 3-2 zeigt die Referenzierung desselben Objekts durch mehrere Referenzvariablen, hier `myBrother` und `otherPerson`.

Abbildung 3-2 *Mehrfache Referenzierung eines Objekts*

Bereits beim bloßen Betrachten erahnt man mögliche Auswirkungen von Änderungen von Attributen: Wird etwa der Wert des Attributs `name` für das durch `myBrother` referenzierte `Person`-Objekt verändert, so wirkt sich das natürlich auch in dem durch `otherPerson` referenzierten Objekt aus. Das ist in diesem Beispiel noch leicht verständlich, verweisen doch beide Referenzen auf *dasselbe* Objekt. In der Praxis sind die Beziehungsgeflechte zwischen Objekten oftmals komplexer, und somit lassen sich Referenzierungen und Auswirkungen von Änderungen an den Daten eines Objekts nicht immer so einfach nachvollziehen wie hier.

[5] Dies entspricht der Adresse im Speicher, wo das Objekt nach seiner Erzeugung abgelegt ist.

Lebenszyklus von Objekten und Speicherfreigabe Der Lebenszyklus eines Objekts beginnt bei dessen Konstruktion. Danach kann es mit anderen Objekten interagieren. Irgendwann ist ein Objekt an sein »Lebensende« gelangt. Dies ist der Fall, wenn keine Referenzvariable mehr auf dieses Objekt verweist. Folglich kann es von keiner Stelle im Programm mehr erreicht und angesprochen werden. Der genaue Zeitpunkt des »Ablebens« und der damit verbundenen Speicherfreigabe kann in der JVM nicht exakt bestimmt werden. Das liegt daran, dass die Aufräumarbeiten und die Speicherfreigabe durch einen speziellen Mechanismus zur Speicherbereinigung, die sogenannte ***Garbage Collection***, erfolgen. Die konkreten Zeitpunkte der Ausführung sind nicht vorhersehbar, da diese von der gewählten Aufräumstrategie und des gerade belegten und derzeit noch freien Speichers abhängig sind. Details dazu beschreibt Abschnitt 12.4.

Sichtbarkeiten Beim objektorientierten Programmieren unterscheidet man verschiedene Sichtbarkeiten, die festlegen, ob und wie andere Klassen auf Methoden und Attribute zugreifen dürfen. Java bietet folgende vier Sichtbarkeiten:

- `public` – Ist von überall aus zugreifbar.[6]
- `protected` – Zugriff für abgeleitete Klassen und alle Klassen im selben Package
- Package-private oder default (kein Schlüsselwort[7]) – Nur Klassen aus demselben Package haben Zugriff darauf.
- `private` – Nur die Klasse selbst und alle ihre inneren Klassen haben Zugriff.

Kapselung Die ***Kapselung*** von Daten (***Information Hiding***) stellt einen Weg dar, die Attribute eines Objekts vor dem direkten Zugriff und der direkten Manipulation durch andere Objekte zu schützen. Anstatt dass eine Klasse ihre Attribute `public` definiert und damit potenziellen Nutzern direkten Zugriff erlaubt, empfiehlt es sich, die Sichtbarkeit der Attribute auf `protected`, Package-private oder `private` zu reduzieren und Zugriffsmethoden (`get()`- und `set()`-Methoden) anzubieten. Diese werden im OO-Sprachjargon auch als ***Accessors*** oder ***Mutators*** bezeichnet.

Durch die Definition von Sichtbarkeitsregeln erreicht man eine Strukturierung und Kapselung: Die Klasse kann durch die Zugriffsmethoden die Möglichkeiten zum Zugriff auf Attribute selbst steuern oder einschränken. Außerdem können verschiedene Sichtbarkeiten für Methoden vergeben werden, wodurch internes Verhalten nicht nach außen sichtbar wird, sondern nur gewünschte verhaltendsdefinierende Methoden, wie die nachfolgend beschriebenen Business-Methoden.

---

[6] Die Bedeutung von `public` ändert sich mit Java 9 und im Kontext der Modularisierung. Darauf gehe ich in Abschnitt 15.3 ein.

[7] Man kann *diese* Sichtbarkeit durch einen Kommentar der Form `/*private*/` bzw. `/*package*/` andeuten und so zeigen, dass dies bewusst gewählt wurde.

Objektverhalten und Business-Methoden Das Verhalten der Instanzen einer Klasse ist durch die bereitgestellten Methoden definiert. Diese Methoden weisen in der Regel unterschiedliche Abstraktionsgrade und verschiedene Sichtbarkeiten auf. Einige Arbeits- und Hilfsmethoden verwenden viele Implementierungsdetails und arbeiten direkt mit den Attributen der Klasse. Sie sollten bevorzugt `private` oder Package-private definiert werden. Meistens realisieren nur wenige der Methoden einer Klasse »High-Level-Operationen« mit einem hohen Abstraktionsgrad. Diese verhaltensdefinierenden Methoden bilden in der Regel die Schnittstelle der Klasse nach außen. Derartige Methoden nennt man auch ***Business-Methoden***. Sie verstecken die komplexen internen Vorgänge. Auf diese Weise erreicht man folgende Dinge:

- **Abstraktion** und **Datenkapselung** – Implementierungsdetails werden versteckt.
- **Klare Abhängigkeiten** – Wenn Zugriffe durch andere Klassen nur über die Business-Methoden erfolgen, existieren wenige und klare Abhängigkeiten.
- **Austauschbarkeit** und **Wiederverwendbarkeit** – Werden die Business-Methoden durch ein Interface beschrieben, so kann ein Austausch der Realisierung (im besten Fall) sogar ohne Rückwirkung auf Nutzer erfolgen.

Tipp: Aufrufhierarchien von Methoden

Um den Sourcecode übersichtlich zu halten, sollten sich öffentliche Methoden aus Methodenaufrufen anderer Sichtbarkeiten zusammensetzen und wenig Implementierungsdetails zeigen. Sofern Methoden keine Methoden höherer Sichtbarkeit aufrufen (also etwa eine private Methode keine öffentliche Methode aufruft), erreicht man eine Aufrufhierarchie und eine Art klasseninterne Schichtenarchitektur. Diese Strukturierung hilft insbesondere dann, wenn man Datenänderungen verarbeiten und an andere Klassen kommunizieren muss. Ohne die Einhaltung dieses Hinweises kommt es schneller zu einem Chaos.

Kohäsion Beim objektorientierten Programmieren verstehen wir unter ***Kohäsion*** den inneren Zusammenhang, also in welchem Maße eine Klasse tatsächlich genau eine Aufgabe oder einen Aufgabenbereich erfüllt. Je höher die Kohäsion, desto besser realisiert eine Klasse lediglich eine spezielle Funktionalität. Bei hoher Kohäsion erfolgt eine gute Trennung von Zuständigkeiten. Klassen mit hoher Kohäsion können normalerweise gut kombiniert werden, um neue Funktionalitäten zu realisieren. Dies hilft bei der Wiederverwendbarkeit. Je niedriger auf der anderen Seite die Kohäsion ist, desto mehr wird unterschiedliche Funktionalität innerhalb einer Klasse realisiert. Analog können wir von Kohäsion einer Methode reden. Sie beschreibt, wie genau abgegrenzt die Funktionalität der Methode ist.

Info: Orthogonalität und Wiederverwendung

In der Informatik spricht man von ***Orthogonalität***, wenn man eine ***freie Kombinierbarkeit*** unabhängiger Konzepte – hier Methoden und Klassen – erreicht. Eine derartige Implementierung zu erstellen, erfordert allerdings einiges an Erfahrung. Häufig sind Methoden und Klassen daher in der Praxis eben leider nicht immer so gestaltet, dass sie nur genau eine Aufgabe erfüllen. Wenn man dann eine benötigte Teilfunktionalität einer Methode oder Klasse verwenden möchte, aber nur diese eine Aufgabe in Form eines Methodenaufrufs nicht bekommen kann, werden als Abhilfe mitunter die entsprechenden Zeilen einfach kopiert, statt eine Methode oder Klasse mit der gewünschten Funktionalität herauszulösen. Eine derartige Sourcecode-Duplikation (Copy-Paste-Wiederverwendung) führt häufig zu einer schwachen Kohäsion und sollte daher möglichst vermieden werden.[a]

[a] Wird eine ganze Methode in eine andere Klasse kopiert, kann sie durchaus für sich eine hohe Kohäsion aufweisen. Allerdings wird die Kohäsion auf Klassenebene dadurch geringer, weil nun zwei Klassen dasselbe tun (können). Das ist wartungsunfreundlich.

Assoziationen Stehen Objekte in Beziehung zueinander, so spricht man ganz allgemein von einer ***Assoziation***. Diese Beziehung kann nach der Zusammenarbeit wieder aufgelöst und es können neue Assoziationen zu anderen Objekten aufgebaut werden.

Man untergliedert Assoziationen in Aggregation und Komposition. Eine ***Aggregation*** beschreibt eine ***Ganzes-Teile-Beziehung***. Insbesondere sind die Objekte nicht mehr gleichwertig, sie können aber unabhängig voneinander existieren. Bei einer ***Komposition*** ist die Ganzes-Teile-Beziehung noch stärker ausgeprägt: Ein enthaltenes Teilobjekt kann ohne das Ganze nicht selbstständig existieren. Ganzes-Teile-Beziehungen findet man häufig in dem Zusammenhang, dass ein Objekt, also das Ganze, eine Menge von anderen Objekten, die Teile, referenziert. Je nach Modellierungsabsicht stellt die in Abbildung 3-3 dargestellte Beziehung zwischen einer Musik-CD und einigen MP3-Liedern eine Aggregation bzw. eine Komposition dar. Wenn man dies als Aggregation betrachtet, so existieren die MP3-Lieder unabhängig v on der CD, z. B. als Dateien einer Playlist. Bei der Komposition sind die MP3-Lieder Bestandteil der CD in Form von Bits und Bytes auf dem Datenträger selbst.

Abbildung 3-3 *Aggregation / Komposition*

Kopplung Unter ***Kopplung*** verstehen wir, wie stark Klassen miteinander in Verbindung stehen, also den Grad ihrer Abhängigkeiten untereinander. Einen großen Einfluss hat, welche Attribute und Methoden sichtbar und zugreifbar sind und wie dieser Zugriff erfolgt. Zwei Klassen sind stark miteinander gekoppelt, wenn entweder viele feingranulare Methodenaufrufe der jeweils anderen Klasse erfolgen oder aber auf Attribute der anderen Klasse direkt zugegriffen wird. Viele Methodenzugriffe zur Veränderung des Objektzustands deuten auf einen Mangel an Kapselung hin. Bei Unachtsamkeit pflanzt sich dieser fort und führt dazu, dass jede Klasse viele Details anderer Klassen kennt und mit diesen eng verbunden (stark gekoppelt) ist. Man kann dann von »Objekt-Spaghetti« sprechen. Häufig ist starke Kopplung durch mangelnde Trennung von Zuständigkeiten, also eine niedrige Kohäsion, begründet.

Ziel einer Modellierung ist es, eine möglichst lose Kopplung und damit geringe Abhängigkeiten verschiedener Klassen untereinander zu erreichen. Durch eine gute Datenkapselung sowie die Definition von Objektverhalten in Form von Business-Methoden erreicht man dies: Bei hoher Kohäsion gibt es klare Zuständigkeiten, wodurch Klassen möglichst eigenständig und unabhängig von anderen werden. Mithilfe einer guten Kapselung wird dieser Effekt dahingehend verstärkt, dass weniger Realisierungsdetails für andere Klassen sichtbar sind. Als Folge kann eine kleine Schnittstelle mit wenigen Methoden zur Kommunikation mit anderen Klassen definiert werden.

Tipp: Vorteile loser Kopplung und hoher Kohäsion

Durch den Einsatz von loser Kopplung lassen sich nachträgliche Änderungen meistens einfacher und ohne größere Auswirkungen für andere Klassen umsetzen. Eine starke Kopplung führt dagegen oft dazu, dass Änderungen in vielen Klassen notwendig werden. ***Gutes OO-Design besteht darin, jede Klasse so zu gestalten, dass deren Aufgabe klar abgegrenzt ist. Zudem sollte jede Aufgabe nur durch eine Klasse realisiert werden und die Abhängigkeiten zwischen Klassen möglichst minimal sein.***

Vererbung Eine spezielle Art, neue Klassen basierend auf bestehenden Klassen zu definieren, nennt sich ***Vererbung***. Diese wird durch das Schlüsselwort `extends` ausgedrückt: Die neu entstehende Klasse erweitert oder übernimmt (erbt) durch diesen Vorgang das Verhalten und die Eigenschaften der bestehenden Klasse und wird ***abgeleitete*** oder ***Sub***klasse genannt. Die wiederverwendete Klasse bezeichnet man als ***Basis***-, ***Ober***- oder ***Super***klasse. Eine Subklasse muss dann in ihrer Implementierung lediglich die Unterschiede zu ihrer Basisklasse beschreiben und nicht komplett neu entwickelt werden. Vererbung ermöglicht also die ***Wiederverwendung*** bereits existierender Funktionalität und erleichtert die Wartung, da bei einer Fehlersuche und -korrektur weniger Sourcecode zu schreiben, zu pflegen und zu analysieren ist. Das ist ein Vorteil gegenüber dem sogenannten ***Copy-Paste-Ansatz***. Darunter versteht man, dass Wiederverwendung dadurch erreicht wird, dass Teile des Sourcecodes (im Extremfall manchmal sogar ganze Klassen) kopiert und entsprechend je nach Bedarf modifiziert werden.

Bereits anhand dieser Beschreibung erahnt man, dass beim Copy-Paste-Ansatz im Rahmen von Erweiterungen oder Korrekturen oftmals Änderungen recht verstreut in vielen Klassen durchzuführen sind. Beim Einsatz von Vererbung und der Extraktion bzw. der Definition passender Basis- und Subklassen sind Änderungen lediglich an wenigen Stellen, im Idealfall sogar nur einer Stelle, notwendig.

Durch Vererbung entsteht eine ***Klassenhierarchie***. Wenn man diese gedanklich in Richtung Subklassen durchläuft, spricht man von einer ***Spezialisierung***, und der Weg in Richtung Basisklassen wird ***Generalisierung*** genannt (vgl. Abbildung 3-4).

Abbildung 3-4 *Generalisierung und Spezialisierung*

Vererbung sollte allerdings nur dann eingesetzt werden, wenn die sogenannte ***»is-a«-Beziehung*** erfüllt ist. Diese besagt, dass ***Subklassen tatsächlich eine semantische Spezialisierung ihrer Basisklasse darstellen*** und dadurch das Verhalten sowie die Eigenschaften der Basisklasse besitzen. Wenn statt einer Basisklasse überall auch eine Spezialisierung davon eingesetzt werden kann, dann spricht man von ***Substituierbarkeit***. Das beschreibt das ***Substitutionsprinzip*** und insbesondere das LISKOV SUBSTITUTION PRINCIPLE (LSP). Dieses lernen wir in Abschnitt 3.5.3 kennen und erfahren dort, dass die Einhaltung der »is-a«-Beziehung bereits eine wesentliche Grundlage ist.

Wird Vererbung wirklich nur dann eingesetzt, wenn tatsächlich die ***»is-a«-Beziehung*** erfüllt ist, so profitiert man davon, dass beim Aufbau einer ***Klassenhierarchie*** durch Ableitung lediglich die Unterschiede zum vererbten Verhalten und Zustand definiert werden müssen. Erweiterungen werden durch neue Methoden und Attribute realisiert. Für Änderungen am bestehenden Verhalten müssen die abgeleiteten Klassen in der Lage sein, die Methoden der Basisklasse zu verändern. Dies wird durch die Technik ***Overriding*** erreicht. Ganz wichtig ist die Abgrenzung zum sogenannten ***Overloading***. Damit ist gemeint, dass mehrere Methoden mit gleichem Namen, aber unterschiedlicher Signatur innerhalb einer Klasse definiert sind. Da selbst erfahrene Entwickler die Techniken Overriding und Overloading gelegentlich nicht immer richtig einsetzen, möchte ich diese im Anschluss detaillierter beschreiben.

Wird die Forderung nach semantischer Spezialisierung nicht beachtet, sondern Vererbung dazu eingesetzt, um aus der Basisklasse benötigte Funktionalität zu übernehmen, dann handelt es sich um eine sogenannte ***Implementierungsvererbung***. Diese ist zu vermeiden, weil damit zwar technisch, aber nicht semantisch ein Subtyp definiert

wird. Als Konsequenz kann man Objekte einer Subklasse konzeptuell nicht mehr als Objekte der Basisklasse betrachten. Abbildung 3-5 zeigt ein Positiv- und ein Negativbeispiel.

Abbildung 3-5 *Vererbung und »is-a«-Beziehung*

Overriding Mit Überschreiben bzw. ***Overriding*** ist das ***Redefinieren von geerbten Methoden*** gemeint. Dazu wird der Methodenname übernommen und in der Subklasse eine neue Implementierung der Methode bereitgestellt. Dadurch können Subklassen Methoden modifizieren oder sogar komplett anders definieren, um Änderungen im Verhalten auszudrücken. Dabei darf weder die Signatur der Methode modifiziert noch die Sichtbarkeit eingeschränkt werden – eine Erweiterung ist dagegen erlaubt.

Overloading Unter Überladen oder ***Overloading*** versteht man die Definition von Methoden gleichen Namens innerhalb der gleichen Klasse, aber mit unterschiedlicher Parameterliste. Im Gegensatz zum Overriding besteht demnach kein Zusammenhang mit Vererbung, sondern es wird eine ***Vereinfachung der Schreibweise*** adressiert. Durch Overloading kann man den Namen von Methoden ähnlicher Intention vereinheitlichen. Betrachten wir als Beispiel eine Klasse `Rectangle` mit folgenden Methoden:

```
drawByStartAndEndPos(int x1, int y1, int x2, int y2)}
drawByStartPosAndSize(Point pos, Dimension size)}
```

Die Handhabung der Klasse wird vereinfacht, wenn man sich als Anwender nur einen Methodennamen zum Zeichnen merken muss. In diesem Fall könnte man folglich den Namen auf `draw()` verkürzen und zwei Methoden mit unterschiedlichen Parameterlisten anbieten. Die JVM wählt beim Aufruf automatisch aufgrund der übergebenen Parameter bzw. deren Anzahl oder Typen die passende Methode.

So praktisch das Ganze auch manchmal ist, so sind dem Overloading doch Grenzen gesetzt: Methoden müssen unterschiedliche Parameterlisten besitzen, damit Overloading möglich ist. Für die folgenden beiden Methoden gilt das nicht und somit ist hier kein Overloading erlaubt:

```
drawByStartAndEndPos(int x1, int y1, int x2, int y2)
drawByStartPosAndSize(int x1, int y1, int width, int height)
```

Sub-Classing und Sub-Typing Spezialisierung ist sowohl zwischen Klassen als auch zwischen Interfaces möglich. Beides wird durch das Schlüsselwort `extends` ausgedrückt. Zwischen Klassen wird durch Spezialisierung ein Vererben von Verhalten erreicht, d. h., eine Klasse ist eine spezielle Ausprägung einer anderen Klasse, übernimmt deren Verhalten und kann eigene Merkmale und Verhaltensweisen hinzufügen. Hier spricht man von ***Sub-Classing***. Eine Spezialisierung eines Interface erweitert die Menge der Methoden eines anderen Interface. In diesem Fall spricht man von ***Sub-Typing***. Bei Klassen spricht man häufiger der Einfachheit halber auch von Sub-Typing. Dies erleichtert die Diskussion, denn eine Vererbung zwischen Klassen ist streng genommen sowohl Sub-Classing als auch Sub-Typing. Das Implementieren eines Interface ist jedoch nur Sub-Typing.

Polymorphie Variablen eines Basistyps können beliebige davon abgeleitete Spezialisierungen referenzieren. Das wird als Vielgestaltigkeit oder ***Polymorphie*** bezeichnet. Polymorphie basiert auf dem Unterschied zwischen dem ***Kompiliertyp*** und dem ***Laufzeittyp***. Der Kompiliertyp ist der zur Kompilierzeit bekannte Basistyp. Der Laufzeittyp entspricht der konkret verwendeten Spezialisierung. Betrachten wir dazu eine Klassenhierarchie der Klassen `Base`, `Sub` und `SubSub` sowie deren Methoden `doIt()` und `doThat()`. Die Klassenhierarchie ist in Abbildung 3-6 dargestellt.

Abbildung 3-6 *Polymorphie und dynamisches Binden*

Ein Beispiel für Polymorphie ist, dass die Variable `base` den Kompiliertyp `Base` besitzt und während der Programmausführung den Laufzeittyp `SubSub`. Damit Polymorphie funktioniert, muss immer die spezialisierteste Methode eines Objekts verwendet werden, d. h. die Implementierung der spezialisiertesten Subklasse, die diese Methode anbietet: Zur Bestimmung der auszuführenden Methode wird dazu startend bei dem Laufzeittyp nach einer passenden Methode gesucht. Für den Aufruf von `doIt()` startet die Suche daher in der Klasse `SubSub`. Dort wird die JVM allerdings nicht fündig, sodass die Suche sukzessive weiter nach oben in der Vererbungshierarchie fortgesetzt wird, bis

eine Methodendefinition gefunden wird. In diesem Beispiel ist dies für `doIt()` in der Klasse `Sub` der Fall. Das beschriebene Verfahren zum Auffinden der auszuführenden Methode wird ***dynamisches Binden*** (***Dynamic Binding***) genannt.

Achtung: Polymorphie – Einfluss von Kompilier- und Laufzeittyp

Eine Quelle für Flüchtigkeitsfehler ist die Annahme, dass die polymorphen Eigenschaften von Objektmethoden auch für Attribute und statische Methoden gelten würden. Attribute und statische Methoden sind nicht polymorph: Es wird immer auf die durch den Kompiliertyp sichtbaren Methoden bzw. Attribute zugegriffen. Dieses wird leicht übersehen und kann zu falschem Systemverhalten führen.

Explizite Typumwandlung – Type Cast (Cast) Unter einer expliziten Typumwandlung, einem sogenannten ***Type Cast*** oder kurz ***Cast***, versteht man eine »Aufforderung« an den Compiler, eine Variable eines Typs in einen anderen angegebenen Typ umzuwandeln. Solange man dabei in der Typhierarchie nach oben geht, ist diese Umwandlung durch die »is-a«-Beziehung abgesichert. Soll jedoch ein Basistyp in einen spezielleren Typ konvertiert werden, so geht man in der Ableitungshierarchie nach unten. Eine derartige Umwandlung wird als ***Down Cast*** bezeichnet und ist durch die »is-a«-Beziehung nicht abgesichert. Betrachten wir dies an einem Beispiel für eine Methode `doSomethingWithSub(Object)`, die einen Cast vornimmt und mit einer Instanz vom Typ `Sub` bzw. `String` aufgerufen wird:

```
// Dieser Aufruf ist problemlos möglich
doSomethingWithSub(new Sub());

// Dieser Aufruf mit einem String führt zu einem Fehler
doSomethingWithSub("This will fail");

// ...

// Diese Methode sollte man mit Referenzen vom Typ Sub aufrufen
void doSomethingWithSub(final Object obj)
{
    final Sub sub = (Sub) obj; // Unsicherer Down Cast: Object -> Base -> Sub
    // ...
}
```

Nur für den Fall, dass die übergebene Referenz `obj` ein Objekt vom Typ `Sub` (oder eine Spezialisierung davon) enthält, ist der Cast erfolgreich. Ansonsten kommt es zu einer Inkompatibilität von Typen, wodurch eine `java.lang.ClassCastException` ausgelöst wird. Dies ist für den zweiten Methodenaufruf des Beispiels der Fall.

Solche Down Casts sollte man daher möglichst vermeiden oder zumindest durch eine explizite Typprüfung mit `instanceof` versehen:[8]

[8] Mehrfache `instanceof`-Prüfungen deuten auf eine Verletzung des sogenannten OPEN CLOSED PRINCIPLE (OCP) (vgl. Abschnitt 3.5.3) hin.

```
if (obj instanceof Sub)    // Absicherung des Down Cast: Object -> Base -> Sub
{
    final Sub sub = (Sub) obj;
    // ...
}
else
{
    //  Problem: Wie soll man auf diese Situation reagieren?
}
```

Mit `instanceof` kann eine Umwandlung abgesichert werden, es stellt sich dann aber die Frage, wie auf alle nicht erwarteten Typen reagiert werden sollte. Mitunter kann sinnvollerweise nur eine Fehlermeldung ausgegeben werden.

Parameterübergabe per Call-by-Value / Call-by-Reference Beim Aufruf von Methoden existieren verschiedene Strategien, wie die Werte von Parametern übergeben werden. Man unterscheidet zwischen der Übergabe eines Werts (***Call-by-Value***) und der Übergabe per Referenz (***Call-by-Reference***). Entgegen der weitverbreiteten Meinung, in Java würden beide Arten der Parameterübergabe unterstützt, und zwar Call-by-Value für Parameter primitiver Typen und Call-by-Reference für Referenzparameter, wird tatsächlich in beiden Fällen Call-by-Value genutzt.

Betrachten wir diese in Java umgesetzte Form und ihre Auswirkungen genauer: Alle primitiven Typen werden in Java als Wert übergeben, genauer als bitweise Kopie des Werts. Dadurch existiert keine Verbindung zwischen dem Wert außerhalb der Methode und dem Wert des Parameters. Deshalb sind Änderungen an den Parameterwerten nur lokal innerhalb der Methode sichtbar, nicht jedoch für aufrufende Methoden.

Bei der Übergabe von Referenzvariablen als Parameter könnte man intuitiv auf die Idee kommen, die Übergabe würde per Referenz erfolgen, wie dies etwa bei C++ möglich ist. Bei Call-by-Reference sind in C++ alle Änderungen am Wert eines derart übergebenen Parameters, die innerhalb der Methode stattfinden, auch außerhalb sichtbar. In Java gilt dies allerdings nicht! Auch die Übergabe von Referenzvariablen als Parameter erfolgt als Kopie – allerdings wird hierbei eine bitweise Kopie des Werts der Referenzvariablen erstellt und nicht eine Kopie des referenzierten Objekts. Das hat folgende Konsequenzen: Innerhalb der aufgerufenen Methode kann man zwar das durch die Referenzvariable referenzierte Objekt durch ein anderes ersetzen, diese Änderung ist aber für die aufrufende Methode nicht sichtbar: Die Referenzvariable verweist nach Aufruf der Methoden auf dasselbe Objekt wie zuvor. ***Allerdings ist es möglich, in der aufgerufenen Methode den Zustand des referenzierten Objekts zu verändern. Dadurch können aber Änderungen am Zustand des referenzierten Objekts auch außerhalb der Methode sichtbar sein*** – ähnlich wie man dies für Call-by-Reference kennt. Man spricht hier von der ***Referenzsemantik*** von Java.

Schauen wir uns zur Klärung die Methode `changeSomething(String[])` an. Diese erhält ein Array von Strings. Zwar kann die Referenz auf das Array in der Methode nicht verändert werden, sehr wohl aber dessen Inhalt:

```
private static void changeSomething(String[] names)
{
    // Änderungen auch im Original-Array
    names[0] = "Michael";
    names[1] = "changed this entry";

    // Keine Auswirkung auf das Original-Array
    names = new String[] { "Nearly Empty" };
}

public static void main(final String[] args)
{
    final String[] names = { "Test1", "Test2", "!!!" };
    changeSomething(names);
    System.out.println(Arrays.toString(names));
}
```

Listing 3.1 *Ausführbar als* '**REFERENCESEMANTICSEXAMPLE**'

Führen wir das Programm REFERENCESEMANTICSEXAMPLE aus, so werden die ersten beiden Einträge des Arrays geändert. Damit ergibt sich folgende Ausgabe:

```
[Michael, changed this entry, !!!]
```

Außerdem wird durch diese Ausgabe offensichtlich, dass die Zuweisung der Referenz innerhalb der Methode sich nicht in der `main()`-Methode auswirkt, Wertänderungen einzelner Elemente dagegen schon. Das gilt gleichfalls für Datenstrukturen aus dem Collections-Framework, etwa für Listen und Mengen. Dieser Sachverhalt sollte einem immer bewusst sein, da sich ansonsten leicht Fehler einschleichen, die schwierig zu finden sind. Call-by-Value bezieht sich in Java wirklich ausschließlich auf den Wert der Referenz bzw. des primitiven Typs. Auf weitere Auswirkungen der Referenzsemantik werde ich insbesondere in Abschnitt 3.4.1 detaillierter eingehen.

3.1.2 Beispielentwurf: Ein Zähler

Nachdem nun die Grundbegriffe beim objektorientierten Entwurf bekannt sind, werden diese im Folgenden anhand eines Beispiels vertieft. Eine grafische Anwendung soll um eine Auswertung der Anzahl gezeichneter Linien und Rechtecke erweitert werden. Dazu ist ein Zähler als Klasse zu entwerfen, der folgende Anforderungen erfüllt:

1. Er lässt sich auf den Wert 0 zurücksetzen.
2. Er lässt sich um eins erhöhen.
3. Der aktuelle Wert lässt sich abfragen.

Mit diesen scheinbar einfach umzusetzenden Anforderungen eines Kollegen »User« lassen wir die zwei exemplarischen Entwickler »Schnell-Finger« und »Überleg-Erst-Einmal« diese Aufgabe realisieren. Schauen wir uns an, wie die beiden arbeiten.

Bevor wir beiden Entwicklern bei der Arbeit zusehen, betrachten wir den Nutzungskontext. Der Anwendungscode ist bereits folgendermaßen rudimentär und nicht besonders elegant mit einem `instanceof`-Vergleich implementiert:

```
// TODO: 2 Zähler initialisieren

for (final GraphicObject graphicObject : graphicObjects)
{
   graphicObject.draw();

   if (graphicObject instanceof Line)
   {
        // TODO: lineCounter erhöhen
   }
   if (graphicObject instanceof Rect)
   {
        // TODO: rectCounter erhöhen
   }
}

// TODO: Zähler auslesen und ausgeben
```

Beim Zeichnen soll abhängig vom Typ der grafischen Objekte ein korrespondierender Zähler inkrementiert werden. Die eigentliche Funktionalität bleibt zunächst unimplementiert, da noch auf die konkrete Realisierung der Klasse `Counter` gewartet wird. Deren Einsatzstellen sind mit `TODO`-Kommentaren markiert.

Hinweis: Anmerkungen zu `instanceof`-Vergleichen

`instanceof`-Vergleiche sind vielfach ein Anzeichen schlechten Programmierstils. Meistens liegt ein Verstoß gegen das sogenannte OPEN CLOSED PRINCIPLE (OCP) vor – einem der SOLID-Prinzipien (siehe Abschnitt 3.5.3). Diesen Nachteil wollen wir für dieses Beispiel akzeptieren, da hier der Fokus auf dem objektorientierten Entwurf eines Zählers liegt.

Entwurf à la »Schnell-Finger«

Herr »Schnell-Finger« überlegt nicht lange und startet sofort seinen Texteditor. Dabei hat er bereits ein paar Ideen. Kaum ist das Editor-Fenster geöffnet, legt er los. Klingt alles recht einfach, also wird kurzerhand die folgende Klasse erstellt:

```
public class Counter
{
    public int count = 0;

    public Counter()
    {
    }

    public void setCounter(int count)
    {
        count = count;
    }
}
```

So, damit ist er schon fertig. Die Variable `count` ist öffentlich, kann also von überall abgefragt werden. Zum Verändern dient außerdem die Methode `setCounter(int)`.

Das Rücksetzen erfolgt durch Übergabe von 0. Zufrieden speichert er die Klasse. Am nächsten Tag fragt der Kollege »User«, wie weit die Klasse `Counter` wäre, und bekommt die Antwort: »Längst fertig!« Der Kollege »User« schaut sich die Klasse an und sagt: »Na ja, richtig objektorientiert ist das nicht. Es fehlt an Kapselung: Du veröffentlichst dein Attribut und bietest Zugriffsmethoden nur unvollständig an und der Defaultkonstruktor ist auch überflüssig.« Etwas verärgert über den pingeligen Kollegen macht sich Herr »Schnell-Finger« nochmals an die Arbeit und kommt nach kurzer Zeit zu folgender Umsetzung:

```
public class Counter
{
    private int count = 0;

    public int getCounter()
    {
        return count;
    }

    public void setCounter(int count)
    {
        count = count;
    }
}
```

Die Klasse `Counter` ist laut Herrn »Schnell-Finger« nun einsatzbereit. Daher wollen wir diese zum Zählen von grafischen Figuren nutzen und fügen sie in das zuvor vorgestellte Anwendungsgerüst an den durch die `TODO`-Kommentare markierten Stellen ein. Dadurch entsteht folgender Sourcecode:

```
// 2 Zähler initialisieren
final Counter lineCounter = new Counter();
final Counter rectCounter = new Counter();

for (final GraphicObject graphicObject : graphicObjects)
{
    graphicObject.draw();

    if (graphicObject instanceof Line)
    {
        // lineCounter erhöhen
        lineCounter.setCounter(lineCounter.getCounter() + 1);
    }

    if (graphicObject instanceof Rect)
    {
        // rectCounter erhöhen
        rectCounter.setCounter(rectCounter.getCounter() + 1);
    }
}

// Zähler auslesen und ausgeben
System.out.println("Number of Lines: " + lineCounter.getCounter());
System.out.println("Number of Rects: " + rectCounter.getCounter());
```

Wirklich zufrieden ist der Kollege »User« mit der Klasse `Counter` nicht. Der Sourcecode sieht irgendwie nicht rund aus. Und tatsächlich kommt es bei einem Test trotz vieler gemalter Linien und Rechtecke zu folgenden unerwarteten Ausgaben:

```
Number of Lines: 0
Number of Rects: 0
```

Der nutzende Anwendungscode scheint fehlerfrei. In der Implementierung der Methode `setCounter(int)` steckt jedoch ein Flüchtigkeitsfehler: Der Übergabeparameter heißt genauso wie das Attribut und ist nicht `final`. Dadurch erfolgt unbemerkt eine Zuweisung an sich selbst. Das lässt sich leicht wie folgt korrigieren:

```
public void setCounter(final int newCount)
{
    this.count = newCount;
}
```

Entwurf à la »Überleg-Erst-Einmal«

Schauen wir auf die Variante von Kollege »Überleg-Erst-Einmal«. Er liest zunächst die Anforderungen und skizziert ein kleines UML-Klassendiagramm von Hand auf einem Blatt Papier, wie in Abbildung 3-7 gezeigt.

Abbildung 3-7 *Die Klasse* `Counter`

Sein Entwurf sieht zum Verarbeiten des Zählers die Methoden `getCounter()` und `setCounter()` vor. Zum Rücksetzen des Werts dient die Methode `reset()`. Da er bereits einige Erfahrung im Softwareentwurf hat, weiß er, dass die erste Lösung meistens nicht die beste ist. Um seinen Entwurf zu prüfen, überlegt er sich ein paar Anwendungsfälle und spielt diese im Kopf durch. Später kann er daraus Testfälle definieren. Zudem schaut er nochmals auf die gewünschten Anforderungen. Dort steht: Der Zähler soll um eins erhöht werden. Folglich entfernt er die `setCounter()`-Methode und führt eine `increment()`-Methode ein. Abbildung 3-8 zeigt das Ergebnis.

Abbildung 3-8 *Die Klasse* `Counter`*, 2. Version*

Als unbefriedigend empfindet er aber noch eine Inkonsistenz in der Namensgebung bezüglich des Methodennamens `getCounter()`. Der Name ist missverständlich: Hier wird kein `Counter`-Objekt zurückgeliefert, sondern dessen Wert. Namen wie z. B. `getValue()` bzw. `getCurrentValue()` wären damit aussagekräftiger und verständlicher. Eine weitere Alternative ist, auf das Präfix `get` im Namen zu verzichten. Damit ergeben sich `value()` und `currentValue()` als mögliche Methodennamen. In diesem Fall entscheidet er sich für letztere Variante, da sich die Namen dann besser lesen lassen (vgl. Abbildung 3-9). Der Einsatz des Präfixes `get` würde allerdings den lesenden Charakter der Zugriffsmethode auf einen Blick sichtbar machen.[9]

Abbildung 3-9 *Die Klasse* `Counter`*, 3. Version*

Noch während Herr »Überleg-Erst-Einmal« implementiert, kommt der Kollege »User« vorbei und berichtet, dass er die Lösung von Herrn »Schnell-Finger« als zu sperrig empfindet. Daher würde er gern eine weitere Lösung begutachten und in seine Anwendung einbauen. Die Implementierung der Klasse `Counter` ist zwar noch nicht ganz abgeschlossen. Deren Schnittstelle, das sogenannte ***API*** (***Application Programming Interface***), d. h. die angebotenen Methoden, sind durch das UML-Diagramm aber bereits definiert. Daher kann der Applikationscode wie folgt auf das API der Klasse `Counter` von Herrn »Überleg-Erst-Einmal« angepasst werden:

```
final Counter lineCounter = new Counter();
final Counter rectCounter = new Counter();

for (final GraphicObject graphicObject : graphicObjects)
{
    graphicObject.draw();

    if (graphicObject instanceof Line)
    {
        lineCounter.increment();
    }

    if (graphicObject instanceof Rect)
    {
        rectCounter.increment();
    }
}

System.out.println("Number of Lines: " + lineCounter.currentValue());
System.out.println("Number of Rects: " + rectCounter.currentValue());
```

[9] Beide Varianten der Namensgebung sind sinnvoll, und die konkrete Wahl ist eine Frage des Geschmacks (oder aber durch sogenannte Coding Conventions (vgl. Kapitel 19) festgelegt).

Der resultierende Sourcecode ist deutlich besser lesbar. Die Zeilen mit dem Aufruf der Methode `increment()` sind nun selbsterklärend, und die Kommentare konnten daher ersatzlos entfallen. Schauen wir abschließend auf den von Kollege »Überleg-Erst-Einmal« erstellten Sourcecode seiner Variante der Klasse `Counter`:

```
public class Counter
{
    private int value = 0;

    public int currentValue()
    {
        return value;
    }

    public void increment()
    {
        value++;
    }

    public void reset()
    {
        value = 0;
    }
}
```

Vergleich der beiden Lösungen

Nachdem wir die zwei verschiedenen Arten der Entwicklung exemplarisch kennengelernt haben, wollen wir die erreichten Lösungen vergleichen und so weitere Erkenntnisse zum objektorientierten Programmieren gewinnen bzw. vertiefen. Wir betrachten dazu die beiden Implementierungen bezüglich ihres Grads an Objektorientierung sowie Erweiter- und Wiederverwendbarkeit.

Grad der Objektorientierung Die Lösung von Herrn »Schnell-Finger« hat durch einige Iterationen und vor allem Review-Kommentare von Kollegen einen brauchbaren Zustand erreicht. Allerdings macht diese Lösung ausschließlich von `get()`-/`set()`-Methoden Gebrauch, ohne aber Verhalten zu definieren. Bei ersten OO-Gehversuchen besteht die Gefahr, dass nahezu alle Methoden zu derartigen reinen Zugriffsmethoden ohne Objektverhalten verkümmern. Hier herrscht meiner Ansicht nach das größte Missverständnis bezüglich der Objektorientierung vor: ***Ein intensiver Gebrauch von `get()`- und `set()`-Methoden ist kein Zeichen von guter Objektorientierung, sondern deutet vielmehr auf einen fragwürdigen OO-Entwurf hin***. Solche Klassen tendieren dazu, kein Verhalten mehr zu kapseln – stattdessen wird dieses stark von nutzenden Klassen oder Methoden gesteuert bzw. in diesen realisiert. Dadurch besteht die Gefahr, dass Attribute derart von außen manipuliert werden, dass sich der Objektzustand in einer unerwarteten (möglicherweise ungewünschten) Art und Weise verändert. Dies ließ sich im Zählerbeispiel von Herrn »Schnell-Finger« sehr schön daran sehen, wie die Inkrementierung des Zählers realisiert ist. Tatsächlich könnte der Zähler durch die angebotene `setCounter(int)`-Methode auf beliebige Werte gesetzt werden, also erhöht

oder erniedrigt werden. Das Inkrement wird also im Applikationscode programmiert. Jede weitere Applikation müsste wiederum diese Funktionalität des Zählers selbst realisieren. Diese Implementierung ist demnach vollständig an der Anforderung vorbei entwickelt, da lediglich ein besserer Datencontainer realisiert wird. Kurz gesagt: Die Klasse hat keine Business-Methoden. Damit verletzt diese Art des Entwurfs zum einen den Gedanken der Definition von Verhalten durch Objekte und zum anderen das Ziel der Wiederverwendbarkeit. Beim `get()`-/`set()`-Ansatz lässt sich somit oftmals keine (oder zumindest kaum) Funktionalität wiederverwenden.

Die Klasse `Counter` von Herrn »Überleg-Erst-Einmal« realisiert dagegen ein logisches Modell. Hier stehen die technischen Details nicht im Vordergrund, sondern man konzentriert sich auf das Erfüllen einer Aufgabe. Design und Implementierung stellen die Funktionalität eines Zählers bereit und verhindern feingranulare Zugriffe auf interne Variablen. Dies verstärkt die Kapselung, da die Methoden `increment()` und `currentValue()` Implementierungsdetails gut verbergen. Es ist dadurch von außen unmöglich, den Zähler um beliebige Werte zu erhöhen oder zu erniedrigen. Diese Umsetzung besitzt zudem eine höhere Kohäsion und damit eine bessere Abschirmung bei anstehenden Änderungen als der `get()`-/`set()`-Ansatz.

Tipp: Logisches Modell und Business-Methoden

Durch das Realisieren eines logischen Modells wird der entstehende Sourcecode in der Regel deutlich besser verständlich und menschenlesbar, da man Konzepten und nicht Programmanweisungen folgt. Veränderungen am Objektzustand werden dabei durch eine Reihe von verhaltensdefinierenden ***Business-Methoden*** implementiert.

Grad der Wiederverwendbarkeit und Erweiterbarkeit Um die Auswirkungen der beiden Entwürfe bezüglich Wiederverwendbarkeit und Erweiterbarkeit zu untersuchen, nehmen wir an, dass wir den Zähler um einen Überlauf bei einer bestimmten Schwelle erweitern und die Anzahl der Überläufe protokollieren wollten. Dies könnte man etwa für eine Spieleapplikation dazu nutzen, um nach 100 aufgesammelten Bonuselementen ein weiteres Leben zu erhalten. Diese Erweiterung müsste bei Einsatz der Realisierung von Herrn »Schnell-Finger« von jeder Applikation selbst implementiert werden. Das kann ziemlich aufwendig werden, zu dupliziertem oder sehr ähnlichem Sourcecode führen und später in einem Wartungsalbtraum enden. Diesen Weg wollen wir nicht weiter betrachten, da wir (glücklicherweise) eine Alternative haben.

Überlegen wir also, wie wir den Zähler von Herrn »Überleg-Erst-Einmal« zur Realisierung der gewünschten Funktionalität verwenden können. Wir erinnern uns an die Möglichkeit der Vererbung und realisieren eine Klasse basierend auf der Klasse `Counter`. Soll eine Applikation nachträglich diese erweiterte Variante des Zählers einsetzen, so ist nur genau eine Stelle zu ändern, nämlich die Konstruktion des Zählers. Das besitzt zudem den Vorteil, dass keine Fortpflanzungen der Änderungen in die nutzenden Applikationen stattfinden – natürlich abgesehen von denjenigen, die die zusätzliche Funktionalität, also den Zählerstand des Überlaufzählers, verwenden.

Realisierung eines Zählers mit Überlauf

Wir implementieren nun eine Klasse `CounterWithOverflow` basierend auf der Klasse `Counter`. Dazu erweitern wir die Basisklasse um eine Konstante `COUNTER_MAX`, ein zusätzliches Attribut `overflowCounter` und eine Zugriffsmethode auf dessen Wert. Das Zählen, Rücksetzen und einiges anderes delegieren wir an die Basisklasse durch Aufruf der entsprechenden Methoden mithilfe des Schlüsselworts `super`. Lediglich die `increment()`-Methode muss selbst realisiert werden. Dort prüfen wir auf einen Überlauf und führen gegebenenfalls ein Rücksetzen des Zählers und eine Erhöhung des Überlaufzählers durch. Das Ganze kann man wie folgt realisieren:

```
public class CounterWithOverflow extends Counter
{
    private static final int COUNTER_MAX = 100;

    private final Counter overflowCounter = new Counter();

    public int overflowCount()
    {
        return overflowCounter.currentValue();
    }

    public void reset()
    {
        super.reset();
        overflowCounter.reset();
    }

    public void increment()
    {
        if (currentValue() == COUNTER_MAX - 1)
        {
            super.reset();
            overflowCounter.increment();
        }
        else
        {
            super.increment();
        }
    }
}
```

Tatsächlich hatte ich diese Klasse zunächst mit einem Überlaufzähler als `int`-Wert realisiert. Viel natürlicher und objektorientierter ist aber die gezeigte Variante, die Überläufe wiederum mithilfe der Klasse `Counter` zählt.

Fazit

Anhand dieses Beispiels lässt sich gut erkennen, dass neben dem puren Einsatz von Klassen und Objekten vor allem auch die Definition von Verhalten und Zuständigkeiten die objektorientierte Programmierung ausmachen. Dabei ist es hilfreich, zusammengehörende Funktionalität innerhalb einer Klasse zu bündeln (***Kohäsion***) sowie gleichzeitig Implementierungsdetails zu verbergen (***Kapselung***). In komplexeren

Entwürfen ergeben sich weitere Vorteile durch den sinnvollen Einsatz von Vererbung und die Konstruktion passender Klassenhierarchien.

Klares Ziel beim objektorientierten Entwurf sollte es sein, Klassen mit Verhalten zu definieren. Der massive Einsatz von `get()`-/`set()`-Methoden widerspricht dem und ist somit vielfach wenig objektorientiert. Dies eignet sich vor allem für Datenbehälter, also Klassen, die eigentlich kein eigenes Verhalten definieren und deren Instanzen damit eher als Hilfsobjekte betrachtet werden sollten. Diese nennt man Value Object oder auch Data Transfer Object (vgl. Abschnitt 3.4.5). Für derartige Klassen sind viele `get()`- und `set()`-Methoden nicht ungewöhnlich, allerdings sollten Value Objects keine Business-Methoden anbieten, da sie ja kein Verhalten definieren.

3.1.3 Vom imperativen zum objektorientierten Entwurf

Nachdem wir diverse OO-Grundlagen beim Entwurf einer Zähler-Klasse und dabei auch mögliche Tücken kennengelernt haben, wollen wir nun den Einfluss verschiedener Vorgehensweisen und Entwurfsstile auf die Entwicklung objektorientierter Programme betrachten.

Bevor die objektorientierte Programmierung modern wurde, hat man vielfach den imperativen Ansatz beim Softwareentwurf eingesetzt: Hierbei werden Befehle in Prozeduren und Funktionen zusammengefasst, die wiederum das Programm ergeben, das auf oftmals globalen Daten arbeitet. Funktionen und Daten existieren unabhängig voneinander. Auch wenn das Vorgehen für den imperativen Ansatz angemessen ist, entstehen mitunter merkwürdige Entwürfe, wenn man versucht, diese Vorgehensweise auf die Objektorientierung zu übertragen.

Entwurf mit verschiedenen Entwurfsstilen

Zum Vergleich verschiedener Entwurfsstile soll eine Methode `drawFigure()` zum Zeichnen von grafischen Objekten auf einer Zeichenfläche implementiert werden. Es sind verschiedene Typen grafischer Elemente (Punkt, Linie und Rechteck) darzustellen. Im Folgenden betrachten wir drei verschiedene Arten der Realisierung in Pseudocode. Das erste Beispiel implementiert die Lösung imperativ, das zweite arbeitet mit Objekten und das dritte ist schließlich objektorientiert und verdeutlicht die Vorteile der Techniken Vererbung und Polymorphie.

Imperativer Entwurf Das folgende Listing zeigt einen Entwurf, wie er von einem Kollegen kommen könnte, der in der imperativen Programmierwelt zu Hause ist. In der Klasse `PaintingArea` realisiert er eine Methode `drawFigure()`, die alle gewünschten Typen grafischer Elemente zeichnen kann:

```
public class PaintingArea extends JComponent
{
    public static final int POINT_TYPE = 0;
    public static final int LINE_TYPE  = 1;
    public static final int RECT_TYPE  = 2;

    void drawFigure(int graphicsObjectType, int x1, int y1, int x2, int y2)
    {
        if (graphicsObjectType == POINT_TYPE)
        {
            drawPoint(x1, y1);
        }
        if (graphicsObjectType == LINE_TYPE)
        {
            drawLine(x1, y1, x2, y2);
        }
        if (graphicsObjectType == RECT_TYPE)
        {
            drawRect(x1, y1, x2, y2);
        }
    }
    // ...
```

Die gezeigte Methode bekommt sämtliche zum Zeichnen der Figuren benötigten Parameter übergeben: Anhand einer `int`-Variablen `graphicsObjectType` wird geprüft, welcher Typ einer grafischen Figur gezeichnet werden soll. Je nach identifiziertem Typ wird eine für diese Art von Figur spezielle Methode mit einer Auswahl der zum Zeichnen benötigten Parameter aufgerufen. Beachten Sie bitte folgenden wesentlichen Nachteil dieses Ansatzes: Zum Zeichnen eines Punkts sind die Werte `x2` und `y2` zwar nicht nötig, müssen aber trotzdem an die `drawFigure()`-Methode übergeben werden.

Entwurf mit Objekten, aber nicht objektorientiert Ein Entwurf, wie er von einem bezüglich Objektorientierung unerfahrenen Entwickler kommen könnte, arbeitet zwar mit Objekten, nutzt jedoch nicht die Vorteile gemeinsamer Basisklassen und der Polymorphie. Positiv ist allerdings, dass die Objekte das Zeichnen mit Methoden selbst erledigen und dass auch die dazu benötigten Koordinaten als Attribute gespeichert werden. Es findet demnach eine Definition von Verhalten sowie eine Datenkapselung statt. Diese Art des Entwurfs befreit die Klasse `PaintingArea` und die `drawFigure()`-Methode von einigen Implementierungsdetails. Besonders auffällig ist dies anhand der benötigten Übergabeparameter. Es wird lediglich noch das zu zeichnende Objekt benötigt. Um die korrekte Methode zum Zeichnen aufzurufen, muss wiederum zunächst der Typ des grafischen Objekts festgestellt werden. Dies geschieht jedoch nicht anhand eines `int`-Werts, sondern basierend auf dem Laufzeittyp des jeweiligen Objekts. Dieser wird über `instanceof` geprüft und auf die entsprechenden Klassen gecastet. Dann kann die spezifische Methode zum Zeichnen aufgerufen werden:

```
void drawFigure(final Object graphicsObject)
{
    if (graphicsObject instanceof Point)
    {
        ((Point) graphicsObject).drawPoint();
    }
    if (graphicsObject instanceof Line)
    {
        ((Line) graphicsObject).drawLine();
    }
    if (graphicsObject instanceof Rect)
    {
        ((Rect) graphicsObject).drawRect();
    }
}
```

Hier erkennen wir einen »händischen Nachbau« von Polymorphie. Ein Variante davon ist, überladene Methoden zum Zeichnen anzubieten, wie dies im folgenden Praxistipp beschrieben wird.

Achtung: Künstliches Overriding und Overloading

In der Praxis kann es schnell zu Missverständnissen bezüglich der Wirkungsweise von Overriding und Overloading kommen. Zum Teil sieht man statt Overriding den Einsatz von Overloading. Es wird dann versucht, ein Verhalten wie beim Overriding nachzubilden. Dies kann unverständlich und unübersichtlich werden.

Betrachten wir das konkret am Beispiel der Klasse `PaintingArea`. Nehmen wir an, diese würde einige überladene `draw()`-Methoden zum Zeichnen von Punkten, Linien und Rechtecken anbieten. Zur Vereinfachung für Nutzer wird eine Methode `drawFigure(Object)` eingeführt. Ziel dabei ist es, ein beliebiges `Figure`-Objekt übergeben zu können und die Wahl der passenden, spezialisierten `draw()`-Methode automatisch erledigen zu lassen:

```
public void drawFigure(final Object obj)
{
    draw(obj); // Hoffnung, eine der überladenen Methoden aufzurufen
}

public void draw(final Point point)
{
    point.drawPoint();
}

public void draw(final Line line)
{
    line.drawLine();
}

// ...
```

Obwohl man intuitiv denken könnte, dass der Aufruf von `drawFigure(Object)` bei der Übergabe eines Parameters vom Typ `Line` die Methode `draw(Line)` aufruft, führt diese Zeile vielmehr zu einem Kompilierfehler: Der Compiler bemängelt das Fehlen der Methode `draw(Object)`.

Selbst wenn wir als Abhilfe eine solche Methode einführen, sind trotzdem sämtliche `draw()`-Methoden lediglich überladen. Damit ist Polymorphie hier nicht wirksam. Um dennoch die gewünschte typabhängige Auswahl der passenden Methode zu erreichen, sieht man dann teilweise die folgende Abhilfe:

```
public void draw(final Object obj)
{
    if (obj instanceof Point)
        draw((Point) obj);

    if (obj instanceof Line)
        draw((Line) obj);

    // ...
}
```

Zunächst wird eine Typprüfung vorgenommen und davon abhängig ein Cast ausgeführt. Anschließend erfolgt ein Aufruf der für diesen Typ überladenen `draw()`-Methode. Dies wirkt bereits optisch merkwürdig. Zudem ist das Design weder elegant noch flexibel erweiterbar. Einerseits blähen die `instanceof`-Prüfungen und die nachfolgenden Casts den Sourcecode auf und erschweren so die Lesbarkeit. Andererseits leidet die Erweiterbarkeit dadurch, dass während der Implementierungsphase bereits alle zu behandelnden Typen bekannt sein müssen.

Objektorientierter Entwurf Beim OO-Entwurf führen wir eine abstrakte Basisklasse `BaseFigure` ein und leiten alle grafischen Objekte von dieser ab. Die Basisklasse definiert eine abstrakte `draw()`-Methode. Alle konkreten Subklassen müssen diese implementieren und dort denn Algorithmus zum Zeichnen entsprechend ihrem Typ realisieren. Über dynamisches Binden wird von der JVM automatisch die passende Methode gewählt. Es ergibt sich folgende Implementierung:

```
public void drawFigure(final BaseFigure baseFigure)
{
    baseFigure.draw();
}
```

Vergleich der Realisierungen

Beim Betrachten der drei zuvor beschriebenen Umsetzungen wird deutlich, dass die objektorientierte Lösung die kürzeste und eleganteste ist, da hier aufgrund der Polymorphie die expliziten Typprüfungen entfallen können. Sowohl der imperative als auch der mit Objekten arbeitende Ansatz können darauf nicht verzichten, denn nur so kann dort die Unterscheidung getroffen werden, welches grafische Element gezeichnet werden soll. Die beiden Realisierungen ähneln einander auf den ersten Blick. Allerdings bietet der Entwurf mit Objekten einen entscheidenden Vorteil gegenüber dem imperativen Entwurf – dort besitzen die einzelnen Objekte alle Daten zum Zeichnen und erledigen dies selbst, statt dies in der Zeichenfläche zu realisieren.

Leider sieht man Sourcecode mit expliziten Typprüfungen viel häufiger, als man denken sollte. Das ist problematisch, weil man sich dadurch auf alle zu verarbeitenden Typen festlegt. Spätere Erweiterungen erfordern Änderungen in der Typprüfung und in der Realisierung. Beide Lösungen verstoßen demzufolge gegen das OPEN CLOSED PRINCIPLE (OCP), eines der fünf SOLID genannten Prinzipien für guten OO-Entwurf, die wir später in Abschnitt 3.5.3 kennenlernen werden.

Die objektorientierte Lösung nutzt die Vorteile der Polymorphie. Dadurch können sogar Figuren gezeichnet werden, deren Klassen zum Zeitpunkt der Implementierung der Klasse `PaintingArea` noch unbekannt sind, solange diese Figurenklassen von der gemeinsamen Basisklasse `BaseFigure` abgeleitet sind.

Mögliche Probleme bei Erweiterungen

Stellen wir uns vor, wir wollten nun auch Polygone oder Kreise zeichnen. Bei der imperativen und der mit Objekten arbeitenden Realisierung müssten wir für jede neue grafische Figur eine weitere Fallunterscheidung in Form einer `if`-Anweisung einfügen, wie dies der folgende Sourcecode exemplarisch für Polygone zeigt:

```
void drawFigure(int graphicsObjectType, int x1, int y1, int x2, int y2,
                int x3, int y3, /*...*/ int xn_1, int yn_1, int xn, int yn)
{
    if (graphicsObjectType == POINT_TYPE)
    {
        drawPoint(x1, y1);
    }

    // usw.

    if (graphicsObjectType == POLYGON_TYPE)
    {
        // Zeichne Polygon durch mehrere Linien
        drawLine(x1, y1, x2, y2);
        drawLine(x2, y2, x3, y3);
        // ...
        drawLine(xn_1, yn_1, xn, yn);
    }
}
```

Im imperativen Fall müssten wir zudem die Parameterliste im Extremfall stark erweitern, um alle benötigten Informationen für alle Arten von Figuren übergeben zu können. Dabei treten leicht Situationen auf, in denen Parameter für andere Figurtypen schlicht unbenutzt und damit überflüssig sind oder sich sogar widersprechen. Ansatzweise haben wir dies bereits für das Parameterpaar `x2`, `y2` und das Zeichnen von Punkten kennengelernt. Noch schlimmer ist, dass wir Erweiterungen in der Klasse `PaintingArea` vornehmen müssten, die das Zeichnen der neu eingeführten grafischen Objekte realisieren. Unter Umständen haben wir keinen Zugriff auf den Sourcecode dieser Klasse oder wir dürfen das API nicht mehr ändern, insbesondere wenn schon eine breite Nutzerbasis

existiert.[10] In beiden Situationen sind Erweiterungen nur noch extrem schwierig zu realisieren, etwa durch Tricks, wie neue Figuren durch bestehende Operationen zeichnen zu lassen. Wir wollen uns gar nicht vorstellen, was passieren würde, wenn wir auf diese Art Kreise oder gefüllte Figuren zeichnen müssten.

Der objektbasierte Entwurf hat zumindest keine Probleme mit den Unzulänglichkeiten bei der Parameterübergabe. Alle Definitions- und Implementierungsdetails, z. B. Ecken, Stützpunkte oder Abmessungen und Zeichenroutinen, werden verborgen. Demnach erspart uns die Kapselung schon viele Probleme der imperativen Lösung.

Wie schon erwähnt, verstoßen beide Lösungen gegen das OPEN CLOSED PRINCIPLE (OCP) (vgl. Abschnitt 3.5.3) und es ist einiges an Aufwand bei Erweiterungen vonnöten. Die objektorientierte Lösung folgt dagegen dem OCP und ist absolut einfach zu erweitern: Die Klasse `PaintingArea` bleibt so, wie sie ist. Jedes neue grafische Objekt von Typ `BaseFigure` erfordert dann lediglich noch eine eigene Realisierung.

3.1.4 Diskussion der OO-Grundgedanken

Nachdem wir mittlerweile einige Erkenntnisse bezüglich OO gewonnen haben, möchte ich die zu Beginn dieses Kapitels kurz vorgestellten Grundgedanken des OO-Entwurfs aufgreifen und diese mit dem neu gewonnenen Verständnis betrachten.

Datenkapselung und Trennung von Zuständigkeiten

Die Datenkapselung, also das Verbergen von Informationen, stellt ein Kernkonzept der objektorientierten Programmierung dar und ermöglicht es, die Daten eines Objekts vor Änderungen durch andere Objekte mithilfe von Zugriffsmethoden zu schützen. Zugriffsmethoden erleichtern auch eine Konsistenzprüfung eingehender Werte und eine zuverlässige Synchronisierung für Multithreading. Des Weiteren erlaubt es die Datenkapselung, die Speicherung der Daten eines Objekts unabhängig von dem nach außen bereitgestellten Interface zu ändern.

Aufgrund der obigen Argumentation ist in vielen Fällen der Einsatz von Zugriffsmethoden dem direkten Zugriff auf Attribute vorzuziehen – aber es gibt auch Ausnahmen! Die Forderung nach Zugriffsmethoden sollte man nicht dogmatisch umsetzen: Für Package-interne Klassen oder lokal innerhalb einer anderen Klasse definierte Hilfsklassen ist es häufig akzeptabel, auf Zugriffsmethoden zu verzichten und direkt auf Attribute zuzugreifen. Dies gilt allerdings nur dann, wenn absehbar ist, dass

1. diese Klasse nicht außerhalb des eigenen Packages verwendet wird,
2. keine größeren strukturellen Änderungen (viele neue Attribute, geänderte Assoziationen usw.) zu erwarten sind und
3. keine anderen Rahmenbedingungen, wie etwa Thread-Sicherheit, gegen eine direkte Nutzung und die dadurch reduzierte Kapselung sprechen.

[10] Änderungen im API würden dann in allen Applikationen, die das API einsetzen, zu Inkonsistenzen oder Kompilierfehlern führen.

Die Vorteile der Datenkapselung wollen wir uns anhand eines Beispiels ansehen: Dazu ist eine Klasse zur Modellierung von Personen und zugehörigen Adressdaten zu erstellen. Im Folgenden betrachten wir drei verschiedene Umsetzungen, die als UML-Klassendiagramm in Abbildung 3-10 dargestellt sind.

Abbildung 3-10 *Kapselung am Beispiel unterschiedlicher Personenklassen*

Die bisher verwendeten UML-Diagramme zeigten intuitiv verständliche Elemente. Hier sind jedoch bereits einige Feinheiten enthalten: Die Zeichen '+' und '–' dienen zur Beschreibung der Sichtbarkeiten `public` und `private`. Attribute können im mittleren Kasten und Methoden im unterer Kasten angegeben werden. Eine Verbindungslinie zwischen zwei Klassen beschreibt eine Assoziation. Weitere Details zur UML finden Sie auf der Webseite zum Buch `www.dpunkt.de/java-profi`.

Eine Realisierung ohne Kapselung ist mit der Klasse `PersonNoEncapsulation` gezeigt. Öffentlich zugängliche Attribute speichern die Informationen zu Name, Alter und Adressdaten. Zudem wird auf Zugriffsmethoden verzichtet. Dadurch müssen andere Klassen direkt auf die öffentlichen Attribute zugreifen. Schnell kommt der Wunsch auf, die Adressinformationen in eine eigenständige Klasse `Address` auszulagern, um eine bessere Trennung von Zuständigkeiten zu erreichen und die Kohäsion der einzelnen Klassen zu vergrößern. Weil andere Klassen bisher direkt auf die jetzt auszulagernden Attribute zugegriffen haben, gestalten sich selbst Änderungen an den Implementierungsdetails, hier das Herauslösen einer Klasse `Address`, als schwierig. Es sind nämlich als Folge (massive) Änderungen in allen einsetzenden Klassen notwendig. ***Voraussetzung für weniger Aufwand bei Änderungen ist demnach, dass der Zugriff auf Attribute gekapselt erfolgt und die Attribute möglichst `private` deklariert sind***.

Wäre die ursprüngliche Klasse analog zur Klasse `PersonWithEncapsulation` mit Kapselung realisiert worden, so wäre ein solcher Auslagerungsschritt wesentlich einfacher möglich. Es wäre lediglich eine Klasse `Address` zu definieren und alle zugehörigen Methodenaufrufe aus der Klasse `Person` an diese weiter zu delegieren. Die

Klasse `PersonWithEncapsulationAndAssociation` setzt dies um und behält das nach außen veröffentlichte API bei. Daher müssen keine Anpassungen in anderen Klassen erfolgen. Anhand dieses Beispiels wird klar, dass die Datenkapselung sehr wichtig ist. Anpassungen lassen sich so meistens auf eine oder wenige Klasse(n) begrenzen, also ohne weitreichende Änderungen in anderen Klassen zu verursachen.

Achtung: API-Design – Einfluss von Kapselung und Kopplung

Kapselung und Kopplung haben einen großen Einfluss auf den Entwurf von APIs:

- Findet keine Datenkapselung statt, so sind alle öffentlichen Attribute nach außen für andere Klassen sichtbar und damit Bestandteil des APIs.
- Trotz Datenkapselung wird in Zugriffsmethoden der Rückgabedatentyp und damit häufig auch der Datentyp des korrespondierenden Attributs veröffentlicht.

Es kann zu Problemen führen, wenn sich Änderungen an internen Daten im API bemerkbar machen und Änderungen in nutzenden Klassen erforderlich sind. Das gilt vor allem für öffentliche Schnittstellen, insbesondere das veröffentlichte und durch andere Programme verwendete API, wenn man allgemeingültige oder zentrale Komponenten realisiert. Handelt es sich bei Rückgabetypen jedoch nur um primitive Typen oder ein Interface, dann gibt es keine Abhängigkeiten von den Implementierungsdetails der realisierenden Klasse. Für andere API-Änderungen müsste man, um Probleme bei Nutzern zu vermeiden, alle diese Anwender der eigenen Klasse kennen und dafür sorgen, dass die notwendigen Folgeanpassungen in den benutzenden Programmen durchgeführt werden. Wird eine Klasse jedoch von diversen externen Klassen verwendet, die man nicht im Zugriff hat, so kann man nachträglich Änderungen zur Verbesserung des Designs nicht mehr ohne Folgen durchführen: Einige Programme lassen sich dann nicht mehr kompilieren oder man ist gezwungen, mehrere ähnliche Methoden anzubieten und die veralteten Stände passend zu kennzeichnen.

Diese kurze Diskussion sensibilisiert für die Probleme, ein gutes API zu entwerfen. Wie schwer dies trotz hoher Kompetenz sein kann, stellt man am JDK mit den vielen als veraltet markierten Methoden fest. Dies kann über den Javadoc-Kommentar `@deprecated`, die Annotation `@Deprecated` oder besser sogar durch beides kenntlich gemacht werden. Derartige Methoden sollte man in neu erstelltem Sourcecode nicht mehr verwenden.

Vererbung und Wiederverwendbarkeit

Mithilfe von Vererbung kann durch den Einsatz von Basisklassen und das Nutzen gemeinsamer Funktionalität ein übersichtlicheres Design erreicht und so mehrfach vorhandener Sourcecode vermieden werden. Allerdings sollte man Vererbung auch immer mit einer gewissen Vorsicht einsetzen. Sie ist zwar ein Mittel, um bereits modelliertes Verhalten vorhandener Klassen ohne Copy-Paste-Ansatz wiederzuverwenden und zu erweitern, aber nicht mit dem alleinigen Ziel, keinen Sourcecode zu duplizieren. Eine

Vererbung, die lediglich dem Zweck dient, bereits existierende Funktionalität aus bestehenden Klassen zu verwenden, führt oft zu zweifelhaften oder gar unsinnigen Designs und wird ***Implementierungsvererbung*** genannt. Für die entstehenden Klassen gilt des Öfteren dann keine wirkliche Subtyp-Beziehung mehr, wodurch es zu Überraschungen bei deren Einsatz kommen kann. Das schauen wir uns im Rahmen der Besprechung des LISKOV SUBSTITUTION PRINCIPLE (LSP) in Abschnitt 3.5.3 etwas genauer an.

Durch mehrere Klassen verwendbare Funktionalität sollte in eine separate Hilfsklasse ausgelagert und dann per ***Delegation***[11] statt Vererbung angesprochen werden.

3.1.5 Wissenswertes zum Objektzustand

In den vorangegangenen Abschnitten wurde schon einiges zum objektorientierten Entwurf thematisiert. Nachfolgend möchte ich das Thema ***Objektzustand*** vertiefen.

Bereits bekannt ist, dass man unter dem momentanen Objektzustand die aktuelle Belegung der Attribute eines Objekts versteht. Die Menge der potenziellen Objektzustände eines konkreten Objekts umfasst sämtliche möglichen Belegungen von dessen Attributen. Die gültigen Zustände beschränken sich meistens auf eine (kleine) Teilmenge daraus. Ausgehend von einem oder mehreren Startzuständen sollten nur solche Zustandsübergänge erlaubt sein, die wiederum zu einem gültigen Objektzustand führen. Dieser Wunsch wird in Abbildung 3-11 verdeutlicht. Dort sind einige gültige Zustände und Zustandsübergänge visualisiert. Der deutlich größere Zustandsraum ist als graues Rechteck dargestellt. Die damit beschriebenen Zustände sind alle möglich, aber nicht jeder verkörpert einen sinnvollen Objektzustand; einige davon sind explizit als `Invalid 1`, `Invalid 2` bzw. `Invalid n` dargestellt. Bei einem `Person`-Objekt könnten dies etwa negative Werte für Alter oder Größe sein.

Abbildung 3-11 *Gültige Objektzustände und Zustandsraum*

[11] Delegation beschreibt, dass eine Aufgabe einem anderen Programmteil übergeben wird, etwa indem eine Klasse eine andere über eine Assoziation kennt und deren Methoden aufruft.

Die Forderung nach der Einhaltung eines gültigen Objektzustands ist gar nicht so leicht zu erfüllen. Tatsächlich wird dies sogar in vielen Programmen vernachlässigt: Vielfach wird der erlaubte Zustandsraum nicht geprüft oder ist nicht einmal festgelegt. Dadurch besitzen Objekte mitunter beliebige, wahrscheinlich auch ungültige Zustände. Dies löst häufig unerwartete Programmreaktionen aus, führt zu Berechnungsfehlern oder Abstürzen und erhöht den Aufwand beim Testen. Um die genannten Probleme zu verhindern, müssen wir besonderes Augenmerk auf all diejenigen Methoden werfen, die zu Änderungen am Objektzustand führen. Eine Möglichkeit dazu ist es, einen gültigen Ausgangszustand unter anderem durch die Prüfung der Eingabeparameter von Business-Methoden sicherzustellen. Ungültige Parameterwerte sollten zurückgewiesen werden, um eine sichere Ausführung von weiteren Methoden zu gewährleisten. Zudem müssen durchgeführte Berechnungen wiederum gültige Objektzustände erzeugen. Das Ganze wollen wir ein klein wenig formaler betrachten, bevor wir es an einem Beispiel verdeutlichen.

Invarianten, Vor- und Nachbedingungen

Sogenannte Vor- und Nachbedingungen sowie Invarianten spielen bei der Programmverifikation und dem Beweis der Korrektheit von Programmen in der theoretischen Informatik eine wichtige Rolle. In der Praxis kann man auch ohne Kenntnis der Details von diesen profitieren und ungültige Objektzustände vermeiden.

Mithilfe einer ***Invariante*** kann man Aussagen über Teile des Objektzustands treffen. Damit beschreibt man *unveränderliche* Bedingungen, die vor, während und nach der Ausführung verschiedener Programmanweisungen gültig sein sollen. Ein Beispiel für eine solche Forderung ist, dass das Alter einer Person nicht negativ sein darf.

Über ***Vorbedingungen*** lässt sich eine zur Ausführung einer Methode notwendige Ausgangssituation beschreiben: Die Vorbedingung einer Suchmethode könnte etwa sein, dass Elemente vorhanden und sortiert sind. Wenn diese Vorbedingung erfüllt ist, so wird nach Ausführung der Methode als Ergebnis die Position des gesuchten Elements zurückgeliefert, oder aber -1, wenn kein solches existiert. Bedingungen nach Ausführung eines Programmstücks werden durch ***Nachbedingungen*** formuliert.

Info: Design by Contract

Der Einsatz von Invarianten, Vor- und Nachbedingungen erinnert an die Forderungen des von Bertrand Meyer beim Entwurf der Programmiersprache Eiffel erfundenen »***Design by Contract***«. Idee dabei ist es, Vor- und Nachbedingungen bei jedem Methodenaufruf abzusichern. Dies soll verhindern, dass eine Methode mit ungültigen Werten aufgerufen wird. Auf diese Weise versucht man, die Konsistenz des Programmzustands zu wahren sowie ein mögliches Fehlverhalten des Programms auszuschließen. Beim Verlassen einer Methode wird durch Nachbedingungen gewährleistet, dass nur erwartete, gültige Werte als Rückgabe möglich sind. Weitere Informationen finden Sie unter `http://www.eiffel.com/developers/design_by_contract.html`.

Der Objektzustand am Beispiel

Im folgenden Beispiel eines grafischen Editors werden wir Vor- und Nachbedingungen nutzen, um sicherzustellen, dass die Koordinaten grafischer Figuren einem Rasterpunkt entsprechen. Nehmen wir an, es soll ein Raster mit einem Abstand von 10 Punkten modelliert werden.

Wir entwerfen dazu eine Klasse `GridPosition`, die ihre Koordinaten in den zwei Attributen `x` und `y` vom Typ `int` speichert. Gültige Punkte, und damit Belegungen der Attribute, sollen immer auf den genannten Rasterpunkten liegen. Die Wertebelegungen der Attribute vom Typ `int` beschreiben den Zustandsraum, d. h. alle potenziell möglichen Objektzustände. Die Menge der erlaubten Zustände ist jedoch wesentlich geringer. Das gilt im Speziellen für dieses Beispiel und lässt sich auf viele Fälle in der Praxis übertragen.

Die Klasse `GridPosition` bietet zwei Business-Methoden, die Objektverhalten beschreiben – dies sind `addOffset(int, int)` und `setSamePosition(int)`. Zur Demonstration der Gefahr von Inkonsistenzen im Objektzustand sind zusätzlich öffentliche `set()`-Methoden für die Attribute `x` und `y` definiert.

```
public final class GridPosition
{
    private static final int GRID_SIZE = 10;

    // Gewünschte Invariante: x, y liegen immer auf einem Raster der Größe 10
    private int x = 0;
    private int y = 0;

    public void addOffset(final int dx, final int dy)
    {
        // Vorbedingung: x, y auf einem beliebigen Rasterpunkt
        checkOnGrid(x,y);

        x += snapToGrid(dx);
        y += snapToGrid(dy);

        // Nachbedingung: x, y immer noch auf einem Rasterpunkt
        checkOnGrid(x,y);
    }

    public void setSamePosition(final int position)
    {
        // Vorbedingung: x, y auf einem beliebigen Rasterpunkt
        checkOnGrid(x,y);

        x = snapToGrid(position);
        y = snapToGrid(position);

        // Nachbedingung: x = y und immer noch auf einem Rasterpunkt
        checkOnGrid(x,y);
    }

    private static void checkOnGrid(final int x, final int y)
    {
        if (x % GRID_SIZE != 0 || y % GRID_SIZE != 0)
            throw new IllegalStateException("invalid position, not on grid");
    }
```

```
    private int snapToGrid(final int value)
    {
        return value - value % GRID_SIZE;
    }

    public int getX()                    { return x; }
    public int getY()                    { return y; }
    public void setX(final int x)        { this.x = x; }  // problematisch!
    public void setY(final int y)        { this.y = y; }  // problematisch!
}
```

Dem aufmerksamen Leser fällt die ungewöhnliche Formatierung der Zugriffsmethoden (`get()`-/`set()`-Methoden) auf. Die hier angewendete kompakte Schreibweise wird in Abschnitt 19.3.1 diskutiert. Nur so viel vorab, sie kann die Lesbarkeit für derart einfache Zugriffsmethoden mit sehr wenig Sourcecode, bevorzugt einer Anweisung, erhöhen.

Veränderungen am Objektzustand

Nach der Konstruktion eines `GridPosition`-Objekts entspricht der Startzustand der Wertebelegung $x = y = 0$. Wie bereits erwähnt, sollten öffentliche Methoden ein Objekt aus einem gültigen Objektzustand in einen anderen gültigen versetzen. Hier gilt:

- Jeder Aufruf von `addOffset(dx, dy)` führt zu einem Zustandswechsel, bei dem Folgendes gilt: $x = x + dx^*$ und $y = y + dy^*$.
- `setSamePosition(pos)` wechselt in den Zustand $x = y = pos^*$.[12]

In der Klasse `GridPosition` werden jedoch zwei öffentliche `set()`-Methoden angeboten, in denen keine Korrektur der Eingabewerte stattfindet. Rufen andere Klassen diese `set()`-Methoden statt der Business-Methoden auf, so können beliebige Werte für die Attribute `x` und `y` gesetzt werden. Diese liegen höchstwahrscheinlich nicht auf dem gewünschten Raster und stellen somit auch keinen gültigen Objektzustand dar. Die gewünschte Invariante kann somit nicht sichergestellt werden.

Tipp: Probleme durch öffentliche `set()`-Methoden

In der Praxis findet man aufgrund fehlender (oder zu weniger) Business-Methoden eine Vielzahl öffentlicher `set()`-Methoden. Deren Aufruf ermöglicht es aber, den Zustand eines Objekts (unerwartet oder zumindest nicht durch das Objekt selbst kontrolliert) zu verändern. Häufig wird dann das Objektverhalten von Aufrufern durch eine Abfolge von `get()`/`set()`-Methoden realisiert. Dieser Weg an den Business-Methoden vorbei erschwert die Einhaltung eines gültigen Objektzustands ungemein. Als Abhilfe sollten einige `set()`-Methoden in ihrer Sichtbarkeit eingeschränkt oder, falls möglich, durch Business-Methoden ersetzt werden. Ein Vorgehen ist als Refactoring ERSETZE MUTATOR- DURCH BUSINESS-METHODE in Abschnitt 17.4.4 beschrieben. In der Klasse `GridPosition` wurden bereits Business-Methoden bereitgestellt, die `set()`-Methoden allerdings (noch) nicht entfernt.

[12] * = angepasster Wert: Um sicherzustellen, dass nur Rasterpunkte eingenommen werden, erfolgt in beiden Fällen durch Aufruf der Methode `snapToGrid(int)` eine Modulo-Korrektur.

Objektzustände während der Objektinitialisierung

Manchmal kommt es während der Objektinitialisierung zu ungewünschten, vermeidbaren oder sogar ungültigen Zwischenzuständen. Dies gilt insbesondere, wenn lediglich ein Defaultkonstruktor und diverse `set()`-Methoden angeboten werden.

Die Grundlage für die Diskussion bildet folgende Klasse `SimpleImage`. Sie verwendet einen Namen, eine Breite und eine Höhe sowie die eigentlichen Bilddaten und speichert diese Werte in vier privaten Attributen. Der Zugriff ist durch öffentliche `get()`- und `set()`-Methoden möglich – hier nur für das Attribut `name` gezeigt:

```
public final class SimpleImage
{
    private String name;
    private int width;
    private int height;
    private byte[] imageData;

    public SimpleImage()
    {}

    public void setName(final String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return this.name;
    }

    // weitere Getter und Setter ...
}
```

Das von der Klasse angebotene API bietet keine Unterstützung zur Sicherstellung und Wahrung eines gültigen Objektzustands. Vielmehr ist dazu erforderlich, dass nach der Objektkonstruktion alle Attribute durch Aufruf von `set()`-Methoden sukzessive mit gültigen Werten initialisiert werden. Betrachten wir den Ablauf anhand des Einlesens aus einer Datei in der folgenden Methode `createSimpleImageFromFile()`:

```
public SimpleImage createSimpleImageFromFile() throws IOException
{
    final SimpleImage simpleImage = new SimpleImage();

    // ACHTUNG: Nur zur Demonstration zu früh registrieren
    storedImages.add(simpleImage);

    final String imageName = readNameFromFile();
    simpleImage.setName(imageName);          // Name setzen

    try
    {
        final int imageWidth = readWidthFromFile();
        simpleImage.setWidth(imageWidth);    // Breite setzen

        final int imageHeight = readHeightFromFile();
        simpleImage.setHeight(imageHeight);  // Höhe setzen
    }
```

```
    catch (final NumberFormatException e)
    {
        // ACHTUNG: Keine Fehlerbehandlung zur Demonstration
    }

    final String imageData = readImageDataFromFile();
    simpleImage.setImageData(imageData);     // Daten setzen

    return simpleImage;
}
```

Ähnliche Programmausschnitte trifft man in der Praxis immer wieder an. Was daran problematisch ist, wollen wir nun analysieren:

1. **Ungültiger Objektzustand** – Es werden ungültige Objektzustände sichtbar. Im Beispiel kann dies dadurch geschehen, dass andere Programmteile die Liste der Bilder `storedImages` zu früh – vor Abschluss der Objektinitialisierung eines neu hinzugefügten Bilds – auslesen.
2. **Teilinitialisierter Objektzustand** – Treten Fehler beim Einlesen aus der Datei oder beim Verarbeiten der eingelesenen Daten auf, so verbleibt das Objekt in einem teilinitialisierten Objektzustand, der höchstwahrscheinlich keinem gültigen Objektzustand entspricht.
3. **Unnötige Zugriffsmethoden** – Es müssen viele, bei einem anderen Design jedoch unnötige Zugriffsmethoden für Attribute bereitgestellt werden. Dies verhindert eine Realisierung, die nur in den wirklich notwendigen Teilen veränderlich ist.
4. **Öffentliche Zugriffsmethoden** – Öffentlich definierte Zugriffsmethoden erlauben beliebige Änderungen durch andere Objekte. Daraus können inkonsistente Objektzustände resultieren.

Ungültiger Objektzustand Im Beispiel wird zunächst ein `SimpleImage`-Objekt über einen Defaultkonstruktor erzeugt. Vorausgesetzt es werden gültige Eingabedaten an die vier `set()`-Methoden übergeben, so führt dies auch zu vier Änderungen im Objektzustand. Würde ein anderes Objekt genau das neu erzeugte `SimpleImage`-Objekt aus der Liste `storedImages` auf Veränderungen untersuchen, so sind im Extremfall vier Änderungen in kurzer Abfolge nacheinander beobachtbar. Erst die letzte davon erzeugt einen gültigen Objektzustand.

Teilinitialisierter Objektzustand Nehmen wir an, einer der eingelesenen Werte für Breite und Höhe wäre keine Zahl und damit ungültig. Die unzureichende Fehlerbehandlung (der leere `catch`-Block) verhindert, dass dieses Problem zum Aufrufer propagiert wird. Stattdessen kommt es zu einer unbemerkten Inkonsistenz im Objektzustand: Das gerade neu konstruierte `SimpleImage`-Objekt besitzt nach dem Einlesen der korrekten Werte von Name und Bilddaten falsche Werte für Breite oder Höhe.

Unnötige Zugriffsmethoden Alle Attribute der Klasse `SimpleImage` sind über `set()`-Methoden veränderbar. Allerdings wurden einige dieser Methoden lediglich eingeführt, um ein sukzessives Setzen der Werte zu ermöglichen. Für die Nutzung der Klasse ist wahrscheinlich, dass Höhe, Breite und Bilddaten voneinander abhängig sind und nur miteinander als Einheit zu verändern sein sollten. Statt einzelner `set()`-Methoden sollte man daher eine Business-Methode `changeImageData(int, int, byte[])` anbieten. Zudem sollte der Name des Bildes unveränderlich sein. Dies wird durch die Deklaration des Attributs `name` als `final` erreicht:

```
public final class SimpleImage
{
   private final String name;
   private int width;
   private int height;
   private byte[] imageData;

   public SimpleImage(final String name, final int width, final int height,
                      final byte[] imageData)
   {
      this.name = name;
      changeImageData(width, height, imageData);
   }

   public void changeImageData(final int width, final int height,
                               final byte[] imageData)
   {
      this.width = width;
      this.height = height;
      this.imageData = imageData;
   }

   public String getName()
   {
      return this.name;
   }

   // weitere get()-Methoden ...
}
```

Die Implementierung gewinnt durch diese Änderungen an Lesbarkeit. Wichtiger ist aber, dass die Abhängigkeiten der Attribute untereinander nun deutlich zu erkennen sind und nur durch eine Methode in der öffentlichen Schnittstelle beschrieben werden. Diese Business-Methode ist auch die richtige Stelle, um Änderungen an andere Objekte zu kommunizieren.

Betrachten wir, wie sich das neue API der Klasse `SimpleImage` im Einsatz auswirkt. Statt der direkten `set()`-Aufrufe erfolgt zunächst eine Zwischenspeicherung in lokalen Variablen. Erst nachdem alle Daten erfolgreich eingelesen werden konnten, wird ein `SimpleImage`-Objekt konstruiert. Dadurch werden nur gültige Zustände nach außen sichtbar. Selbst die fehlende Behandlung eines Parsing-Fehlers hat bei dieser Art der Realisierung deutlich weniger negative Auswirkungen. Es wird dann einfach kein Objekt erzeugt. Folgendes Listing zeigt das neue API im Einsatz:

```
public SimpleImage createSimpleImageFromFile() throws IOException
{
   final String imageName = readNameFromFile();

   try
   {
      final int imageWidth = readWidthFromFile();
      final int imageHeight = readHeightFromFile();
      final String imageData = readImageDataFromFile();

      final SimpleImage simpleImage = new SimpleImage(imageName, imageWidth,
                                                      imageHeight,
                                                      imageData.getBytes());

      // Das ist jetzt immer möglich, da SimpleImage korrekt erzeugt wurde
      storedImages.add(simpleImage);

      return simpleImage;
   }
   catch (final NumberFormatException ex)
   {
      log.error("failed to create SimpleImage object from file", ex);
   }

   throw new IOException("illegal content. failed to parse width/height", ex);
}
```

Falls Zugriffsprobleme beim Lesen aus der Datei auftreten, wird kein `SimpleImage`-Objekt erzeugt, sondern eine `IOException` ausgelöst. Darüber hinaus gibt es noch den Spezialfall zu behandeln, dass die hinterlegten Angaben zu Breite und Höhe keine Zahlenwerte darstellen, was zu einer `NumberFormatException` führt. Wir wollen dieses Detail jedoch nicht direkt an den Aufrufer der Methode weiterleiten. Daher wrappen wir die Exception in eine `IOException` und geben dort die Information über das Problem mit. Alternativ könnte man den Wert `null` zurückliefern. Das würde aber aufseiten des Aufrufers eine Spezialbehandlung erfordern, weshalb wir hier die zuvor beschriebene Variante wählen. Seit Java 8 bietet sich für derartige Fälle noch die Verwendung der Klasse `java.util.Optional<T>` an (vgl. Abschnitt 6.5).

Öffentliche Zugriffsmethoden Alle Attribute der eingangs gezeigten Klasse `SimpleImage` waren über öffentliche `set()`-Methoden veränderlich. Damit konnten Klienten[13] beliebige Änderungen am Objektzustand durchführen. Im vorherigen Schritt haben wir bereits die Anzahl öffentlicher, den Objektzustand beeinflussender Methoden verringert. Als Folge konnten sämtliche `set()`-Methoden entfallen. Eine Zustandsänderung ist nur noch durch Aufruf der Methode `changeImageData(int, int, byte[])` möglich. Erfolgt innerhalb dieser Methode eine Parameter- und Konsistenzprüfung, so kann das Objekt von anderen Klassen nicht mehr in einen ungültigen Objektzustand versetzt werden. Dies gilt so jedoch nur für Singlethreading. Für Multithreading müsste dazu noch eine geeignete Synchronisierung erfolgen, um Inkonsistenzen sicher auszuschließen.

[13] Darunter versteht man beliebige andere Klassen, die Methoden der eigenen Klasse aufrufen.

Weitere Komplexitäten durch Multithreading

Anweisungen werden nicht als Ganzes abgearbeitet, sondern in Form kleiner Schritte. Dadurch kommt es während der Ausführung der Anweisungen teilweise kurzzeitig zu ungültigen oder unerwarteten Belegungen im Objektzustand. Abbildung 3-12 zeigt dies für den Aufruf `addOffset(50, 30)`. Dort wird temporär ein Zustand `x=50; y=0` eingenommen, der aber logisch keinem Methodenaufruf entspricht.

Abbildung 3-12 *Objektzustandsübergänge*

Diese Zwischenzustände lassen sich systembedingt nicht verhindern – sind bei Singlethreading jedoch nicht beobachtbar. Für Multithreading muss dafür gesorgt werden, dass diese Zwischenzustände für andere Objekte nicht zugreifbar sind. Einen wichtigen Beitrag dazu leisten Datenkapselung sowie die Definition kritischer Bereiche.

Datenkapselung Mit Datenkapselung lässt sich vermeiden, dass andere Objekte jederzeit einen Blick in die »Innereien« eines anderen Objekts werfen können und möglicherweise mit ungültigen Zwischenzuständen weiterarbeiten. Im ungünstigsten Fall sind ohne Datenkapselung sogar externe, verändernde Zugriffe möglich, wodurch das Objekt die Kontrolle über seinen Objektzustand abgibt. Das ist nahezu immer unerwünscht. Bei Datenkapselung verhindern die `private` definierten Attribute die zuvor beschriebenen Probleme.

Kritische Bereiche Um bei Multithreading unerwünschte Zwischenzustände durch Thread-Wechsel während der Berechnungen zu vermeiden, kann man kritische Bereiche nutzen. Das ist notwendig, weil ein aktiver Thread nahezu jederzeit in seiner Ausführung unterbrochen und ein anderer Thread aktiviert werden kann. Geschieht ein solcher Thread-Wechsel während der Abarbeitung einer Methode, sind eventuell noch nicht alle Zuweisungen an die Attribute erfolgt und ein anderer Thread sieht einen Zwischenzustand. Kapitel 10 behandelt diese Problematik ausführlich.

3.2 Grundlegende OO-Techniken

Bis hierher haben wir einige wichtige Einblicke in das objektorientierte Design bekommen. Dieser Abschnitt geht nochmals etwas genauer auf grundlegende OO-Techniken ein, die wir für den Entwurf größerer Softwaresysteme als Basisbausteine benötigen: Schnittstellen, Basisklassen und deren Kombination. All dies hilft dabei, Funktionalität zu kapseln und Komponenten (bestehend aus Klassen, Interfaces und Packages) voneinander möglichst unabhängig sowie änder- und erweiterbar zu halten.

Abschnitt 3.2.1 behandelt die Definition von Schnittstellen. Durch explizit definierte Schnittstellen, auch Interfaces genannt, lässt sich ein »Angebot von Verhalten« in Form einer Menge von Methoden festlegen. In Abschnitt 3.2.2 lernen wir zunächst, wie und warum Basisklassen entstehen, und dann, wozu diese dienen. Danach stelle ich eine spezielle Form von Basisklassen vor, nämlich abstrakte Basisklassen, die sich dadurch auszeichnen, dass sie nicht instanziierbar sind. Abstrakte Basisklassen können als Stellvertreter für konkrete Subklassen agieren, die das gewünschte, spezifische Verhalten realisieren. Für sich allein eingesetzt sind die Techniken Interface und abstrakte Basisklasse bereits hilfreich, aber sie lassen sich darüber hinaus gewinnbringend kombinieren. Darauf geht Abschnitt 3.2.3 genauer ein.

3.2.1 Schnittstellen (Interfaces)

Wenn Aufrufe an Methoden anderer Klassen erfolgen, dient dazu eine sogenannte Use-Beziehung: Eine Klasse verwendet eine andere. Dabei gibt es einen Aufrufer und einen Bereitsteller von Funktionalität. Man spricht zum Teil auch von Client und Server. Im einfachsten Fall ergibt sich die Schnittstelle, die ein Client von einem Server nutzen kann, implizit über die öffentlichen Methoden. Manchmal ist es jedoch zweckmäßiger, Methoden zu gruppieren und diesen Funktionsblöcken einen eigenen Namen zu geben. Dies ist durch die nun besprochene Technik der Interfaces möglich.

Einführung

Am Beispiel eines Kunden (Client), der bei einem Pizza-Online-Lieferservice (Server) Bestellungen aufgeben kann, lernen wir die Definition einer Schnittstelle mithilfe des Schlüsselworts `interface` kennen. Wir schreiben Folgendes:

```
// Zeigt drei Varianten der Angabe von Zugriffsmodifiern
public interface IOnlinePizzaService
{
    public long registerNewCustomer(String name, Address address);
    abstract boolean orderPizza(long customerId, Pizza pizza);
    public abstract DeliveryInformation deliverTo(long customerId);
}
```

Ganz nebenbei gibt es folgende Besonderheit: Unabhängig von der expliziten Angabe der gezeigten Zugriffsmodifier besitzen die Methoden alle die Sichtbarkeit `public` und sind zudem `abstract`.

Tipp: Implizite Zugriffsmodifier

In Java sind alle in Interfaces definierten ...

- **Methoden** immer mit den Zugriffsmodifiern `public` und `abstract` definiert.
- **Variablen** implizit als `public`, `static` und `final` definiert. Diese sind dadurch als Konstanten nach außen sichtbar.

Die obigen Zugriffsmodifier können explizit im Sourcecode angegeben werden, und zwar einzeln oder gemeinsam, und sind unabhängig davon immer vorhanden. Alle anderen Zugriffsmodifier sind in Interfaces nicht erlaubt. Seit Java 8 bzw. 9 gilt das Ganze nicht mehr so streng, denn es sind nun Defaultmethoden sowie statische und sogar private Methoden als Sprachfeature möglich.

Interfaces als »Angebot von Verhalten«

Weil die in Interfaces aufgelisteten Methoden so etwas wie einen Vertrag beschreiben, können sich sowohl Client als auch Server darauf stützen. Dies ermöglicht eine Trennung von Realisierung und Spezifikation. Diese Entkopplung ist die Basis für viele Entwurfsmuster, wie ITERATOR, DEKORIERER, PROXY u. v. m. (vgl. Kapitel 18).

Die Definition eines Interface hat zwei Facetten: Zum einen wird für Clients festgelegt, welche Methoden sie aufrufen können. Zum anderen wird für einen Server bestimmt, welche Methoden durch ihn implementiert werden müssen, um das Interface zu erfüllen. Übertragen auf das Beispiel des Pizza-Online-Lieferservice bedeutet dies Folgendes: Für den Kunden ist der reale Pizza-Betrieb, also die konkrete Ausprägung, nicht von Interesse. Er benötigt für seine Bestellungen lediglich die Schnittstelle, z. B. die Internetseite (und eine Möglichkeit, diese zu erhalten[14]). Damit der Pizza-Betrieb seine Leistungen online zur Verfügung stellen kann, muss dieser wiederum nur wissen, welche Funktionalität er zur Erfüllung der Schnittstelle umsetzen muss. Möglicherweise können auch andere Pizza-Betriebe später die geforderte Schnittstelle erfüllen und ihre Dienste dann alternativ oder ergänzend anbieten.

Bevor Sie damit beginnen, jeweils ein Interface für Ihre Klassen einzuführen, bedenken Sie aber bitte Folgendes: Ein Interface ist ***kein Selbstzweck*** und es stellt per se auch kein gutes Design dar. Wie oben schon angedeutet, helfen Interfaces bei der ***Definition von Verhalten*** und bei der ***Entkopplung von Komponenten***. Dagegen ist innerhalb eigener Packages ein solcher Abstraktionsschritt eher selten vonnöten.

Nach diesen kurzen Vorbetrachtungen kommen wir nochmals auf Wissenswertes zur Definition einer Schnittstelle zurück.

[14] Eine zentrale Stelle wird häufig durch einen sogenannten Service Locator realisiert, meistens eine Klasse, die einen wohldefinierten Zugriff auf Services und Schnittstellen bereitstellt.

Wissenswertes zur Definition von Interfaces

Wie schon bekannt, wird durch ein Interface keine Realisierung vorgegeben, sondern nur einige ***Methoden ohne Implementierung***, d. h. ***abstrakte Methoden***. Beim Betrachten des obigen Interface kommen wir zu einer weiteren wichtigen Erkenntnis: Sowohl Nutzer als auch Implementierer verwenden lediglich die angegebene Menge an Methoden. Um beiden das Verständnis der bereitzustellenden Funktionalität und den Einsatz zu erleichtern, sollten die Methodennamen in Interfaces besonders sprechend und klar gewählt werden. Diese Forderung ist im zuvor gezeigten Interface `IOnlinePizzaService` schon recht gut erfüllt. Trotzdem ist nicht eindeutig, was die Methoden im Detail tun sollen und was die Eingabe- und Rückgabewerte bedeuten. Schauen wir uns exemplarisch eine für die Methode `registerNewCustomer(String, Address)` verbesserte Variante des Interface an:

```
public interface IOnlinePizzaService
{
    /**
     * create a new customer with the given name and delivery address
     *
     * @param  name -- the name of the customer to create
     * @param  deliveryAddress -- the customer's address
     *
     * @return the id of the newly created customer
     * @throws DuplicateCustomerException if a customer with the same
     *         name already exists
     */
    long registerNewCustomer(String name, Address deliveryAddress)
                             throws DuplicateCustomerException;

    boolean orderPizza(long customerId, Pizza pizza);
    DeliveryInformation deliverTo(long customerId);
}
```

In diesem Beispiel wurde aus Gründen der Übersichtlichkeit nur eine knappe Dokumentation für eine Methode gezeigt. Aber schon daran erkennt man, wie dies zum Verständnis beitragen kann. In der Praxis ist für alle Methoden eines Interface eine gute Dokumentation besonders empfehlenswert, damit Nutzer und Implementierer eines Interface die gleiche Vorstellung von dem bereitgestellten bzw. bereitzustellenden Verhalten haben.

Abgrenzung von Interfaces und Vererbung

Sowohl das Implementieren eines Interface als auch Vererbung sind Techniken, um Funktionalität zu beschreiben bzw. bereitzustellen. Während Vererbung eine Spezialisierung ausdrückt, liegt bei Interfaces der Fokus auf dem Angebot oder Ausüben von Funktionalität bzw. der Beschreibung einer ***Verhaltensweise*** (auch ***Rolle*** genannt). Es wird eine ***»can-act-like«-Beziehung*** oder auch ***»provides«-Beziehung*** realisiert.

Mit einem ***Interface*** wird für ein Objekt beschrieben, ***was es kann***. Im Gegensatz dazu wird mit ***Vererbung*** für ein Objekt beschrieben, ***was es ist***. Insbesondere sollte dabei die »is-a«-Beziehung eingehalten sein.

Mehrfachvererbung der Definition von Verhaltensweisen

Während in Java jede Klasse nur genau eine Basisklasse besitzen kann und somit Mehrfachvererbung von Zustand oder Verhalten durch mehrere Basisklassen nicht möglich ist, sind Klassen nicht darauf festgelegt, nur ein einziges Interface zu implementieren.

Eben erwähnte ich, dass mithilfe eines Interface für ein Objekt beschrieben wird, was es kann. Dieses Können einer Klasse muss jedoch nicht nur auf eine Funktionalität, Verhaltensweise oder Rolle beschränkt sein, sondern kann durchaus mehrere umfassen. Somit können Klassen bei Bedarf mehrere voneinander unabhängige Funktionalitäten oder Verhaltensweisen gemäß »can-act-like« ausüben.

Namensgebung von Interfaces

Teilweise ist es hilfreich, dass Interfaces als solche anhand ihres Namens erkennbar sind. Im JDK wird vielfach die Endung `-able` verwendet, um die »can-act-like«-Beziehung zu betonen, also die Beschreibung von Rollen. Nicht in jedem Fall macht diese Namensgebung tatsächlich Sinn und ist anwendbar. Während bei `java.lang.Runnable` recht klar wird, dass eine Schnittstelle vorgegeben wird, gilt dies für das Interface `javafx.event.EventHandler` zur Behandlung von GUI-Events in JavaFX nicht mehr. Anhand der für letzteres Interface gewählten Namensgebung ist keine Unterscheidbarkeit zu Klassen gegeben. Mitunter ist es aber wünschenswert, bereits am Namen des Typs einer Referenzvariablen zwischen Interface und Klasse unterscheiden zu können, um so zu erkennen, dass man lose Kopplung einhält und nicht versehentlich eine konkrete Klasse referenziert.

Für selbst definierte Interfaces wird in diesem Buch normalerweise das Präfix `I` verwendet – ab und zu kommt auch das Postfix `IF` zum Einsatz. Beides soll schon im Namen die Interface-Eigenschaft betonen. Dieses Vorgehen mit dem Prefix `I` findet man z. B. auch in der Programmierung von SWT (Eclipse RCP) oder als Standard in der Microsoft-Welt. Der Einsatz hängt stark davon ab, ob man eine Unterscheidung zwischen Interfaces und Klassen direkt durch den gewählten Namen sichtbar machen möchte. Die Kennzeichnung ist also tendenziell eine Frage von persönlichen Vorlieben oder durch Coding Conventions vorgegeben (vgl. Kapitel 19).

Bewertung

Der Einsatz von Interfaces ...

+ entkoppelt Spezifikation und Realisierung: Dadurch kennen sich nutzende und Funktionalität bereitstellende Klassen nicht direkt, sondern kommunizieren nur über das Interface.
+ erleichtert den Austausch einer konkreten Implementierung: Ein Beispiel dafür ist das Interface `List` aus dem Collections-Framework mit seinen Realisierungen `ArrayList`, `LinkedList` und `Vector`. Nutzende Klassen dürfen sich daher nicht

auf spezielle Implementierungen oder Implementierungsdetails verlassen. Für Listen darf man also weder einen konstanten Aufwand beim indizierten Zugriff auf Elemente annehmen (`ArrayList`) noch einen konstanten Aufwand beim Hinzufügen am Ende der Liste (`LinkedList`). Details dazu finden Sie in Abschnitt 6.1.5.

+ ermöglicht die spezifische Definition einer oder mehrerer wohldefinierter Schnittstellen. Eine Klasse kann dann je nach Anwendungsfall die eine oder andere Rolle spielen, abhängig davon, über welches Interface sie angesprochen wird.
+ erleichtert den Überblick über die angebotenen Methoden, da diese in Interfaces zentral definiert werden. Verzichtet man auf die Definition von Interfaces, so bilden alle öffentlichen Methoden die Schnittstelle einer Klasse. Diese ist einerseits nicht immer offensichtlich, da die öffentlichen Methoden im Sourcecode verstreut (bzw. durchmischt mit Methoden geringerer Sichtbarkeiten) sind, und andererseits wenig spezifisch, da sich eben alle öffentlichen Methoden ohne semantische Gruppierung in der Klasse befinden. Die Sammlung von Methoden in einem Interface bietet außerdem den Vorteil, dass sich leicht ein Eindruck verschaffen lässt, ob das Interface gut geschnitten ist (d. h., es besteht aus sinnvollen Methoden mit sprechenden Namen, die eine spezifische Funktionalität abdecken) oder eben nicht. Für folgendes Beispiel ist das offensichtlich nicht der Fall:

```
// Achtung: Negativbeispiel
interface IUniversalDiskPizzaAndMore
{
    void scanDisk(Drive drive) throws IOException;

    Iterable<Customer> getAllCustomers();

    boolean orderPizza(long customerId, Pizza pizza);
}
```

o bietet keinen Zugriff auf Konstruktoren realisierender Klassen und keine Möglichkeit zur Erzeugung von Realisierungen,[15] weil nur Methoden deklariert werden.
o erfordert die Realisierung öffentlicher Methoden – nämlich derjenigen im Interface.
o erhöht leicht die Komplexität. Die Abstraktion von Funktionalität über ein Interface wird häufig erst dann benötigt, wenn es mehr als eine Realisierung gibt (oder sicher geben wird) oder spezielle Funktionalitäten extrahiert werden sollen. Im Speziellen sind Interfaces und die damit einhergehende etwas höherer Komplexität sinnvoll, wenn eine saubere Trennung und Abstraktion über Package- oder Modulgrenzen erreicht werden soll.

[15] Diese Einschränkung ist in vielen Fällen eher ein Vorteil als ein Nachteil. Außerdem können mithilfe der Entwurfsmuster ERZEUGUNGSMETHODE und FABRIKMETHODE (vgl. Abschnitte 18.1.1 und 18.1.2) oder aber auch Frameworks wie Spring entsprechende Realisierungen des Interface erzeugt werden.

3.2.2 Basisklassen und abstrakte Basisklassen

In diesem Abschnitt schauen wir kurz darauf, warum und wie man Basisklassen einsetzt. Zudem motiviere ich mögliche Einsatzgebiete für abstrakte Basisklassen.

Basisklassen und ihre Vorteile

Bekanntermaßen führt man Basisklassen beim Entwurf dann ein, wenn man erkennt, dass Klassen eine oder mehrere Gemeinsamkeiten besitzen. Diese lagert man dann in eine Basisklasse aus. Dabei durchläuft man in etwa folgende vier Schritte, um gemeinsame Funktionalität in einer Basisklasse zusammenzuführen:

1. Identifiziere potenzielle Klassen als Kandidaten mit genügend Gemeinsamkeiten.
2. Erstelle eine Hierarchie mit gemeinsamer Basisklasse (wodurch die Klassen dann zu Subklassen werden).
3. Lagere die Gemeinsamkeiten geeignet in die Basisklasse aus.
4. Entferne diese Gemeinsamkeiten aus den Subklassen.

Dieses Vorgehen ist insofern praktisch, weil es Sourcecode-Duplikation vermeidet, die ohne gemeinsame Basisklasse durch die mehrfache Realisierung von Funktionalität in den jeweiligen Klassen entstehen würde. Ein weiterer Grund für die Einführung einer gemeinsamen Basisklasse ist, dass man mehrere Klassen einheitlich behandeln möchte, ohne die Details der Spezialisierungen kennen zu müssen. Die gemeinsame Basisklasse definiert dann die Schnittstelle und auch das gemeinsame, standardisierte Verhalten.

Am Beispiel von grafischen Figuren lässt sich das Ganze gut verdeutlichen. Stellen wir uns eine Applikation vor, die Figuren zeichnen soll. Diese werden jeweils als eigenständige Klassen modelliert, die als Basisfunktionalität Methoden zum Zeichnen anbieten. Ohne Abstimmung über die zu verwendenden Methodennamen und mehrere an der Implementierung beteiligte Entwickler entstehen schnell leicht unterschiedliche Namen, etwa neben `draw()` auch die Varianten `drawLine()`, `drawRect()` usw. Das ist für einige Figurenklassen in Abbildung 3-13 gezeigt.

Abbildung 3-13 *Klassen für grafische Figuren (ohne Basisklasse)*

Durch die unterschiedlichen Signaturen sowie die nicht existente gemeinsame Basisklasse wird die Handhabung der Figuren für Klienten umständlich und nicht intuitiv: Die Methodennamen unterscheiden sich von Klasse zu Klasse. Um diesen Missstand zu beheben, führen wir eine Basisklasse `Figure` mit einer `draw()`-Methode ein und vereinheitlichen in den Subklassen die Methodennamen, was für Konsistenz sorgt. Zudem lässt sich die von Subklassen gemeinsam genutzte Funktionalität in der Basisklasse zentral definieren.

Einsatzgebiet abstrakter Basisklassen

Bei der Implementierung der Basisklasse `Figure` stoßen wir jedoch auf ein Problem: Wie sollen wir die Methode `draw()` implementieren? Und: Was soll diese zeichnen? Diese Fragen stellen sich, weil die Klasse das abstrakte Konzept einer Figur modelliert, aber keine konkrete Ausprägung. Ohne Kenntnis abstrakter Basisklassen und abstrakter Methoden würde man wohl die `draw()`-Methode mit einem leeren Methodenrumpf implementieren, weil die Basisklasse `Figure` nichts Sinnvolles zeichnen kann. Dies ist aber wenig hilfreich,[16] da es natürlicher ist, *eben keine* Implementierung anzubieten, wenn eine Methode, wie hier `draw()`, nicht zweckmäßig realisiert werden kann.

Genau für solche Fälle eignen sich abstrakte Basisklassen. Diese erlauben es, Methodensignaturen vorzugeben, die von Subklassen zu implementieren sind. Das geschieht mithilfe abstrakter Methoden, die durch das Schlüsselwort `abstract` gekennzeichnet sind, keine Implementierung besitzen und somit durch Subklassen realisiert werden müssen. Im Gegensatz zu den Methoden in Interfaces können abstrakte Methoden neben der Sichtbarkeit `public` auch die Sichtbarkeit `protected` besitzen. In letzterem Fall blähen sie die öffentliche Schnittstelle nicht auf und können eine Art internes Interface für Subklassen vorgeben.

Im Beispiel wird die Klasse `Figure` zu einer abstrakten Klasse und in `BaseFigure` oder alternativ `AbstractGraphicsFigure` umbenannt.[17] Die Realisierung des Zeichnens erfolgt dann in den Subklassen, die genau wissen, wie die durch sie repräsentierten Figuren gezeichnet werden. Abbildung 3-14 zeigt das Klassendiagramm.

Abbildung 3-14 *Grafische Figuren mit abstrakter Basisklasse*

Eben haben wir gesehen, dass im Rahmen einer Generalisierung derart allgemeine Klassen entstehen, dass dort nicht jede Methode implementiert werden kann. In diesem Fall spricht man von ***abstrakten Basisklassen***. Davon abgeleitete Klassen können dann die abstrakten Methoden passend realisieren. Ist eine Klasse vollständig implementiert, so spricht man von einer ***konkreten Klasse*** bzw. einer ***konkreten Realisierung***.

[16] Eine solche Leerimplementierung kann dann nützlich sein, wenn diese Methode nicht zwangsläufig von Subklassen mit spezifischer Funktionalität versehen werden muss.

[17] Beide Namen sind valide und besitzen Vor- und- Nachteile. Mit `BaseFigure` werden Signaturen besser lesbar – das Präfix `Abstract` ist dort eher störend, weil es suggerieren würde, dass Instanzen abstrakter Klassen übergeben würden. Allerdings wird durch Namen wie `AbstractGraphicsFigure` kommuniziert, dass man keine Instanz erzeugen kann.

Es gibt die Besonderheit einer abstrakten Klasse, die keine abstrakten Methoden besitzt. Eine solche Klasse wird explizit über das Schlüsselwort `abstract` speziell gekennzeichnet, sodass die Instanziierung dieser Klasse verhindert wird.

Bewertung

Abstrakte Basisklassen ...

+ helfen bei einer konsistenten Implementierung von Funktionalitäten. Subklassen können nur an dafür vorgesehenen Stellen ihre Realisierungen einbringen.
+ verringern Sourcecode-Duplikation und dadurch bedingten Wartungsaufwand.
+ fördern die Erweiterbarkeit, weil beim Erzeugen neuer Subklassen in der Regel nur wenige Anpassungen erfolgen müssen.
- haben den Nachteil, dass sie dazu verleiten, Vererbung unüberlegt einzusetzen und dadurch eine tiefe Klassenhierarchie zu erzeugen. Oftmals ist der Einsatz von Vererbung jedoch nicht der richtige Designweg: Vererbung sollte nur dann eingesetzt werden, wenn eine »is-a«-Beziehung vorliegt (vgl. Abschnitt 3.3.2).

3.2.3 Interfaces und abstrakte Basisklassen

Die Kombination von Interfaces und abstrakten Basisklassen bietet sich an, um die Vorteile beider Techniken zu vereinen, etwa wenn einige Klassen einer Klassenhierarchie bereits über ein Interface abstrahiert werden und diese Klassen gemeinsame Funktionalität besitzen, weshalb sich eine Basisklasse empfiehlt.

Beispiel

Wir greifen das Beispiel der grafischen Figuren wieder auf und stellen uns dabei vor, dass wir ein Framework für grafische Applikationen erstellen wollen. Den Vorteil der Kombination von Interfaces und abstrakten Basisklassen kann man sich wie folgt verdeutlichen. Mithilfe des Interface `IFigure` wird die von den Figuren bereitgestellte Funktionalität festgelegt und durch die Klasse `AbstractGraphicsFigure` werden gewisse Funktionalitäten bereits realisiert (vgl. Abbildung 3-15).

Gebräuchliche Anwendung in der Praxis

Neben dem beschriebenen Anwendungsfall bietet sich eine Kombination aus Interface und abstrakter Basisklasse insbesondere dann an, wenn Klienten aus anderen Packages auf die Funktionalität des eigenen Packages zugreifen und man eine lose Kopplung erreichen möchte. Dabei verfolgt man das Ziel, den externen Klassen keinen Zugriff auf die Details der internen Klassen zu gewähren. In diesem Fall sollte man die Funktionalität für Aufrufer über Interfaces bereitstellen und dazu für alle extern genutzten

Abbildung 3-15 *Einfache Kombination von Interface und abstrakter Basisklasse*

Klassen jeweils ein öffentliches Interface anbieten.[18] Zudem definiert man die meisten Klassen mit der Sichtbarkeit Package-private, um deren Implementierung für Package-externe Klassen unzugänglich zu machen und diese nach außen zu verbergen. So hält man die Klienten unabhängig von den Details realisierender Klassen und erzielt eine lose Kopplung zwischen den Klassen in unterschiedlichen Packages.

Nehmen wir an, dass innerhalb des Packages mehrere Varianten der Realisierung eines Interface bereitgestellt werden sollen. Wenn diese Implementierungen alle gemeinsame Funktionalität besitzen, dann empfiehlt es sich, zusätzlich zum Interface eine abstrakte Basisklasse einzusetzen – natürlich ist auch möglich, dass es einzelne Realisierungen des Interface geben kann, die sich nicht auf die abstrakte Basisklasse stützen. Diese Interna bleiben für Klienten transparent, da diese lediglich über das Interface kommunizieren. Abbildung 3-16 zeigt diesen Fall für ein Interface `ServiceIF`, eine abstrakte Basisklasse `AbstractBaseService` sowie zwei konkrete Realisierungen.

Abbildung 3-16 *Komplexere Kombination von Interface und abstrakter Basisklasse*

[18] Teilweise möchte man Interfaces auch explizit zusammenfassen, um Aufrufern die Handhabung zu erleichtern. In Abschnitt 18.2.1 wird das für das Entwurfsmuster FASSADE diskutiert.

Bewertung

Eine Kombination von Interfaces und abstrakter Basisklasse ...

+ ermöglicht sowohl eine Kapselung der Funktionalität über Interfaces sowie eine sinnvolle Vorgabe von Basisfunktionalität durch eine abstrakte Basisklasse.
+ ermöglicht, dass Subklassen nur noch an speziellen, gewünschten Stellen ihre Erweiterungen einbringen müssen.
+ erlaubt es oftmals, die konkreten Implementierungen austauschen zu können.
+ ist insbesondere für Frameworks sinnvoll und empfehlenswert. Ein schönes Beispiel ist das Collections-Framework aus dem JDK, wo dieses Schema sauber umgesetzt wird.
o erfordert nur einen geringen Mehraufwand in der Implementierung.
o erhöht allerdings (geringfügig) die Komplexität. Ein Einsatz sollte demzufolge nicht ohne Grund und insbesondere eher selten innerhalb eines Packages erfolgen, da die Nachvollziehbarkeit durch die Indirektionen und Abstraktionen leiden kann.

3.3 Wissenswertes zu Vererbung

Wir haben nun schon einiges über den OO-Entwurf und mögliche Fallstricke gelernt. Bisher haben wir vor allem die positiven Seiten von Vererbung betrachtet. Dass diese auch Nachteile besitzt, habe ich bereits angedeutet. Es erfordert einige Erfahrung, diese Probleme zu erkennen, zu verstehen und zu beheben.

Beim Erweitern einer bestehenden Klassenhierarchie können Schwierigkeiten auftreten, wenn man dazu unüberlegt Vererbung einsetzt. Dies diskutiert Abschnitt 3.3.1. Werden Eigenschaften oder verschiedene feingranulare Funktionalitäten in Form einzelner Subklassen definiert, kann eine Modellierung über Vererbung problematisch sein und besser mithilfe von Delegation geregelt werden. Das stellt Abschnitt 3.3.2 vor.

3.3.1 Probleme durch Vererbung

In diesem Abschnitt möchte ich anhand eines Beispiels darstellen, welche Herausforderungen beim Erweitern einer bestehenden Klassenhierarchie auftreten können.[19]

Nehmen wir an, wir hätten bereits eine erste Programmversion einer Simulation einer biologischen Umwelt mit verschiedenen Lebensräumen als Klassen implementiert. Dazu sei eine Klasse `Simulation` definiert. Eine weitere Klasse `Environment` beschreibt die Lebensräume für verschiedene zu simulierende Tierarten. Diese werden mithilfe der abstrakten Basisklasse `Animal` realisiert. Als Tierarten sind dann beispielsweise Füchse, Hasen u. v. m. als konkrete Subklassen (`Fox`, `Rabbit` usw.) modelliert.

[19] Die Grundidee des Beispiels habe ich dem Buch »Einstieg in Java 6« von Bernhard Stephan [70] entnommen.

Da diese Tiere als Gemeinsamkeit alle eine Größe, ein Gewicht usw. besitzen, sind diese Eigenschaften durch Attribute in der Basisklasse `Animal` beschrieben. Des Weiteren bewegen sich unsere Tierarten durch Laufen fort. Dies wird daher durch eine Methode `walk()` realisiert. Abbildung 3-17 zeigt das zugehörige Klassendiagramm.

Abbildung 3-17 *Die Klasse* `Animal` *und ihre Vererbungshierarchie*

Zur Simulation einer Umwelt werden diverse Spezialisierungen der Klasse `Animal` erzeugt und danach wiederholt deren Methoden, etwa `walk()`, aufgerufen. Wie so häufig in der Praxis anzutreffen, kommen beim Betrachten und Testen der ersten Version einer Software neue Ideen und dadurch Änderungs- oder Erweiterungswünsche auf. Zeigen wir das Programm einem Ornithologen, so fällt diesem auf, dass unsere Simulation ohne Vögel unvollständig ist. Ein Meeresbiologe könnte etwa einwenden, dass Fische unberücksichtigt sind. Beide Erweiterungswünsche führen in der ursprünglichen Modellierung zu folgenden Problemen:

1. Vögel können zwar in der Regel laufen, aber ihre natürliche Fortbewegungsart ist meistens das Fliegen – Ausnahmen sind z. B. die Vogelarten Strauß und Pinguin.
2. Für Fische ist die Methode `walk()` eine Fehlmodellierung. Wenn man in dieser Methode eine schwimmende Fortbewegung realisieren würde, wäre das Design unverständlich. Nachfolgende Bearbeiter der Klasse werden darunter leiden und Sie möglicherweise dafür verfluchen.

Umgang mit Erweiterungen

Sollen nun die genannten Erweiterungen umgesetzt werden, so müssen wir dabei die zuvor genannten Kritikpunkte beachten. Damit das Klassendesign sauber und logisch strukturiert bleibt, wird die Basisklasse `Animal` wie folgt geändert: Die Basisklasse erhält eine allgemeinere Methode `move()` zur Modellierung der Fortbewegung. Diese nimmt keine Festlegung der tatsächlichen Bewegungsart vor.[20] Außerdem werden nun weitere Basisklassen einer zweiten Gliederungsebene erstellt, etwa die Klassen `Fish`,

[20] Bemerkenswerterweise gibt es sogar einige Tierarten, die sich nicht fortbewegen, etwa Schwämme und Korallen. Daran sieht man, wie schwierig oder teilweise gar nicht realisierbar es ist, ein möglichst allgemeingültiges Modell zu definieren.

`Mammal` und `Bird` zur Modellierung der Tierkategorien Fisch, Säugetier und Vogel. In diesen Basisklassen können wir jeweils die Methoden `swim()`, `walk()` und `fly()` zur Fortbewegung gemäß der Tierkategorie definieren, ohne diese Methoden bereits auf dieser Abstraktionsebene implementieren zu wollen. Das liegt daran, dass die Art der Fortbewegung von einer konkreten Tierart abhängt. Für Klienten soll es nicht von Interesse sein, dass diese Zwischenschicht von Basisklassen in das Design eingezogen wurde. Vielmehr ist es das Ziel, eine Abstraktion von den konkreten Klassen mithilfe der Methode `move()` zu gewährleisten. Basierend auf dieser Argumentation müssen die Methoden `swim()`, `walk()` und `fly()` in ihrer Sichtbarkeit eingeschränkt werden, aber auch ein Überschreiben in Subklassen erlauben, was die Sichtbarkeit `protected` ermöglicht. Das resultierende Klassendiagramm ist in Abbildung 3-18 dargestellt.

Abbildung 3-18 *Variante 2 der Klasse* `Animal` *und ihrer Vererbungshierarchie*

Obwohl diese Modellierung bereits deutlich besser als die ursprüngliche Umsetzung ist, gibt es immer noch einige Schwachstellen:

1. Wie zuvor angemerkt, können Vögel häufig nicht nur fliegen, sondern auch laufen bzw. hüpfen. Außerdem können einige Vogelarten überhaupt nicht fliegen. Allerdings bietet die Basisklasse `Bird` derzeit nur die Methode `move()`, die per Standard das Fliegen modelliert. Wollte man damit auch Laufvögel, wie Sträuße mit der Klasse `Ostrich`, adäquat modellieren, so hätte man ein Problem. Eine mögliche Lösung bestünde darin, die Vererbungshierarchie in die zwei Basisklassen `FlyingBird` und `NonFlyingBird` aufzuspalten. Damit sorgt man zwar für mehr Unterscheidbarkeit, aber es lässt sich immer noch nicht abbilden, dass die meisten Vogelarten sowohl (bevorzugt) fliegen als auch laufen oder hüpfen können. Zudem entsteht dadurch bereits eine recht tiefe Vererbungshierarchie. Eine solche erschwert häufig Wartungsarbeiten und Änderungen. Und die Realität ist noch komplexer: Wie integrieren wir die vorwiegend schwimmenden und tauchenden Pinguine und die korrespondierende Klasse `Penguin`?

2. Auch die Säugetierklasse `Mammal` wirft ein Modellierungsproblem auf: Solange man damit ausschließlich Landsäugetiere modelliert, ist alles in Ordnung. Was ist aber mit Walen? Schließlich kann ein Wal nicht laufen. Müssen wir hier wiederum ein `WalkingMammal` und ein `SwimmingMammal` einführen? Je nach Anwendungsfall ist das wohl notwendig. Und es würde noch komplexer, wenn wir Fledermäuse als fliegende Säugetiergattung simulieren müssten.

Anhand dieser Beschreibung wird klar, dass die objektorientierte Modellierung mit Vererbung auch ihre Tücken besitzt. Im Besonderen gilt, dass in Basisklassen nur die Funktionalitäten realisiert werden sollten, die allen möglichen Subklassen gemein sind.

Wissenswertes zu Vererbung und Wiederverwendbarkeit

An diesem Beispiel erkennen wir, dass Vererbung nicht immer eine einfache Wiederverwendung ermöglicht, wie dies leider viel zu häufig in der Einsteigerliteratur zum objektorientierten Programmieren dargestellt wird. Vielmehr wird deutlich, dass Vererbung gut überlegt eingesetzt werden sollte, da sie ansonsten Probleme bereiten kann: Schnell entsteht eine schlechte Klassenstruktur, die einem die Möglichkeiten raubt, die Vererbung sonst bietet. Betrachten wir das Ganze im Folgenden detaillierter.

Der korrekte Zuschnitt von Basisklassen ist stark von dem zu modellierenden System abhängig. Für unsere Simulation einer biologischen Umwelt gilt z. B.: Spielen Vögel oder Fische eine Rolle? Müssen Sonderfälle wie Laufvögel berücksichtigt werden? Aufgrund der unterschiedlichen Anforderungen ist eine Wiederverwendbarkeit durch Vererbung zwischen verschiedenen Applikationen daher nur recht selten möglich. Innerhalb einer Applikation kann durch Vererbung allerdings eine Duplizierung von Funktionalität vermieden werden. Oftmals ist der Mehrwert der Wiederverwendung jedoch weniger ausgeprägt, als er gerne dargestellt wird.

Des Weiteren ziehen Änderungen in Basisklassen teilweise Anpassungen in deren Subklassen nach sich. Das hat weitreichende Konsequenzen: Wurden Basisklassen eingangs ungünstig gewählt, so können sich Korrekturen daran in vielen Subklassen auswirken. Das lässt für das OO-Design wiederum folgende Schlüsse zu:

- ***Basisklassen müssen mit besonderer Sorgfalt entworfen werden.*** Das ist dadurch begründet, dass Änderungen in Basisklassen ansonsten diverse Folgeänderungen in deren Subklassen verursachen. Änderungen in den Subklassen besitzen dagegen meist recht lokale und gut überschaubare Auswirkungen.
- ***Umfangreiche (tiefe oder breite) Ableitungshierarchien sind zwar ein Indikator für eine gute Wiederverwendung.*** Einher geht aber – wie gerade besprochen – auch eine erhöhte Anfälligkeit für Folgeänderungen in Subklassen bei Änderungen in deren Basisklassen. ***Deswegen ist eine komplexe Ableitungshierarchie immer auch kritisch zu betrachten und zu hinterfragen***. Im Beispiel erkennt man, dass trotz der gemeinsamen Basisklassen tatsächlich recht wenig Funktionalität wiederverwendet werden konnte. Stattdessen muss durch die Betrachtung der ganzen Sonderfälle fast alle Funktionalität in den Subklassen realisiert werden.

Hinweis: Vererbung, Objektmodellierung und Wiederverwendbarkeit

Ein Modell stellt eine Vereinfachung (Abstraktion) eines realen Gegenstands oder Systems dar und definiert die für den jeweiligen Anwendungsfall benötigten Aspekte. Daran orientiert sich der Entwurf von Klassen mit ihren Attributen und Methoden. Es kommt leicht zu dem Missverständnis und der unerfüllbaren Hoffnung, die für einen Anwendungsfall modellierten (Basis-)Klassen für alle möglichen Applikationen wiederverwenden zu können. Das ist aber nur eher selten möglich, da die jeweiligen Applikationen in der Regel andere Verhaltensweisen und Attribute der modellierten Objekte benötigen.

Betrachten wir das konkret für die Modellierung einer Person als Klasse. Für einen Onlineshop sind Name, Anschrift und eventuell Bonität wichtige Kriterien. In einer Patientenverwaltung eines Hausarztes werden neben Name und Anschrift eher Patientendaten, wie Größe, Gewicht, Vorerkrankungen usw., als wichtige Aspekte der Modellierung berücksichtigt werden. Für eine Klasse `Person` gelten in den beiden genannten Systemen also deutlich unterschiedliche Anforderungen. Diese mit nur einer Klasse zu beschreiben, wäre wahrscheinlich nicht sinnvoll. Allerdings könnte man sich eine Modellierung vorstellen, die eine gemeinsame Basisklasse `Person` aus den Anforderungen extrahiert. Diese bekommt die für beide Anwendungen gebräuchlichen Attribute wie Name und Anschrift. Für den Onlineshop könnte man eine Subklasse `Kunde` von der Basisklasse `Person` ableiten und weitere Attribute, etwa die oben geforderte Bonität, hinzufügen. Ebenso würde man bei der Patientenverwaltung vorgehen. Dort könnte eine Subklasse `Patient` als Spezialisierung der Basisklasse `Person` definiert und um Attribute zu dem Patienten sowie seinen Krankendaten ergänzt werden.

Abstrakte vs. konkrete Basisklassen

Das Beispiel zur Modellierung der Tierkategorien und die durch unsere Erweiterungen entstandenen Basisklassen werfen die folgende Fragestellung auf: Wann sind Basisklassen abstrakt zu definieren und wann nicht? Meistens sollten Basisklassen abstrakt definiert werden. In diesem Beispiel ist das für die Basisklasse `Animal` geschehen. Vermutlich empfiehlt sich dies ebenfalls für die Basisklassen `Fish`, `Mammal` und `Bird`. Im Speziellen hängt das davon ab, ob unsere Simulation tatsächlich Objekte dieser Basistypen erzeugen und in der Simulation verwenden können soll. Als Faustregel gilt, dass Basisklassen idealerweise so lange abstrakt definiert werden, wie durch sie lediglich allgemeine Konzepte modelliert werden und noch keine konkreten Implementierungen für alle Methoden vorgegeben werden können.

Da die Simulation mit einzelnen Individuen spezieller Subklassen der Tierkategorien arbeiten soll, führen wir einen letzten Korrekturschritt an unserem Modell durch: Wir definieren die Basisklassen `Fish`, `Mammal` und `Bird` abstrakt (Abbildung 3-19).

Abbildung 3-19 *Variante 3 der Klasse* `Animal` *und ihrer Vererbungshierarchie*

Dieses Beispiel der Modellierung von Tieren für eine Simulationssoftware zeigt anschaulich, dass es oftmals sinnvoll ist, Basisklassen abstrakt zu definieren. Man verhindert dadurch die Instanziierung von Klassen, die unvollständige Konzepte darstellen. Oder anders formuliert: ***Konkrete Basisklassen sollten sehr aufmerksam betrachtet und nur in gut begründeten Ausnahmefällen verwendet werden.***

Fazit

Nutzt man Vererbung als Instrument der Wiederverwendung ohne viel Nachdenken, so kann dies zu unübersichtlichen Designs führen. Abschließend bleibt die Erkenntnis, dass die früher vielfach als Allheilmittel beim OO-Entwurf angesehene Technik der Vererbung durchaus nicht unproblematisch ist. Tatsächlich ist sie oftmals weit weniger praktikabel zur Wiederverwendung als häufig angenommen bzw. propagiert.

Wie jedes Werkzeug kann man auch Vererbung ungeschickt einsetzen oder Unsinn damit anstellen. Im folgenden Abschnitt betrachten wir weitere Problemfälle tiefer oder breiter Vererbungshierarchien. Ganz allgemein kann man aber bereits Folgendes feststellen: ***Sie sollten einen prüfenden Blick auf Ihre Klassenstruktur werfen, sobald der Ableitungsbaum drei oder mehr Hierarchiestufen erreicht oder einzelne Verzweigungen mehr als etwa zehn Klassen besitzen.*** Im Zusammenhang mit Frameworks (etwa Swing, Spring) sind solche Werte allerdings durchaus normal. Diese Richtwerte werde ich in Abschnitt 19.4.1 bei der Beschreibung verschiedener Metriken aufgreifen.

3.3.2 Delegation statt Vererbung

Betrachten wir im Folgenden zunächst, welche Folgen es hat, wenn man für jede neu zu modellierende Eigenschaft eine neue Klasse definiert, die eine andere Klasse erweitert. Anschließend sehen wir uns an, was passiert, wenn man auf diese Art verschiedene Verhaltensweisen eines Objekts, sogenannte ***Rollen***, modelliert. Eine Person könnte etwa die Rolle Student, aber auch (gleichzeitig) die Rolle Hotelgast innehaben.

Klassenexplosion durch Modellierung von Eigenschaften

Am Beispiel einer Klasse `Window` analysieren wir die Modellierung von Eigenschaften durch Vererbung. Aufgabe ist es, diese Klasse um zusätzliche Funktionalität (Titelleiste, Rahmen usw.) zu erweitern. Eine mögliche Realisierung erstellt für jede Eigenschaft eine Subklasse als Spezialisierung. Dies zeigt Abbildung 3-20.

Abbildung 3-20 *Modellierung von Eigenschaften durch Vererbung*

Diese flache und breite Vererbungshierarchie wirkt zunächst nur unelegant. Problematisch wird dies in dem Moment, in dem man verschiedene Eigenschaften kombinieren möchte. Dadurch werden weitere Ableitungen erforderlich und es kommt zu einer kombinatorischen Explosion der Klassen, die vielfach mit einer missverständlichen Namensgebung einhergeht. In Abbildung 3-21 wird dies verdeutlicht.

Abbildung 3-21 *Klassenexplosion durch Modellierung von Eigenschaften*

Es stellt sich die Frage, ob es tatsächlich Unterschiede bei verschiedenen Ableitungsreihenfolgen gibt: Ist ein `ScrollableFramedWindow` semantisch verschieden zu einem

`FramedScrollableWindow`? Und wenn ja, wo liegen die Unterschiede? Spätestens dann, wenn man sich diese Frage stellt, erkennt man, dass man mit dem Einsatz von Vererbung an die Grenzen einer derartigen Modellierung stößt.

Eine mögliche Lösungsidee ist, statt Vererbung boolesche Variablen zur Auswahl gewünschter Funktionalitäten zu verwenden. Funktionalitäten aus den Spezialisierungen werden dazu in die Basisklasse verlagert. Dort erfolgt dann, abhängig vom jeweiligen booleschen Wert, ein Aufruf an die zu aktivierende Funktionalität. Diese Lösung stößt, ebenso wie der Einsatz von Vererbung, vor allem dann an Grenzen, wenn viele oder komplexe Eigenschaften modelliert werden sollen. Dann degeneriert die Basisklasse zu einer »One-for-All«-Funktionssammlung von Optionen. Die Komplexität der Abfragen steigt zunehmend, insbesondere wenn nur spezielle Kombinationen von Eigenschaften erlaubt sind.

Eine Abhilfe bietet der Einsatz des DEKORIERER-Musters (vgl. Abschnitt 18.2.3). Damit kann Funktionalität dynamisch, ohne Vererbung hinzugefügt und beliebig miteinander kombiniert werden. Man verhindert dadurch einerseits eine »explodierende« Klassenhierarchie und erreicht andererseits eine gute Trennung von Zuständigkeiten: Jede Dekoriererklasse trägt den jeweiligen Funktionsanteil bei. Die Gesamtfunktionalität entsteht aus der Kombination der verwendeten Dekoriererklassen. Allerdings gibt es auch einen Haken: Während mit booleschen Flags Funktionalität sogar dynamisch deaktiviert werden kann, ist dies mit Dekorierern nahezu unmöglich.

Dekorierer eignen sich besonders, wenn die modellierten Eigenschaften orthogonal zueinander, d. h. voneinander unabhängig sind. In der Praxis hängen Eigenschaften und Funktionalitäten häufig voneinander ab und schließen sich im Extremfall sogar gegenseitig aus. Es ist dann Aufgabe des Entwicklers, nur erlaubte Kombinationen von Dekorierern zu verwenden. Dies kann man wiederum durch den Einsatz geeigneter Erzeugungsmuster (vgl. Abschnitt 18.1), etwa FABRIKMETHODE und ERBAUER, lösen.

Klassenexplosion durch Modellierung von Rollen

Anhand der Basisklasse `Person` betrachten wir nun, welche Folgen die Modellierung von Rollen durch Vererbung haben kann. Spezialisierungen implementieren dann mitunter lediglich zeitweilig ausgeübte Rollen. Dadurch ergeben sich mehrere Schwierigkeiten. Einerseits müssen alle möglichen Kombinationen von Rollen im Voraus in Form von Klassen realisiert werden. Andererseits bedingt ein Rollenwechsel den Wechsel des Typs und wirft die Frage auf, wie ein Objekt seinen Typ wechseln soll.[21]

Analysieren wir kurz mögliche Probleme durch die Kombination unterschiedlicher Rollen. Bei der Umsetzung durch Vererbung kann es, wie zuvor bei der Modellierung von Eigenschaften, zu einer kombinatorischen Explosion von Klassen kommen. Nehmen wir beispielsweise eine Person, die studiert und in einem Café als Kellner arbeitet. Ein studierender Café-Kellner, der in einem Hotel übernachten möchte, ist durch Vererbung kaum sinnvoll abzubilden. Man stelle sich vor, was passiert, wenn das Studium beendet oder der Job gewechselt wird. Man möchte sicher nicht noch weitere Klassen

[21] Es gibt Programmiersprachen, wo dies möglich ist. In Java geht das jedoch nicht.

einführen, um auch diese neuen Rollen mit allen bisherigen Kombinationen darstellen zu können. Abbildung 3-22 deutet eine kombinatorische Explosion an.

Abbildung 3-22 *Klassenexplosion durch Modellierung von Rollen mit Vererbung*

Es ist wahrscheinlich, dass einige Kombinationen von Rollen widersprüchlich sind, etwa die eines zu Fuß gehenden Radfahrers. Im Extremfall passen dann einige Methoden einer Spezialisierung nicht mehr in das Konzept. Es kann kein sinnvolles Verhalten als Methodenimplementierung definiert werden. In einem solchen Fall kann zwar eine `java.lang.UnsupportedOperationException` ausgelöst werden, allerdings ist das meistens nur eine Notlösung und weist auf ein vorliegendes Designproblem hin. Dann sollte man die Klassenhierarchie überdenken, Klassen und Interfaces anders schneiden oder die »unerfüllbare« Methode in ein separates Interface auslagern.

Da mit Vererbung lediglich statische Eigenschaften ausgedrückt werden können, muss jede einzelne Rolle und jede Kombination als Klasse realisiert werden. Um unterschiedliche Rollen auszuführen, muss ein Objekt als Folge eines Rollenwechsels auch seinen Typ ändern. Das kann ein Objekt jedoch nicht selbst durchführen. Es wird demnach eine steuernde Einheit benötigt, die die passenden Instanzen erzeugt und verwaltet. Eine weitere Schwierigkeit besteht darin, die Instanz einer neuen Rolle weiterhin mit der ursprünglichen Instanz in Verbindung zu bringen.

Wie man sieht, lauern einige Fallstricke bei der Modellierung von Rollen durch Vererbung. Im Gegensatz zur Realisierung von Eigenschaften ist als Abhilfemaßnahme der Einsatz des DEKORIERER-Musters ungünstig, da zusätzliche Funktionalität in diesem Fall nicht transparent hinzugefügt, sondern explizit angesprochen werden soll. Es bietet sich ein anderes Vorgehen zur Lösung an: Die einzelnen Vererbungsbeziehungen werden jeweils analysiert und können anhand der Regeln aus folgender Aufzählung umgewandelt werden.

Eine Vererbungsbeziehung zwischen Klassen, die ...

- sich mit »***can-act-like***« beschreiben lässt, wird durch das Einführen eines speziellen ***Interface*** aufgelöst, das die statische Rolle oder Funktionalität abbildet, die zur Kompilierzeit bekannt ist. Bei Bedarf kann dann per Delegation auf eine Klasse zugegriffen werden, die die Rolle implementiert.

- eine dynamische Eigenschaft, also eine Rolle, modelliert, lässt sich durch Vererbung nur schwierig adäquat ausdrücken. Auch das Implementieren eines Interface ist unpassend. Stattdessen wird die Beziehung umgekehrt und mit ***Delegation*** wird auf die ursprüngliche Basisklasse und deren Funktionalität zugegriffen.

Beide Möglichkeiten werden in Abbildung 3-23 dargestellt. Die Studentenrolle modellieren wir als statische Rolle und verwenden ein Interface `IStudent`. Das Agieren als `Kellner` oder `Gast` wird dynamisch als Rolle abgebildet. Dabei können Rollen entweder als gemeinsame Basisklassen `Kellner` oder direkt als eigene Klasse modelliert werden, hier für die Rollen `CafeGast` und `HotelGast` gezeigt.

Abbildung 3-23 *Delegation statt Vererbung*

Bewertung

Der Einsatz von Delegation ...

+ erlaubt, dass Objekte ihr Verhalten zur Laufzeit ändern. Ableitung bietet nur eine statische Verbindung zwischen Klassen und durch diese mögliche Rollen.
+ verhindert, dass Klassen aus dem modellierten System von rein technischen Utility-Klassen erben. Nutzt man Vererbung statt Delegation, so kann es zu Problemen bei Veränderungen in Basisklassen kommen: Mögliche Änderungen an Utility-Klassen wirken sich dann im modellierten System aus. Das zeugt von schlechtem Design.
- erzeugt für jede delegierte Methode etwas Overhead. Je nach Anzahl der Methoden kann der Sourcecode sich dadurch ziemlich »aufblähen«. Man implementiert im Extremfall fast ausschließlich Delegationen an Methoden anderer Objekte; durch das Objekt selbst wird nahezu kein eigenes Verhalten mehr realisiert.

3.4 Fortgeschrittenere OO-Techniken

Dieser Abschnitt wendet die bereits vorgestellten grundlegenden OO-Techniken an, um nützliche Bausteine für den Entwurf größerer Softwaresysteme zu realisieren. Ein Baustein sind sogenannte READ-ONLY-INTERFACEs, die einen ausschließlich lesenden Zugriff auf Objektdaten erlauben. Das ist Thema von Abschnitt 3.4.1. Der Schutz vor Modifikationen wird mit der Technik der IMMUTABLE-KLASSE logisch fortgeführt und sichert die Unveränderlichkeit des Objektzustands. Darauf geht Abschnitt 3.4.2 ein. Eigenschaften von Klassen lassen sich durch sogenannte MARKER-INTERFACEs beschreiben, die in Abschnitt 3.4.3 erläutert werden.[22] Aufzählungen gruppieren verschiedene semantisch zusammengehörende Werte. Einige Realisierungsmöglichkeiten werden in Abschnitt 3.4.4 diskutiert. Abschließend stellt Abschnitt 3.4.5 sogenannte VALUE OBJECTs (oder auch DATA TRANSFER OBJECTs genannt) vor, die mehrere Attribute oder Parameter zu neuen Einheiten gruppieren, wodurch sich Parameterlisten übersichtlich halten und Aufrufe vereinfachen lassen.

3.4.1 Read-only-Interface

Das mit der Technik READ-ONLY-INTERFACE verfolgte Ziel ist die Definition eines separaten Interface, das ausschließlich Lesezugriff auf die Attribute einer Klasse anbietet. Im Prinzip ist diese Technik nur ein Spezialfall eines Interface, und zwar genau mit der Einschränkung auf Lesezugriffe, um den Objektzustand vor Modifikationen zu schützen. Ich nutze gern das Postfix `RO`, um die Eigenschaft read-only explizit auszudrücken. Abbildung 3-24 stellt dies für die Klasse `Figure` und das Interface `IFigureRO` dar.

Abbildung 3-24 *Read-only-Interface und zugehörige Implementierung*

[22] Heutzutage wird man bei Neuentwicklungen eher auf Annotations setzen, Marker-Interfaces findet man jedoch noch vielfach im JDK, weshalb sie hier behandelt werden.

Bewertung

Ein Read-only-Interface ...

+ hilft, Änderungen am Objektzustand durch andere Klassen zu verhindern. Dies ist allerdings lediglich für Attribute primitiver Datentypen sicher möglich. Bei Rückgabe von Referenzvariablen besteht die Gefahr unerwarteter Änderungen. Darauf geht der folgende Abschnitt detailliert ein.
+ kann immer dann sinnvoll sein, wenn eine Klasse über Package-Grenzen hinweg ausschließlich lesend angesprochen werden soll.
+ sorgt für geringere Komplexität und höhere Robustheit, da Änderungen nur innerhalb der Klasse oder des Packages erfolgen.
o kann Modifikationen nur dann sicher verhindern, wenn ausschließlich Referenzen auf Read-only-Interfaces, unveränderliche Klassen (vgl. Abschnitt 3.4.2) oder aber (losgelöste, tiefe) Kopien angeboten werden.

Probleme trotz Read-only-Interface durch die Referenzsemantik von Java

In Java können `get()`-Methoden neben Werten primitiver Typen auch Referenzen auf Objekte zurückliefern. Für letztere sind beliebige Methodenaufrufe[23] möglich – im Speziellen auch solche, die den Zustand verändern. Liefern `get()`-Methoden Referenzen auf Collections zurück, so ist die Situation noch problematischer, da sowohl die Collections als auch deren Elemente veränderlich sind.

Wir nutzen ein Beispiel, weil das Ganze noch etwas kompliziert klingt. Der Datencontainer `ClubMembers` verwaltet `Person`-Objekte und bietet ein Read-only-Interface `IClubMembersRO` zum Datenzugriff an (vgl. Klassendiagramm in Abbildung 3-25).

Abbildung 3-25 *Probleme trotz Read-only-Interface*

[23] Entsprechende Sichtbarkeiten und Zugriffsrechte vorausgesetzt.

Hierbei lernen wir mit der Definition `List<Person>` ein bisher nicht vorgestelltes Sprachelement von Java kennen: die sogenannten ***Generics***. Zunächst reicht es, zu wissen, dass damit typsichere Containerklassen realisiert werden können: Die Schreibweise `List<Person>` erlaubt in einer so definierten Liste die Speicherung von Referenzen vom Typ `Person` (und auch von Subtypen). Abschnitt 3.7 stellt das Thema Generics ausführlicher vor.

Selbst wenn man – wie hier mit `IClubMembersRO` – ein Read-only-Interface einsetzt, sind Änderungen am Objektzustand der Klasse `ClubMembers` indirekt möglich:

- Die Methode `getMemberByName()` bietet Zugriff auf ein Objekt der Klasse `Person`. Dort können Modifikationen erfolgen.
- Die Methode `getAllMembers()` bietet Zugriff auf die interne Liste.

Probleme in der Zugriffsmethode `getMemberByName()` Die nur lesende Methode `getMemberByName()` liefert eine Referenz auf ein Objekt der Klasse `Person` zurück. Aufrufer können durch einen Aufruf von `set()`-Methoden somit Änderungen an den Werten der Attribute bewirken. Es könnte etwa durch einen Aufruf von `setName()` zu einem unerwarteten Namenswechsel kommen: Aus Herrn Meyer könnte Herr Müller werden usw. Abbildung 3-26 macht dies deutlich.

Abbildung 3-26 *Mögliche Probleme nach Aufruf von* `getMemberByName()`

Probleme in der Zugriffsmethode `getAllMembers()` Die Zugriffsmethode `getAllMembers()` ist problematisch, weil sie Zugriff auf interne Daten der Klasse `ClubMembers` ermöglicht:

```
// Achtung: Problematische Realisierung
public List<Person> getAllMembers()
{
    return members;
}
```

Klienten können mit der zurückgegebenen `List<Person>`-Referenz `members` folglich Änderungen am internen Objektzustand vornehmen und am Interface der Klasse `ClubMembers` vorbei programmieren: Beispielsweise lassen sich `Person`-Objekte einfügen, ohne die dafür vorgesehene Methode `addMember(Person)` aufzurufen. Mithilfe der zurückgelieferten `List<Person>`-Referenz wäre es sogar möglich, die komplette Mitgliederliste durch einen Aufruf der Methode `clear()` zu löschen. Anschließend könnten über den Aufruf von `add(Person)` die Personen »Schulze« und »MisterX« in den Club aufgenommen werden. In Abbildung 3-27 ist dieser Ablauf dargestellt. Die Rückgabe von Collection-Referenzen erlaubt es, dass sich der Objektzustand also (unerwartet) beliebig von außen ändern lässt, ohne dass das `ClubMembers`-Objekt etwas davon »bemerkt«![24] Weiterhin existieren die zuvor angesprochenen Probleme für herausgereichte Referenzen auf `Person`-Objekte.

Abbildung 3-27 *Mögliche Probleme nach Aufruf von* `getAllMembers()`

Lösungsmöglichkeiten für die »Schlupflöcher«

Das vorherige Beispiel zeigt, dass der Einsatz eines Read-only-Interface Änderungen am internen Zustand eines Objekts bei Rückgabe von Referenzen nicht sicher verhindern kann. Das widerspricht jedoch dem verfolgten Ziel der Unveränderlichkeit. Was kann man gegen unbeabsichtigte Änderungen unternehmen?

Wenn man Referenzen zurückliefert, empfiehlt es sich, die Technik der Read-only-Interfaces zu nutzen. Betrachten wir nun, was dies konkret für Objekte und Collections für das vorgestellte Beispiel bedeutet.

[24] Natürlich könnte man diese Veränderung im Objekt selbst feststellen. Hier ist aber gemeint, dass keine Methode der Klasse aufgerufen wird, sondern die Änderungen indirekt erfolgen.

Lösungsmöglichkeiten für Objekte Zunächst wird für die Klasse `Person` ein Read-only-Interface `IPersonRO` eingeführt. Änderungen lassen sich dann – sofern wie hier nur unveränderliche Typen für die Attribute genutzt werden – unterbinden, wenn die Methode `getMemberByName()` dieses Interface zurückliefert. Dadurch erhält ein Aufrufer lediglich eine Read-only-Sicht auf `Person`-Objekte. Abbildung 3-28 zeigt das zugehörige Klassendiagramm.

Abbildung 3-28 *Einführen eines Read-only-Interface* `IPersonRO`

Achtung: Probleme trotz Read-only-Interface durch »böse« Casts

Wenn Klienten böswillig wären, könnten diese durch einen Down Cast auf einen Subtyp aus Read-only-Interfaces wieder `Person`-Objekte referenzieren:

```
// NIEMALS MACHEN, NUR ZUR DEMONSTRATION
final IPersonRO readOnlyPerson = clubMembers.getMemberByName("Müller");
final Person modifiablePerson = (Person)readOnlyPerson;
```

Dieses Beispiel sollte reichen, um zu zeigen, weshalb Casts vermieden werden sollten. Ergänzend sollte man die Sichtbarkeit von Klassen auf Package-private einschränken: Package-externe Klienten besitzen dann keinen Zugriff auf die Klasse, und das verhindert automatisch derartige Missbrauchsversuche.

Einfache Lösungsmöglichkeiten für Collections Für Collections sind mehrere Dinge zu beachten: Zunächst müssen wir verhindern, dass Änderungen an der Collection an sich vorgenommen werden. Oft wollen wir zudem Modifikationen an den gespeicherten Daten möglichst unterbinden. Für Ersteres nutzen wir die Methode `unmodifiableList()` aus dem Collections-Framework (vgl. Abschnitt 6.3.2):

```
public List<Person> getAllMembersAsUnmodifiable()
{
    // Read-only-Collection
    return Collections.unmodifiableList(members);
}
```

Dabei werden die Elemente der ursprünglichen Liste nicht kopiert, sondern es wird eine unveränderliche Sicht auf die übergebene Liste bereitgestellt: Lesezugriffe werden an die ursprüngliche Liste weiter delegiert, Schreibzugriffe lösen Exceptions aus.

Zum Schutz vor Modifikationen kann man alternativ eine Kopie der Collection anfertigen und diese zurückgeben.[25] Aufrufer können dann Änderungen an der Zusammensetzung der Kopie vornehmen, ohne dadurch Rückwirkungen auf die Zusammensetzung der ursprünglichen Collection auszulösen:

```
public List<Person> getAllMembersAsCopy()
{
    // Kopie
    return new ArrayList<Person>(members);
}
```

Diese Variante der Kopie kann vorteilhaft sein, wenn Klienten mit den zurückgelieferten Daten beliebig weiterarbeiten möchten.

Allerdings ermöglichen beide genannten Techniken nur den Schutz der Datenstruktur selbst. Die dort gespeicherten Elemente sind weiterhin veränderlich.

Aufwendigere Lösungsmöglichkeiten für Collections Wollen wir verhindern, dass die Daten verändert werden, so gibt es dafür zwei mögliche Lösungen: Entweder werden für alle Elemente ausschließlich Read-only-Referenzen zurückgegeben oder es werden sogenannte ***tiefe Kopien*** erstellt (siehe folgenden Praxistipp).

Wir werden hier Ersteres umsetzen. Basierend auf dem Read-only-Interface `IPersonRO` für die Klasse `Person`, könnte man intuitiv Folgendes schreiben:

```
public List<IPersonRO> getAllMembers()
{
    // Compile-Error: Cannot cast from List<Person> to List<IPersonRO>
    return (List<IPersonRO>)Collections.unmodifiableList(members);
}
```

So wünschenswert diese Schreibweise auch ist, so führt sie doch zu Problemen bei der Typprüfung und löst einen Kompilierfehler aus. Der Grund ist, dass für Generics eine `List<Person>` nicht zuweisungskompatibel zu einer `List<IPersonRO>` ist. Details werden später in Abschnitt 3.7 behandelt. Hier lässt sich als Abhilfe folgende Methode `getAllMembersRO()` einführen:

```
public List<IPersonRO> getAllMembersRO()
{
    final List<IPersonRO> readOnlyMembers = new ArrayList<>(members);

    // Read-only-Collection auf Read-only-Daten
    return Collections.unmodifiableList(readOnlyMembers);
}
```

[25] Allerdings ist zu bedenken, dass bei sehr umfangreichen Datenstrukturen auch einiges an Speicher belegt und eine entsprechende Anzahl an Referenzen kopiert wird – nicht jedoch die Objekte selbst!

Abbildung 3-29 *Einsatz der Methode* `getAllMembersRO()`

Außerdem kann man die ursprünglich öffentliche Methode `getAllMembers()` in ihrer Sichtbarkeit auf `protected` einschränken. Damit gewährt man Subklassen weiterhin Schreibzugriffe, Klienten können jedoch nur noch lesend über das Interface `IClubMembersRO` zugreifen. Abbildung 3-29 zeigt dies im Kontext der anderen Klassen.

Tipp: Kopierstrategien

Eine sogenannte ***flache Kopie*** (***Shallow Copy***) erzeugt eine exakte, bitweise Kopie eines Originals. Es entstehen sowohl für primitive Elemente als auch für Referenzvariablen Wertkopien. Für primitive Typen ist das nicht problematisch. Bei Referenzen verweisen Kopie und Original so jedoch auf die gleichen Daten, sodass spätere Änderungen am Original folglich auch in der Kopie sichtbar sind (und umgekehrt). Man kann dies vermeiden, wenn man eine sogenannte ***tiefe Kopie*** (***Deep Copy***) erstellt. Dazu werden alle enthaltenen Objekte rekursiv kopiert. Dies ist vom Entwickler selbst zu programmieren und kann aufwendig werden, wenn viele Objekte referenziert werden.

3.4.2 Immutable-Klasse

Ziel beim Einsatz der Technik IMMUTABLE-KLASSE ist es, den Objektzustand nach der Objekterzeugung unveränderlich zu halten. Eine wichtige Voraussetzung besteht darin, alle Attribute `private` und `final` zu deklarieren. Dies drückt aus, dass nach der initialen Zuweisung bei der Konstruktion keine Veränderungen stattfinden sollen, und schützt zudem vor versehentlichen Änderungen innerhalb der Klasse selbst. Dazu müssen allerdings bereits zur Konstruktion alle benötigten Daten vorliegen und an Attribute zugewiesen werden. Dadurch bietet die Klasse dann nur noch lesenden Zugriff auf ihre Daten. Ein Beispiel zeigt Abbildung 3-30.

Abbildung 3-30 *Klassendiagramm für eine Immutable-Klasse*

Wird eine Veränderbarkeit unter gewissen Bedingungen dennoch gewünscht oder benötigt, so kann man dies durch einen Trick erreichen: Statt die Werte der Attribute zu modifizieren, wird ein neues Objekt mit passender Wertebelegung erzeugt.[26]

Am Beispiel der folgenden Klasse `ImmutablePosition` verdeutliche ich das zuvor Beschriebene anhand der Methode `offset(int, int)`. Diese erzeugt für jede Positionsänderung neue `ImmutablePosition`-Objekte:

```
public final class ImmutablePosition
{
    private final int x;
    private final int y;

    public ImmutablePosition(final int x, final int y)
    {
        this.x = x;
        this.y = y;
    }

    public int getX()    { return x; }
    public int getY()    { return y; }

    public ImmutablePosition offset(final int xOffset, final int yOffset)
    {
        return new ImmutablePosition(x + xOffset, y + yOffset);
    }
}
```

Variante mit Java 14 Records Greifen wir der Beschreibung der Neuerungen aus Java 14 in Kapitel 14 etwas vor und betrachten die Records als neues Sprachfeature, mit dem sich eine Immutable-Klasse wie folgt implementieren lässt:

```
record ImmutablePositionJdk14(int x, int y)
{
    public ImmutablePositionJdk14 offset(final int xOffset, final int yOffset)
    {
        return new ImmutablePositionJdk14(x + xOffset, y + yOffset);
    }
}
```

[26] Dies kann durch die Referenzsemantik zu Missverständnissen oder Fehlern führen. Diverse Methoden der Klasse `String`, des bekanntesten Vertreters dieser Technik aus dem JDK, suggerieren durch ihren Namen eine Veränderlichkeit. Dies ist jedoch nicht der Fall, stattdessen liefern die Methoden jeweils neu erzeugte Stringobjekte zurück (vgl. Abschnitt 5.3.1).

Umgang mit veränderlichen gespeicherten Werten

Das einleitende Beispiel lässt vermuten, dass es leicht wäre, eine Klasse unveränderlich zu machen. Für Attribute primitiver Datentypen reicht es tatsächlich bereits aus, diese `final` zu deklarieren und während der Konstruktion zu initialisieren. Es gibt einige Spezialfälle für die dies nicht gilt, etwa immer dann, wenn über die sogenannten »Escaping References«, die wir später in Kapitel 16 kennenlernen werden, andere Objekte bereits Zwischenzustände von Wertebelegungen sehen können.

Werden in Attributen jedoch Referenzen auf andere Klassen, insbesondere auf Containerklassen, gespeichert, so ist es nicht so einfach möglich, Unveränderlichkeit zu erreichen. Die Gründe wurden bereits zuvor in der Beschreibung der Read-only-Interfaces ausführlich dargestellt. Betrachten wir nun, was notwendig ist, um eine Klasse unveränderlich zu machen, wenn Referenzattribute auf veränderliche Klassen existieren.

Schutz von Referenzattributen Nehmen wir an, wir wollten eine Zeitspanne beschreiben. Die folgende Klasse `PeriodOfTime` modelliert diese und nutzt dazu zwei Attribute `start` und `end` vom Typ `java.util.Date`:

```
public final class PeriodOfTime
{
    private final Date start;
    private final Date end;

    public PeriodOfTime(final Date start, final Date end)
    {
        Objects.requireNonNull(start, "start must not be null");
        Objects.requireNonNull(end, "end must not be null");

        if (start.after(end))
            throw new IllegalArgumentException(start + " must be <= " + end);

        this.start = start;
        this.end = end;
    }

    public Date getStart()   {  return start;   }
    public Date getEnd()     {  return end;     }
}
```

Bei dieser Realisierung sind mehrere Dinge erwähnenswert. Beginnen wir mit den Prüfungen der Konstruktorparameter. Dazu wird die Methode `requireNonNull(T, String)` aus der mit JDK 7 eingeführten Utility-Klasse `java.util.Objects` genutzt. Ansonsten scheint diese Implementierung ähnlich zu der zuvor gezeigten Klasse `ImmutablePosition` zu sein und somit dem beschriebenen Muster der Unveränderlichkeit zu folgen. Es gibt aber Unterschiede. Betrachten wir dies im Detail: Die Implementierung der Klasse `PeriodOfTime` prüft im Konstruktor die Gültigkeit der Parameter und damit auch die Einhaltung der gewünschten Invariante $start \leq end$. Die Attribute sind zudem `final`. Allerdings führt dies lediglich zu einer Unveränderlichkeit der `Date`-Referenzen. Aber die dort gespeicherten Werte sind *veränderlich*, da die Klasse `Date` selbst veränderlich ist: Sie besitzt `set()`-Methoden für ihre Attribute.

Das folgende Listing zeigt die Veränderlichkeit exemplarisch anhand des Aufrufs der Methode `setTime(700)` für die Instanz `end`:

```
final Date start = new Date(1000);
final Date end = new Date(2000);
final PeriodOfTime period = new PeriodOfTime(start, end);

// Invariante (start <= end) von PeriodOfTime wird durch setTime(700) verletzt
end.setTime(700);
```

Möchte man Änderungen im Objektzustand sicher ausschließen, so muss man übergebene Referenzen und deren Speicherung in Attributen entkoppeln. Dazu erzeugt man aus den an den Konstruktor übergebenen `Date`-Objekten neue Instanzen wie folgt:

```
public class PeriodOfTime(final Date start, final Date end)
{
    Objects.requireNonNull(start, "start must not be null");
    Objects.requireNonNull(end, "end must not be null");

    if (start.after(end))
        throw new IllegalArgumentException(start + " must be <= " + end);

    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
}
```

Nun sind die Attribute geschützt: Wenn Änderungen an den übergebenen `Date`-Objekten durchgeführt werden, führt dies nicht mehr zu Änderungen im Objektzustand. Es existieren aber weitere Schlupflöcher:[27] Da die Zugriffsmethoden `getStart()` und `getEnd()` Referenzen auf die internen `Date`-Objekte liefern, ist eine Veränderung von außen tatsächlich immer noch möglich. Um auch dies zu unterbinden, müssen die herausgereichten Referenzen wie gezeigt von den internen Daten abgekoppelt werden:

```
public Date getStart()  { return new Date(start.getTime()); }
public Date getEnd()    { return new Date(end.getTime()); }
```

Schutz von Collections Werden Referenzen auf Containerklassen gehalten, so ist zusätzlicher Aufwand nötig, um die Unveränderlichkeit (möglichst weitgehend) herzustellen – eine vollständige Unveränderlichkeit erhält man nur, wenn man tiefe Kopien erstellt. Wir werden am Ende dieses Abschnitts eine schwächere Abkopplung in Form von Kopien der eingehenden Collections realisieren.

Wir betrachten nun den Schutz von Collections und greifen das Beispiel der Klasse `ClubMembers` wieder auf, und zwar konkret für das Attribut `members`, das die Mitglieder des Clubs in Form einer `java.util.ArrayList<Person>`, also als `Person`-Objekte, speichert. Nehmen wir an, es wäre ein Konstruktor hinzugefügt worden:

[27] Implementiert eine Klasse das Interface `java.io.Serializable`, so sind weitere Anstrengungen nötig, um die Unveränderlichkeit sicherzustellen. Details dazu finden Sie in der zweiten Auflage des Buchs »Effective Java« von Joshua Bloch [6].

```
public final class ClubMembersUnsafeCollectionExample
{
    // Sowohl die Referenz als auch die Zusammensetzung sind veränderlich
    private List<Person> members = new ArrayList<>();

    public ClubMembersUnsafeCollectionExample(final List<Person> members)
    {
        this.members = members;
    }
}
```

Diese in der Praxis relativ häufig zu sehende Umsetzung birgt einige Risiken. Leider findet man mitunter darüber hinaus noch eine `set()`-Methode, die ebenfalls eine Referenzzuweisung etwa folgendermaßen durchführt:

```
public void setMember(final List<Person> newMembers)
{
    // Achtung: ungeprüfte Zuweisung!
    this.members = newMembers;
}
```

Wie bereits bei den Read-only-Interfaces erwähnt, wirken sich bei der Speicherung von Referenzen auf Collections spätere Änderungen anderer Objekte auf den eigenen Objektzustand aus. Das ist natürlich für Immutable-Klassen nicht erwünscht. Ähnliches gilt aber meistens auch für »normale« Klassen, weil deren Objektzustand dann von außen veränderlich ist. Es existiert ein weiteres Problem, weil im Konstruktor eine `null`-Referenz übergeben werden kann. Beim späteren Zugriff auf das Attribut `members` kommt es dann zu einem Laufzeitfehler in Form einer `NullPointerException`. Wir korrigieren das Beispiel wie folgt:

```
public final class ClubMembersSafeCollectionExample
{
    // Referenz fix, Inhalt veränderlich
    private final List<Person> members;

    public ClubMembersSafeCollectionExample(final List<Person> newMembers)
    {
        Objects.requireNonNull("parameter 'newMembers' must not be null!");

        // Sorgt für Unabhängigkeit von Änderungen im Original
        this.members = new ArrayList<>(newMembers);
    }
}
```

Im Listing sehen wir, dass die Referenz auf die Liste `final` ist und mit einer `ArrayList<Person>` initialisiert wird. Es erfolgt keine Zuweisung der Referenz der übergebenen Liste, sondern es wird deren Inhalt in eine interne Liste übertragen. Somit wirken sich Änderungen an der Zusammensetzung des Originals nicht im neu konstruierten Objekt aus.

Selbstverständlich darf die Klasse keine `set()`-Methode für die Collection anbieten, und es sind die zuvor beschriebenen Vorkehrungen in der `get()`-Methode nötig.

Bewertung

Immutable-Klassen ...

+ verhindern Änderungen am Objektzustand und sorgen für weniger Komplexität und mehr Robustheit der eigentlichen Anwendung.
+ benötigen keine Synchronisation von Zugriffen auf Attribute bei Multithreading, da nach der Objektkonstruktion keine Wertänderungen mehr erfolgen (können).
- erfordern bei Attributen von Referenztypen und insbesondere von Collection-Typen deutlich mehr Aufwand zum Schutz von gespeicherten Daten vor Veränderungen als bei der Nutzung primitiver Datentypen.
- führen selbst bei kleinen Änderungen am Objektzustand immer zur Konstruktion neuer Objekte. Dadurch kommt es zu ständig wechselnden Referenzen. Falls diese in anderen Klassen zwischengespeichert werden, können wiederum Probleme bzw. Inkonsistenzen durch möglicherweise veraltete Daten auftreten.

3.4.3 Marker-Interface

Im vorherigen Abschnitt haben wir verschiedene Methoden kennengelernt, Eigenschaften oder Verhaltensweisen ohne den Einsatz von Vererbung auszudrücken. Eine weitere Möglichkeit dazu bietet die Technik MARKER-INTERFACE. Mithilfe eines solchen Interface lassen sich Metainformationen über eine Klasse zur Verfügung stellen. Man kann sich dies in etwa wie ein konstantes statisches boolesches Attribut einer Klasse vorstellen, das eine Eigenschaft ausdrückt. Seit JDK 5 sollte man dafür allerdings besser die Technik der Annotations nutzen, die ich später in Abschnitt 12.2 beschreibe.

Um wirklich nur eine Markierung vorzunehmen, enthält ein Marker-Interface weder Methoden noch Konstanten: Es ist also leer. Wenn eine Klasse ein Marker-Interface implementiert, so wird damit gekennzeichnet, dass eine spezielle Eigenschaft erfüllt werden soll. Abbildung 3-31 zeigt dies am Beispiel des Marker-Interface `java.io.Serializable`.

Abbildung 3-31 *Klassendiagramm des Marker-Interface*

Die Abbildung zeigt eine Klasse `SerializableClass`, die das genannte Marker-Interface erfüllt, und deshalb von der in die JVM integrierten Serialisierungsautomatik verarbeitet werden kann, d. h. sich in eine bzw. aus einer Folge von Bytes wandeln lässt.

Die weiterhin gezeigte Klasse `NonSerializableClass` erfüllt das Interface nicht und kann daher von der Automatik nicht verarbeitet werden (vgl. Abschnitt 12.3).

Bewertung

Der Einsatz von Marker-Interfaces ...

+ erlaubt es, markierte Eigenschaften allein basierend auf dem Typ abzufragen, also ohne dass eine Instanz einer Klasse benötigt wird.
o erfordert einen Aufruf von `instanceof`, um die Existenz einer markierten Eigenschaft zu prüfen. Für das Beispiel der Serialisierung wäre etwa eine Abfrage über eine Methode `isSerializable()` klarer.
- garantiert nicht, dass sich eine implementierende Klasse auch entsprechend der angegebenen Eigenschaft verhält, weil es ja nur eine Markierung ist. Bei einem normalen Interface würde man zumindest die Implementierung der dort definierten Methoden durch den Compiler prüfen können.

3.4.4 Konstantensammlungen und Aufzählungen

Unter Aufzählungen versteht man Sammlungen semantisch zusammengehörender Werte. Statt der Definition von Konstanten sieht man manchmal allerdings den Einsatz der jeweils benötigten konkreten Werte als ***Literale***[28]. Man spricht auch von ***Magic Numbers***. Deren Nutzung erschwert in der Regel die Verständlichkeit und Wartbarkeit. Als Abhilfe ist es wünschenswert, die konstanten Werte semantisch zu gruppieren und zentral zu definieren. Dadurch kann man die Konstanten über deren Namen innerhalb von anderen Klassen ansprechen. Dazu sind folgende Realisierungen gebräuchlich:

- Definition einiger Konstanten in der verwendenden Klasse oder in einer separaten Konstantensammlungsklasse
- Definition eines eigenen Aufzählungstyps gemäß dem ENUM-Muster
- Definition eines eigenen Aufzählungstyps mit dem Schlüsselwort `enum`

Seit JDK 5 sollte man bevorzugt das Schlüsselwort `enum` verwenden, um Aufzählungen zu definieren. Da man aber immer noch alle genannten Formen der Realisierung antrifft, thematisiere ich nachfolgend kurz mögliche Auswirkungen ihres Einsatzes.

Konstantensammlungen

Mitunter sieht man eine einfache Umsetzung von Aufzählungen in einer Klasse, die lediglich mit öffentlichen Konstanten arbeitet. In folgendem Beispiel sind in der Klasse `ErrorStateAsIntConstants` einige Fehlerwerte als `int`-Konstanten definiert:

[28] Mit Literalen bezeichnet man Zeichenfolgen, die Werte von Basistypen darstellen, etwa die Zeichenfolge '`123`' als `int`-Literal oder '`true`' als Literal vom Typ `boolean`.

```
public final class ErrorStateAsIntConstants
{
    public static final int OK                = 0;
    public static final int INVALID_POSITION  = 3;
    public static final int INPUT_BUFFER_FULL = 8;

    private ErrorStateAsIntConstants()
    {}    // Vermeide Konstruktion dieser Klasse
}
```

Diese Realisierung weist zwei Nachteile auf. Zum einen sind derart definierte Konstanten weder typsicher noch deren Nutzung überraschungsfrei: Überall, wo die Werte dieser Konstanten zu übergeben sind, könnten beliebige andere `int`-Werte verwendet werden, und nicht nur diejenigen, für die Konstanten definiert sind: Statt der Konstanten `OK` könnte der weniger aussagekräftige, aber entsprechende Zahlenwert `0` genutzt werden.

Aufzählungstypen gemäß dem ENUM-Muster

Um die geschilderten Probleme reiner Konstantensammlungen zu adressieren, wurde das ENUM-Muster erdacht. Hierbei wird ein neuer, eigenständiger Datentyp in Form einer Klasse definiert. Diese dient nicht nur als Namensraum und Sammelstelle für Konstanten, sondern definiert vor allem einen eigenständigen Typ und zugehörige Aufzählungskonstanten als Objekte.

Im folgenden Beispiel werden die zuvor als `int`-Werte implementierten Konstanten nun als statische, öffentliche Objekte vom Typ `ErrorState` definiert. Die Klasse ist `final`, um eine Ableitung zu verbieten. Eine Definition zusätzlicher Konstanten dieses Typs an anderer Stelle als dieser Klasse wird durch den privaten Konstruktor verhindert.

```
public final class ErrorState
{
    public static final ErrorState OK          = new ErrorState(0, "Ok");
    public static final ErrorState INVALID_POS = new ErrorState(3,
                                                  "Ungültige Positionsangabe");
    public static final ErrorState BUFFER_FULL = new ErrorState(8,
                                                  "Empfangspuffer voll");

    private final int              value;
    private final String           description;

    private ErrorState(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }

    public int getValue()               { return value; }
    public String getDescription()      { return description; }
}
```

Zur Modellierung der Konstanten werden hier die Attribute `value` und `description` verwendet. Im Attribut `value` wird der Wert der `int`-Konstante gespeichert und in

`description` ein beschreibender Text. Neben den eingangs genannten Vorteilen besitzen derart definierte Aufzählungen folgende Beschränkungen:

1. **Fehlende Sortierreihenfolge** – Die Klasse `ErrorState` definiert zunächst lediglich eine lose Sammlung von Konstanten ohne eine Ordnung. Vielfach ist dies bereits ausreichend, wenn es nur um Typsicherheit geht. Die Realisierung einer Sortierreihenfolge muss selbst implementiert werden, etwa mithilfe des Interface `Comparable<T>` (vgl. Abschnitt 6.1.8) und der Methode `compareTo(T)`. Dadurch kann eine Reihenfolge für die Konstanten festgelegt werden.
2. **Fehlende Serialisierbarkeit** – Sollen die Konstanten serialisiert werden, so muss die Aufzählungsklasse das Interface `Serializable` (vgl. Abschnitt 12.3) implementieren und zudem einen inhaltlichen Vergleich durch Überschreiben der Methode `equals(Object)` (vgl. Abschnitt 5.1.2) realisieren.

Beide Nachteile können durch etwas Implementierungsaufwand behoben werden. Sinnvoller ist aber der Einsatz des Schlüsselworts `enum`, wie dies nun vorgestellt wird.

Aufzählungstypen mit dem Schlüsselwort `enum`

Werden Aufzählungen mithilfe des Schlüsselworts `enum` definiert, so besitzen die Konstanten die zuvor genannten Nachteile nicht, sondern sind out of the box vergleichbar und serialisierbar, da sie von Hause aus die Interfaces `Comparable<T>` und `Serializable` implementieren. Außerdem besitzen sie eine Reihenfolge gemäß ihrer Definition (dem Auftreten im Sourcecode), die durch die Methode `ordinal()` abgefragt werden kann.

Folgendes Listing zeigt die Definition der Fehlerwerte als `enum`-Aufzählung `ErrorStateEnum`. Diese enthält, wie die zuvor selbst erstellte Lösung in Form einer Klasse, zwei eigene Attribute `value` und `description`:

```
public enum ErrorStateEnum
{
    OK(0, "Ok"),
    INVALID_POS(3, "Ungültige Positionsangabe"),
    BUFFER_FULL(8, "Empfangspuffer voll");

    private final int    value;
    private final String description;

    private ErrorStateEnum(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }

    public int getValue()               { return value; }
    public String getDescription()      { return description; }
}
```

Kombination von Eigenschaften

Bis hierher haben wir lediglich Aufzählungen betrachtet, deren Werte sich gegenseitig ausgeschlossen haben. Zum Teil beschreiben Aufzählungswerte aber Eigenschaften, die kombinierbar sein sollen. Das gilt etwa für Darstellungsattribute eines Zeichensatzes. Eine Implementierung für diese könnte wie folgt aussehen, wenn wir keine `enum`-Aufzählung verwenden, sondern lediglich `int`-Konstanten:

```
public final class FontAttributes
{
    public static final int NORMAL    = 0;
    public static final int BOLD      = 1;
    public static final int ITALIC    = 2;
    public static final int UNDERLINE = 4;

    private FontAttributes()
    {} // Vermeide Konstruktion dieser Klasse
}
```

Zur Kombination von Eigenschaften müssen diese bitweise mit ODER (Operator '`|`') verknüpft werden. Zur Sicherstellung der Eindeutigkeit bei der Kombination von Eigenschaften sind für die Werte Zweierpotenzen zu wählen, die in ihrer binären Darstellung jeweils einem gesetzten Bit entsprechen. Möchte man ermitteln, ob eine Eigenschaft gewählt ist, müssen die Werte durch ein bitweises UND (Operator '`&`') geprüft werden. Folgendes Listing zeigt dies in drei Varianten (V1, V2 und V3):

```
public static void main(final String args[])
{
    final int fontStyles = BOLD | ITALIC ;

    // Variante 1: Prüfe Attribut und zeige Implementierungsdetails
    final boolean isUnderline = (fontStyles & UNDERLINE) == UNDERLINE; // V1

    // Variante 2+3: Prüfe Attribute, Abstraktion von Implementierungsdetails
    final boolean isBold = isBold(fontStyles);                          // V2
    final boolean isItalic = isAttributeEnabled(fontStyles, ITALIC);   // V3

    System.out.println("isUnderline? " + isUnderline); // false
    System.out.println("isBold?      " + isBold);      // true
    System.out.println("isItalic?    " + isItalic);    // true
}

// Variante 2
private static boolean isBold(final int fontStyles)
{
    return (fontStyles & BOLD) == BOLD;
}

// Variante 3
private static boolean isAttributeEnabled(final int fontStyles,
                                          final int attributeValue)
{
    return (fontStyles & attributeValue) == attributeValue;
}
```

Listing 3.2 *Ausführbar als* **'FONTATTRIBUTESEXAMPLE'**

Das Programm FONTATTRIBUTESEXAMPLE gibt Folgende aus:

```
isUnderline? false
isBold?      true
isItalic?    true
```

Die im oberen Listing gezeigten Prüfungen werden schnell unübersichtlich, wenn man sie, wie für das Attribut `isUnderline` gezeigt, selbst programmiert (Variante 1). Dies macht sich besonders negativ bemerkbar, wenn viele Abfragen unterschiedlicher Eigenschaften erfolgen sollen. Die Lesbarkeit des Sourcecodes leidet und die Gefahr für Fehler steigt. Es bietet sich an, die Prüfung der Eigenschaften in Methoden auszulagern, wie dies im Beispiel für die Methode `isBold()` (Variante 2) umgesetzt ist. Dadurch lässt sich nutzender Sourcecode besser lesen. Sind allerdings viele Eigenschaften zu prüfen, so steigt auch die Anzahl trivialer Prüfmethoden. Das Programm nimmt an Umfang zu, ohne viel mehr Funktionalität bereitzustellen. Es bietet sich dann eine allgemeinere Prüfmethode ähnlich zu der gezeigten `isAttributeEnabled()` an (Variante 3).

Elegantere Realisierung mithilfe der Klasse `EnumSet` Einfacher und eleganter lässt sich die Aufgabenstellung realisieren, wenn man die Darstellungsattribute als `enum`-Aufzählung `FontAttributesEnum`[29] realisiert:

```
enum FontAttributesEnum
{
    BOLD, ITALIC, UNDERLINE;
}
```

Durch Einsatz der generischen Klasse `EnumSet<E>` kann man auf effiziente Weise Mengen von `enum`-Aufzählungswerten vom Typ `E` verwalten. Zudem können wir auf die Definition des zuvor benötigten Werts `NORMAL` verzichten, da sich dieser als leere Menge darstellen lässt. Wir schreiben das Beispiel wie folgt um:

```
public static void main(final String args[])
{
    final EnumSet<FontAttributesEnum> fontStyles = EnumSet.of(BOLD, ITALIC);

    System.out.println("isUnderline? " + fontStyles.contains(UNDERLINE));
    System.out.println("isBold?      " + fontStyles.contains(BOLD));
    System.out.println("All  " + EnumSet.allOf(FontAttributesEnum.class));
    System.out.println("None " + EnumSet.noneOf(FontAttributesEnum.class));
}
```

Listing 3.3 *Ausführbar als* **'FONTATTRIBUTESENUMEXAMPLE'**

Das Programm FONTATTRIBUTESENUMEXAMPLE produziert folgende Ausgaben:

```
isUnderline? false
isBold?      true
All  [BOLD, ITALIC, UNDERLINE]
None []
```

[29] Das Namenssuffix `Enum` wurde hier nur zur Unterscheidung von der auf `int`-Konstanten basierenden zuvor vorgestellten Aufzählung gewählt und sollte normalerweise entfallen.

Im Programm wurden verschiedene der im Folgenden beschriebenen FABRIKMETHODEN (vgl. Abschnitt 18.1.2) der Klasse `EnumSet<E>` eingesetzt. Diese ermöglichen es, Mengen von `enum`-Aufzählungswerten zusammenzustellen:

- `EnumSet<E> of(E first, E... others)` – Erstellt eine Menge, die aus den angegebenen Werten besteht.
- `EnumSet<E> range(E from, E to)` – Beschreibt eine Menge, die alle Werte enthält, die zwischen den beiden angegebenen Werten liegen, auch die angegebenen Grenzwerte.
- `EnumSet<E> allOf(Class<E> elementType)` – Erstellt eine Menge, die alle Werte des angegebenen Typs enthält.
- `EnumSet<E> noneOf(Class<E> elementType)` – Konstruiert eine typsichere leere Menge vom Typ `EnumSet<E>`.

Hinweis: Sprachelemente Varargs und Klasseninformationen

Trotz der Kürze des Beispiels sind verschiedene Dinge erwähnenswert: Neben der neu kennengelernten Klasse `EnumSet<E>` wurden zwei bisher in diesem Buch nicht beschriebene Sprachelemente genutzt, die ich nachfolgend kurz erläutere.

Zum einen ist dies die sogenannte variable Argumentliste (kurz ***Varargs***), die beliebig viele Übergabeparameter eines Typs erlaubt. Syntaktisch wird dies durch drei Punkte nach der Typangabe, etwa `String...` in einer Methodensignatur ausgedrückt. Das bedeutet hier also 0 bis n Elemente vom Typ `String`. Diese Notation wird im Beispiel etwa beim Aufruf der Methode `EnumSet.of(E, E...)` eingesetzt und erfordert die Angabe eines Elements des Enum-Typs `E`, hier also vom Typ `FontAttributesEnum`, gefolgt von beliebig vielen weiteren Elementen des gleichen Typs. Für dieses praktische Feature gibt es jedoch eine Einschränkung: Es kann nur einen Vararg-Parameter geben und dieser muss als Letzter angegeben werden.

Zum anderen haben wir die Angabe von Informationen zum Typ beim Aufruf von `EnumSet.allOf(Class<E>)` genutzt, mit deren Hilfe wir alle Elemente vom Typ `FontAttributesEnum.class` als `EnumSet<FontAttributesEnum>` erhalten. Wie im Beispiel gesehen, kann man auf diese Typinformationen über die Notation `<Typ>.class`, etwa `String.class` oder `String[].class`, zugreifen und erhält dann ein Objekt vom Typ `Class<E>`. Ganz allgemein wird jeder Typ in Java durch ein `Class<E>`-Objekt beschrieben und bietet darüber Zugriff auf die Metainformationen von Klassen: Man kann etwa Methoden und Attribute sowie deren Sichtbarkeiten abfragen. Außerdem lässt sich der Typ einer Instanz mithilfe der Methode `getClass()` ermitteln und diese Rückgabe mit dem Typ anderer Objekte vergleichen.

3.4.5 Value Object (Data Transfer Object)

Das Implementierungsmuster VALUE OBJECT[30] (oder auch DATA TRANSFER OBJECT) folgt der Idee, einen Container für mehrere Attribute oder Parameter bereitzustellen, um diese zu neuen Strukturen zusammenzufassen. Dies ist beispielsweise immer dann sinnvoll, wenn man Parameter in einer Methodensignatur oder Attribute in einer Klasse semantisch gruppieren möchte. Zudem wird die ursprünglich benötigte Anzahl an Parametern bzw. Attributen reduziert. Je nach Einsatzort und -zweck besitzt dieses Muster diverse Namen: Fasst es verschiedene Parameter in einer Methodensignatur zusammen, so spricht man häufig auch von einem ***Parameter Value Object*** oder kurz ***Parameter Object***. Im Bereich von Client-Server-Architekturen kennt man diese Art der Realisierung unter dem Namen ***Data Transfer Object*** (DTO). Hier sorgt es dafür, dass die Übertragung von Daten nicht feingranular, Attribut für Attribut, sondern in einem Rutsch erfolgen kann. Das bietet den Vorteil, dass nur einmalig statt mehrfach die Kosten für Netzwerkzugriffe zu zahlen sind, und macht sich positiv bemerkbar: Insbesondere verbessert es die Performance und erleichtert Transaktionalität.

Als Beispiel betrachten wir hier eine Klasse `Person`, die zwei Value Objects referenziert, die die Adressdaten und die Kontaktinformationen (Telefon, E-Mail usw.) beinhalten. Abbildung 3-32 zeigt dies.

Abbildung 3-32 *Beispiele für Realisierungen von Value Objects*

Java 14 Records als Alternative Mit Java 14 wurden Records eingeführt. Diese erlauben die Definition von unveränderlichen Datencontainerklassen. Details finden Sie in Abschnitt 14.1.3.

[30] Die Herkunft der Begrifflichkeit wird unter `http://www.adam-bien.com/roller/abien/entry/value_object_vs_data_transfer` dargelegt.

Anmerkungen

Häufig ist es wünschenswert und sinnvoll, semantisch zusammengehörende Werte zu gruppieren. In einigen Fällen kann die semantische Klammer auch nur darin bestehen, die Übergabe oder Datenspeicherung zu kapseln. Dabei wird bewusst kein Verhalten in Form von Business-Methoden angeboten, sondern lediglich ein »dummer« Container für Daten, der Zugriff auf die gespeicherten Werte durch feingranulare `get()`- und `set()`-Methoden erlaubt. Die in diesem Kapitel zuvor angeführte Kritik an `get()`- und `set()`-Methoden greift hier nicht, da kein Verhalten modelliert wird. Mehr noch, vielfach kann man sogar auf Zugriffsmethoden vollständig verzichten, wenn die Value Objects lokal im eigenen Programmcode genutzt werden.

Bewertung

Der Einsatz eines Value Object / Data Transfer Object ...

+ erlaubt es, einen eigenen Datencontainer zu realisieren, dessen Verwendung Details versteckt, da weniger Attribute und Parameter zum Einsatz kommen.
+ fasst Daten zusammen und kann für klarere Methodensignaturen sorgen.
o modelliert nur einen Ausschnitt aus einem größeren Objekt und repräsentiert zudem die Wertebelegung zu einem speziellen Zeitpunkt. Ein Konsistenzabgleich zwischen unterschiedlichen Ständen kann schwierig sein.

Tipp: Vermeide Hilfsmethoden in DTOs

Manchmal ist man versucht, Hilfsmethoden in einem DTO zu definieren. Dieser Versuchung sollte man jedoch widerstehen. Zum einen widerspricht dies dem Gedanken, einfache Datencontainer zur Verfügung zu stellen. Zum anderen kann dies zu Problemen führen, wenn ein Transport per Netzwerk benötigt wird.

Die Übertragung von DTOs über ein Netzwerk wird vereinfacht, wenn diese das Interface `Serializable` implementieren. Dadurch können sie von der JVM automatisch in eine Folge von Bytes umgewandelt werden. Dabei wird eine spezielle Versionsnummer benutzt, um die Typen der Objekte zu identifizieren. Diese Versionsnummer kann explizit als Attribut `serialVersionID` im Sourcecode angegeben werden. Häufig ist dies aber nicht der Fall. Dann wird automatisch beim Kompilieren eine solche Versionsnummer berechnet und beim Serialisieren verwendet. In die Berechnung gehen unter anderem die in einer Klasse definierten Methoden ein. Jede neu eingeführte Methode verändert damit diese automatisch ermittelte Versionsnummer und führt zu Inkompatibilitäten zu älteren Versionen des DTOs, ohne dass sich die strukturelle Zusammensetzung (die Attribute) des Containers geändert hätte. Details zur Serialisierung beschreibt Abschnitt 12.3.

Es ist demnach nicht nur aus der Sicht der Trennung von Daten und Verarbeitung, sondern auch aus Gründen der Kompatibilität sinnvoll, für ein DTO benötigte Hilfsmethoden in separaten Toolkit- oder Utility-Klassen zu definieren.

3.5 Prinzipien guten OO-Designs

Mittlerweile sollten Sie einen ersten Eindruck davon gewonnen haben, was gute objektorientierte Entwürfe ausmacht und welche Probleme einem mitunter dabei begegnen können. Bevor wir uns gleich intensiver mit gutem Design beschäftigen, möchte ich einführend auf Kennzeichen für schlechtes Design eingehen.

Was ist schlechtes Design? Woran erkennt man es?

Sicherlich ist Ihnen beim Überarbeiten von Programmen schon das eine oder andere Problem begegnet. Ein Zeichen für schlechtes Design ist häufig, wenn ...

- der Sourcecode wenig verständlich ist, unter anderem verursacht durch ***ungünstige Namensgebung*** oder eine ***fehlende Dokumentation*** komplizierterer Stellen.
- ***unnötige Komplexität*** oder ein extrem flexibles Design mit hohem Variantenreichtum existiert, obwohl diese Extras (teilweise) nicht benötigt werden.
- sich Erweiterungen oder ***Modifikationen nur schwierig durchführen lassen*** und zum Teil ***große Auswirkungen*** auch in anderen Teilen des Sourcecodes besitzen.
- das zu lösende Problem anhand der Implementierung nicht abgeleitet werden kann, etwa weil es an semantischer Strukturierung fehlt oder ***kein klares Layout des Sourcecodes*** vorliegt.
- keine oder nur ***wenige Tests*** existieren und Testbarkeit kaum gegeben ist, z. B. weil sich ***Klassen nur schwierig eigenständig testen lassen, weil Objekte stark voneinander abhängen***, sodass lediglich schwierig überprüfbare Objekthaufen vorhanden sind.
- ***diverse Fehler*** bereits bekannt sind und wahrscheinlich noch viele weitere versteckt lauern. Das erkennt man an Folgendem: Behebt man einen Fehler, so stößt man recht schnell auf einen weiteren. Behebt man auch diesen, so kommt es zu weiteren Problemen an mitunter sogar ganz anderer Stelle. Das System ist so weit verrottet, dass nur noch ein Aufräumkraftakt oder aber ein vollständiges Redesign helfen.

Gutes Design

In den vorangegangenen Abschnitten wurde das Thema Design vor allem praxisbezogen betrachtet. Nun wollen wir das Ganze etwas formalisieren und uns dazu fragen, welche Regeln oder Vorgehensweisen man befolgen sollte, um die zuvor geschilderten Negativpunkte schlechter Designs möglichst zu vermeiden. Wie entstehen also wartbare und gut strukturierte Programme und was zeichnet gelungenes objektorientiertes Design aus? Und wie erreicht man Eigenschaften wie Erweiterbarkeit, Wartbarkeit und Verständlichkeit.

Darüber haben sich schon diverse erfahrene Leute Gedanken gemacht und ihre Erkenntnisse zu verschiedenen Regeln, Leitsätzen und Prinzipien vereinigt. Man sollte aus diesem Erfahrungsschatz lernen und einiges davon kennen und einhalten.

3.5.1 Geheimnisprinzip nach Parnas

Gemäß dem ***Geheimnisprinzip*** (***Information Hiding***) sollten Klienten einer Komponente zu deren Nutzung keine Kenntnis über die internen Details besitzen (müssen). Schon 1972 wurde das Prinzip durch Parnas formuliert. Auf die heutige Zeit und Java übertragen, bedeutet dies, dass jede Komponente (Klasse, Package) eines Programms ihre Implementierungsdetails vor anderen Komponenten verbergen sollte. Daraus ergibt sich die Forderung, nur diejenigen Bestandteile einer Komponente nach außen zugänglich zu machen, die für andere Komponenten zur Zusammenarbeit wirklich relevant sind. Für Klassen erreicht man dies, indem das Objektverhalten lediglich über öffentliche Business-Methoden bereitgestellt wird. Somit kann der Objektzustand von außen nur über diese Business-Methoden modifiziert werden. Für Attribute und Methoden ermöglicht eine stark eingeschränkte Sichtbarkeit, Implementierungsdetails möglichst geheim zu halten. Dies wird durch die mit Java 9 eingeführte Modularisierung deutlich vereinfacht (vgl. Kapitel 15).

Befolgt man das Geheimnisprinzip, so kann man oftmals die einzelnen Bestandteile einer Software als Blackbox mit definierter Schnittstelle ansehen.

Bewertung

Wenn man das Geheimnisprinzip anwendet, so ...

+ entsteht ein eher lose gekoppeltes System mit einer guten Modularisierung.
+ sollten sich einzelne Teile leichter unabhängig voneinander testen lassen. Auch die Anzahl der benötigten Tests reduziert sich, da Aufrufe nur über die öffentliche Schnittstelle erfolgen und Zustandsänderungen besser kontrolliert werden können.
+ erleichtert dies das Verständnis und die Nachvollziehbarkeit. Allein basierend auf der öffentlichen Schnittstelle sollte die Funktionalität ermittelbar sein.
+ können Details der Implementierung ohne Rückwirkungen auf Nutzer geändert werden. Beispielsweise können Daten entweder als Attribut gehalten, bei Bedarf berechnet oder aus einer externen Quelle gelesen werden.[31]
- ist ein gewisser Mehraufwand zur Implementierung von Zugriffsmethoden zur Datenkapselung und für deren Aufruf notwendig.

[31] Eine Austauschbarkeit der Implementierung ohne Nebenwirkungen gilt für die Praxis nur dann, wenn keine zeitlichen Randbedingungen vom Aufrufer angenommen werden.

3.5.2 Law of Demeter

Beim ***Gesetz von Demeter*** (Law of Demeter, kurz ***LoD***) geht es darum, die Kopplung auf ein verständliches und wartbares Maß zu reduzieren. Da das noch recht abstrakt ist, betrachten wir zunächst einige Negativbeispiele, die Verstöße gegen das LoD zeigen.

Negativbeispiele

Jeder kennt sicher Gebilde, die mehrere Methodenaufrufe wie folgt mithilfe der .-Notation miteinander verknüpfen:

```
getPreferencesService().getDimension(MAIN_WINDOW_ID).setWidth(700);
getPreferencesService().getColorScheme(OCEAN).getTextColor().setColor(BLUE);

if (getCommandProcessor().getPool().getSize() >= MAX_POOL_SIZE)
{
    // warning
}
else
{
    // process command
}
```

Dieses Beispiel zeigt, dass das eigene Objekt durch die Aufrufe recht tief in die Interna anderer Objekte eingreift bzw. diese abfragt. Dabei werden diverse Annahmen getroffen, etwa darüber, dass die Komponenten alle zugreifbar und korrekt initialisiert sind. Weil man dies nicht immer voraussetzen kann, sollten gegebenenfalls `null`-Prüfungen vor den Zugriffen erfolgen. Allerdings können diese die Lesbarkeit beeinträchtigen, weil sie den Sourcecode aufblähen und somit schwieriger nachvollziehbar und schlussendlich schlechter wartbar machen. Es gibt aber noch einen viel schwerwiegenderen Nachteil bei der Missachtung des LoD: Beim Einsatz der gezeigten Aufrufketten erfordern Änderungen an den Details genutzter Klassen nahezu zwangsläufig auch Änderungen in der eigenen Klasse. Schauen wir uns dies genauer an und abstrahieren dazu ein wenig und gehen von folgendem Aufruf aus:

```
ownObject.getObjA().getObjB().methodC();
```

Das sieht noch recht harmlos aus, aber betrachten wir die Konsequenzen möglicher Änderungen. Es gibt Probleme, wenn ...

- etwas im Design geändert wird und das Objekt `objB` nicht mehr durch Objekt `objA` bereitgestellt wird.
- die Methode `methodC()` verändert, umbenannt oder in eine andere Klasse verschoben oder gar gelöscht wird.

Durch die enge Verquickung des eigenen Objekts `ownObject` mit den anderen Implementierungen erfordert all dies auch Modifikationen an der eigenen Klasse. Das kann aufwendig werden und sowohl die Wartbarkeit als auch die lose Kopplung beeinträchtigen. Die Auswirkungen sind besonders störend, wenn die Aufrufketten über System-,

Modul- oder Package-Grenzen hinweg gehen. Möglicherweise lassen sich die referenzierten Klassen nur mühsam oder gar nicht anpassen. Zum Teil kennen die Implementierer dieser Klassen Sie als Nutzer nicht, wodurch die inkompatiblen Änderungen nicht einmal kommuniziert werden können. Schauen wir uns nun an, wie die Einhaltung des Gesetzes von Demeter uns vor derartigen Problemen schützen soll.

Regeln beim Gesetz von Demeter

Das Gesetz von Demeter ist unter dem plakativen Begriff »***Don't talk to strangers***« bekannt und definiert ***Regeln zur Gestaltung von Aufrufen***. Es wird gefordert, möglichst nur diejenigen Methoden oder Attribute anderer Klassen in Aufrufen zu nutzen, die direkt (ohne Indirektionen) bekannt und zugreifbar sind. Konkret bedeutet dies Folgendes: Nutze nur ...

1. Methoden der eigenen Klasse,
2. Methoden von Objekten, die als Parameter übergeben werden,
3. Methoden von Objekten, die das eigene Objekt selbst erzeugt, oder
4. Methoden assoziierter Klassen.

Neben diesen Regeln hilft zur Anschauung folgende Abbildung 3-33 mit drei Objekten. Das eigene Objekt `Me` referenziert ein Objekt `A` und dieses wiederum ein Objekt `B`. Gemäß dem LoD darf das eigene Objekt nur Methoden aus Objekt `A` aufrufen, nicht jedoch solche von Objekt `B`.

Abbildung 3-33 *Law of Demeter*

Beispiel für die Befolgung der Regeln

Um ein Gespür für die Regeln des LoD zu bekommen, betrachten wir eine Klasse `Person`, die ein `Address`-Objekt erzeugt und ein optionales `Company`-Objekt referenziert. Im Listing sind verschiedene Methodenaufrufe markiert, die die obigen Regeln einhalten. Beispiele für Verstöße sind in Kommentarzeilen dargestellt:

```
// Getter und Setter aus Gründen der Übersichtlichkeit nicht gezeigt
class Person
{
    final String name;
    final boolean isAdult;
    int age;
    Address homeAddress = new Address()
    Company company = null;
```

```
    Person(final String name, final int age)
    {
        this.name = name;
        this.age = age;

        // Regel 1
        this.isAdult = isOlderThan(18);
    }

    // Regel 1
    boolean isOlderThan(final int desiredAge)
    {
        return getAge() > desiredAge;
    }

    boolean sameAge(final Person other)
    {
        // Regel 2
        return getAge() == other.getAge();
    }

    boolean livesIn(final City city)
    {
        // Regel 3
        return getAddress().getCity().equals(city);
        // Verstoß: getAddress().getCity().getZipCode() == city.getZipCode()
    }

    boolean isManager()
    {
        // Regel 4
        return (company != null && company.isInManagingPosition(name));
        // Verstoß: company.getStuffMembers().isManager(name)
    }
}
```

Bewertung

Wenn man das Gesetz von Demeter einhält, so ...

+ entsteht ein eher lose gekoppeltes System mit weniger verzweigten Abhängigkeiten der Klassen untereinander, was die Wiederverwendbarkeit erleichtert.
+ lassen sich einzelne Klassen leichter modifizieren und diese Änderungen pflanzen sich mit geringerer Wahrscheinlichkeit im System fort.
+ kann man einzelne Klassen einfacher und unabhängig von anderen testen.
- müssen zum Teil weitere Methoden in das Interface der eigenen Klasse aufgenommen werden, weil einige Methoden eventuell nun nicht mehr über Indirektionen angesprochen werden dürfen.

3.5.3 SOLID-Prinzipien

SOLID ist ein Akronym und beschreibt fünf von Robert C. Martin vorgestellte Prinzipien guten OO-Entwurfs, die sich insbesondere mit Design beschäftigen:

- **S** – SINGLE RESPONSIBILITY PRINCIPLE – Jede Klasse sollte möglichst nur genau für eine Aufgabe zuständig sein.
- **O** – OPEN CLOSED PRINCIPLE – Dieses Prinzip besagt, dass Klassen offen für Erweiterungen sein sollen, aber geschlossen gegenüber Änderungen.
- **L** – LISKOV SUBSTITUTION PRINCIPLE – Hierbei geht es um Ersetzbarkeit von Basisklassen durch Subklassen.
- **I** – INTERFACE SEGREGATION PRINCIPLE – Entwerfe Interfaces, sodass sie gut für Klienten passen. Bevorzuge mehrere feingranulare Interfaces gegenüber einem oder wenigen umfangreichen Interfaces.
- **D** – DEPENDENCY INVERSION PRINCIPLE – Vermeide direkte Abhängigkeiten auf konkrete Klassen.[32]

Single Responsibility Principle

Das ***Single Responsibility Principle*** (SRP) besagt, dass eine Klasse (möglichst) ***genau eine klar definierte Aufgabe erfüllen soll*** und es folglich nur einen oder ***wenige Gründe für Änderungen*** geben sollte.[33] Umgangssprachlich könnte man sagen, ***wer viele Dinge auf einmal tut, dem gelingen selten alle gut***: Die Wahrscheinlichkeit ist recht hoch, dass keine der Aufgaben richtig erfüllt wird. Zudem besitzen komplexe Klassen oftmals zu viele und zum Teil unerwünschte Abhängigkeiten auf andere Klassen.

Wird das SRP eingehalten, so erzielt man (in der Regel) eine hohe Kohäsion, also einen hohen Zusammenhalt einer Klasse. Die Ausrichtung einer Klasse auf genau eine Funktionalität führt auch zu Orthogonalität. Damit meint man, dass Funktionalitäten ohne (größere) Nebenwirkungen miteinander kombiniert werden können. Das erleichtert es, größere Systeme wartbar aus kleineren Bestandteilen zusammenzusetzen.

Woran kann man aber erkennen, dass das SRP verletzt wird? Ein Indiz dafür ist, dass es schwerfällt, prägnante Namen für eine Methode oder eine Klasse zu finden oder dass dieser Name mehrere Verben oder Nomen enthält. Methoden besitzen dann etwa Namen wie `collectAndFilter()`, `selectAndUpdate()` oder `sortAndPrint()`. Außerdem ist die schiere Methodenlänge ein recht guter Indikator: Je länger die Methode, desto größer ist die Wahrscheinlichkeit für einen Verstoß gegen das SRP.

[32] Teilweise ist es hilfreich, wenn Objekte über klar definierte Schnittstellen (in Form eines Interface oder einer abstrakten Klasse) miteinander kommunizieren. Zudem sollten die Abhängigkeiten (Dependencies) in der Regel nicht von Klassen selbst aufgebaut, sondern besser von außen und möglichst spät »injected« (eingeimpft) werden.

[33] Diese Aussage lässt sich für Methoden, Klassen und Komponenten nutzen, wobei der Kontext der Aufgabe natürlich größer wird, wenn man von Methoden bis hin zu Komponenten geht.

Open Closed Principle

Das ***Open Closed Principle*** (OCP) zielt auf die ***leichte Erweiterbarkeit*** und ***korrekte Kapselung*** sowie ***Trennung von Zuständigkeiten***. Wenn man es extrem sehen möchte, sollte sich eine Klasse nach ihrer Fertigstellung nur noch dann ändern müssen, wenn komplett neue Anforderungen oder Funktionalitäten zu integrieren sind oder aber Fehler korrigiert werden müssen. Dahingegen sollten Änderungen an der eigenen Klasse nicht dadurch erforderlich sein, dass sich andere Klassen ändern. Hier besteht auch ein Zusammenhang zu dem zuvor beschriebenen Gesetz von Demeter.

Schauen wir uns ein Beispiel an, um das OCP ein wenig besser zu verstehen. Dazu betrachten wir eine Spieleapplikation, die verschiedene Bonuselemente, wie Extraleben oder Zusatzausrüstungen, als Anreiz anbietet. Ihre Aufgabe als Entwickler ist es nun, ein neues Level zu gestalten und dort neue Arten von Bonuselementen zu integrieren. Wünschenswert wäre es, wenn dies möglichst einfach realisierbar wäre, etwa wenn man lediglich neue Klassen für die neuen, speziellen Bonuselemente erstellen müsste. Das wäre beispielsweise dann möglich, wenn die Bonuselemente alle ein gemeinsames Interface `IBonusElement` erfüllen oder eine (abstrakte) Basisklasse implementieren würden. In diesem Fall sollte die restliche Applikation kaum oder im besten Fall gar nicht von Änderungen bzw. Erweiterungen der Bonuselemente betroffen sein. Eine gemeinsame Basis aus Interface und/oder abstrakter Basisklasse und der Möglichkeit zur Definition von Spezialisierungen sind eine Variante, das OCP umzusetzen. Ein Verstoß gegen OCP besteht dann, wenn die beschriebene Erweiterung an diversen Stellen zu Änderungen führen würde.

Auch das Entwurfsmuster SCHABLONENMETHODE (vgl. Abschnitt 18.3.3) kann dabei helfen, das OCP zu realisieren. Dort gibt man gewisse Rahmenbedingungen vor, erlaubt aber Modifikationen an verschiedenen Sollbruchstellen.

Info: OPEN CLOSED PRINCIPLE im realen Leben

Ein Paradebeispiel aus der realen Welt für das OCP sind HiFi-Verstärker. Diese können jede beliebige Quelle verstärken, solange diese über Cinch-Buchsen angeschlossen werden kann. Auf diese Weise kann man alte Kassettenrekorder, MiniDisc-Player, CD-Player, aber auch moderne Blu-Ray-Player anschließen und deren Ton wiedergeben.

Im Zuge der Digitalisierung musste dann eine weitere Schnittstellenklasse in Form optischer bzw. koaxialer Anschlüsse geschaffen werden. Deutlich später stellte die Erweiterung auf Heimkino-Surround-Ton eine derart massive Änderung dar, dass wiederum eine neue Schnittstelle bzw. eine Erweiterung notwendig war und HDMI als neue Schnittstelle Einzug hielt. In allen Fällen wurde aber das OCP befolgt: Eine neue Schnittstelle entsprach einer neuen Anforderung und führte einmalig zu einer Änderung. Danach konnten damit alle kompatiblen Geräte angeschlossen werden. So ist es auch über einen 25 Jahre alten Verstärker möglich, den Ton von DVD oder Blu Ray wiederzugeben, obwohl diese Technologien zum Herstellungszeitpunkt des Verstärkers noch lange nicht existierten.

Liskov Substitution Principle

Das ***Liskov Substitution Principle*** (LSP) behandelt die ***Ersetzbarkeit*** und die ***Einhaltung der »is-a«-Beziehung***. Damit ist gemeint, dass eine ***Instanz einer Subklasse überall dort problemlos genutzt werden können sollte, wo eine Instanz der Basisklasse zum Einsatz kommt***. Es geht darum, dass ein Nutzer ein erwartungskonformes Verhalten erhält, also dass ein von Subklassen in Methoden realisiertes Verhalten mit demjenigen der Basisklassen kompatibel sein muss.

Weil das kompliziert klingt (und es teilweise auch ist), betrachten wir ein Beispiel und nutzen dazu erneut grafische Figuren. Die Basisklasse `BaseFigure` bildet die Grundlage für Spezialisierungen wie Kreise, Rechtecke usw. Das Zeichnen sollte ohne Beachtung des konkreten Figurentyps (also dem OCP entsprechend) in etwa wie folgt möglich sein:

```
figuresToDraw.add(new Circle(60, 70, 80));
figuresToDraw.add(new Rectangle(50, 50, 100, 200));

// ...

for (final BaseFigure figure : figuresToDraw)
{
    figure.draw(graphicsContext);
}
```

Das gewünschte und erwartete Verhalten scheint recht natürlich. Ob dieses auch erreicht wird, hängt jedoch stark von den konkreten Subklassen ab – hier etwa, ob die Methode `draw()` wirklich das tut, was man anhand des Namens erwartet. Ein Verstoß bestünde etwa darin, einfach nichts zu zeichnen oder eine Datenbankabfrage auszuführen, also Dinge, die gegen die natürliche Erwartungshaltung bei Aufruf von `draw()` (massiv) verstoßen.

LSP für Eingabeparameter, Rückgabewerte und Exceptions Das LSP gilt auch für die Typen von Rückgabewerten und Exceptions. Dazu nehmen wir folgende Hierarchie von Exceptions, Rückgabewerten und Klassen an:

```
class CalculationException extends Exception
{
    // ...
}
class SpecialCalculationException extends CalculationException
{
    // ...
}

class BaseFigure
{
    Number calcArea() throws CalculationException
    {
        return new Double(getWidth() * getHeight());
    }
}
```

```
class Polygon extends BaseFigure
{
    // Speziellere Rückgabe (Double extends Number) und speziellere Exception
    Double calcArea() throws SpecialCalculationException
    {
        // Bewusst, um gleich ein Problem zu zeigen
        return null;
    }
}
```

Wir erkennen Folgendes: Beim ***Überschreiben von Methoden*** dürfen Subklassen als Rückgabe einen ***spezielleren Typ zurückliefern*** als die Basisklasse. Es dürfen ebenfalls ***spezifischere Exceptions ausgelöst*** werden (Details dazu finden Sie in Abschnitt 3.6 bei der Besprechung verschiedener Formen der Varianz). Weil dies unspektakulär klingt, betrachten wir nun einen möglichen Nutzer in Form der folgenden Klasse `Client`:

```
class Client
{
    static void doSomethingWithFigure(final BaseFigure figure) throws
          CalculationException
    {
        final Number result = figure.calcArea();
        // NullPointerException für Polygon
        final double area = result.doubleValue();
        // ...
    }
}
```

Ebenso wie für das Zeichnen durch Aufruf der Methode `draw()` sollte auch für die Berechnung der Fläche durch die Methode `calcArea()` keine Unterscheidung der Figurentypen notwendig sein. Zur Demonstration der Verletzung des LSP gibt die Klasse `Polygon` jedoch statt einer Zahl den Wert `null` zurück, was beim Aufrufer zu einer `NullPointerException` führt. Ähnliche Probleme diskutieren wir später als BAD SMELL: UNBEDACHTE RÜCKGABE VON `NULL` in Abschnitt 16.3.5.[34]

Besonderheiten in Klassenhierarchien Im vorangegangenen Beispiel und bei Beachtung der Voraussetzungen einer »is-a«-Beziehung scheint das LSP nicht ganz so schwierig einzuhalten zu sein. Es gibt jedoch einige Beispiele aus der OO-Modellierung, wo gemäß der normalen Logik eine »is-a«-Beziehung gilt, bei deren Realisierung als Klassenhierarchie es aber Probleme gibt – insbesondere auch mit dem LSP. In Abschnitt 3.3.1 wurden am Beispiel einer Simulationssoftware für Lebensräume und verschiedene Tierarten einige Problemfälle der Vererbung aufgezeigt, etwa eine Basisklasse `Bird` mit der Methode `fly()` sowie den Subklassen `Ostrich` und `Penguin` für Sträuße bzw. Pinguine. Beides sind ganz offensichtlich Vogelarten und somit gilt die »is-a«-Beziehung, die eine Spezialisierung ausdrückt. Jedoch können diese speziellen Vogelarten nicht fliegen und somit lässt sich auch die Methode `fly()` nicht

[34] Als Bad Smell bezeichnet man eine (potenziell) problematische Programmstelle.

sinnvoll implementieren. Es verbleibt als möglicher Ausweg, entweder die Methode leer zu implementieren oder aber eine Exception auszulösen.

Aber selbst bei (scheinbar) einfachen Vererbungsbeziehungen können sich Verstöße gegen das LSP ergeben, etwa bei der Abbildung der Beziehungen von Rechteck und Quadrat sowie von Ellipse und Kreis mithilfe von Vererbung. Mathematisch betrachtet sind ein Quadrat und ein Kreis jeweils eine Spezialform eines Rechtecks bzw. einer Ellipse. Somit könnte man beim objektorientierten Programmieren auf die (ansonsten meistens sinnvolle) Idee kommen, die reale Welt in Form einer analogen Klassenhierarchie abzubilden. Versuchen wir dies und schauen, warum es für OO leider so nicht funktioniert. Nehmen wir an, die Klasse `Square` wäre eine Subklasse von `Rectangle`, für die jeweils Breite und Höhe über Zugriffsmethoden veränderlich sind:

```
public class Rectangle
{
    private int width;
    private int height;

    public void setWidth(final int width)
    {
        this.width = width;
    }

    public void setHeight(final int height)
    {
        this.height = height;
    }

    public int calcArea()
    {
        return getWidth() * getHeight();
    }

    public int getHeight()  { return this.height; }
    public int getWidth()   { return this.width;  }
}
```

Die Klasse `Square` erbt die Methoden zum Setzen der Breite und Höhe – allerdings besitzt ein Quadrat per Definition immer gleiche Werte für diese Seitenlängen. Damit müssen wir die `set()`-Methoden für Breite und Höhe anpassen, sodass sie immer beide Seitenlängen setzen. Dazu führen wir eine Methode `setSideLength(double)` ein, die zum Setzen der jeweiligen Werte die Methoden der Basisklasse aufruft:

```
public class Square extends Rectangle
{
    public void setSideLength(final int sideLength)
    {
        // gleiche Seitenlänge sicherstellen
        super.setWidth(sideLength);
        super.setHeight(sideLength);
    }

    public void setWidth(final int width)
    {
        setSideLength(width);
    }
```

```
    public void setHeight(final int height)
    {
        setSideLength(height);
    }
}
```

Eine nutzende Klasse, etwa ein Unit Test, könnte nun prüfen, ob für Rechtecke nach dem Setzen unterschiedlicher Werte für Breite und Höhe z. B. den Werten 5 und 10 auch 50 als Fläche berechnet wird:

```
@Test
public void testAreaCalculation()
{
   rectangle.setWidth(5);
   rectangle.setHeight(10);

   assertEquals(50, rectangle.calcArea());
}
```

Sofern die Variable vom Typ `Rectangle` ist, wird der Test bestanden. Würde man den Test allerdings mit einem `Square`-Objekt als Subtyp von `Rectangle` ausführen, bekäme man als Fläche entweder 25 oder 100, je nachdem, in welcher Reihenfolge man Breite oder Höhe als Zweites setzt. Damit widerspricht das Ergebnis der Flächenberechnung für eine `Square`-Instanz aber der Erwartungshaltung an die Berechnung für eine Instanz vom Typ `Rectangle`. Somit verletzt die gezeigte Implementierung mit der Ableitung `Square extends Rectangle` das LSP.

Lösungsmöglichkeiten Das klingt nicht gut und vor allem scheint die Lösung recht verzwickt. Tatsächlich kann man keine sinnvolle Lösung finden, sofern man eine Veränderlichkeit der Seitenlängen erlaubt. Wenn wir die Klassen jedoch in unveränderliche Klassen umwandeln, was in diesem Fall lediglich das Entfernen der `set()`-Methoden und die Definition der Attribute als `final` erfordert, dann existiert auf einmal kein Problem mehr, weil bereits zum Konstruktionszeitpunkt für Quadrate gleiche Seitenlängen sichergestellt und diese nachträglich nicht geändert werden können:

```
Square(final int sideLength)
{
    super(sideLength, sideLength);
}

Rectangle(final int width, final int height)
{
    this.width = width;
    this.height = height;
}
```

Für diverse Anwendungsfälle ist der Verzicht auf Veränderlichkeit eine gute Lösung. Allerdings gibt es dabei noch etwas zu beachten.

Weitere Fallstricke Nehmen wir an, wir würden mit den beiden Klassen folgenden Sourcecode schreiben:

```
final Square square = new Square(10);
final Rectangle rectangle = new Rectangle(10, 10);
```

Hier wird ein Quadrat vom Typ `Square` mit der Seitenlänge 10 und ein Rechteck vom Typ `Rectangle` mit der Breite 10 und der Höhe 10 erzeugt. Von seinen Abmessungen entspricht es also einem Quadrat. Nun hat man semantisch zwei Quadrate der Seitenlänge 10, aber eins davon besitzt den Typ `Square` und das andere den Typ `Rectangle`. Puh! Was nun? Zunächst einmal erkennt man, dass nicht alle vermeintlich trivialen Klassenhierarchien auch wirklich so einfach sind, wie sie scheinen, und dass deutlich mehr Fallstricke auf dem Weg zu einem gelungenen OO-Design lauern, als man sich gemeinhin so vorstellt.

Lösungsmöglichkeiten ohne Vererbung Eine mögliche Lösung besteht darin, die Klasse `Square` zu entfernen und nur noch mit dem Typ `Rectangle` zu arbeiten. Um zu ermitteln, ob ein Rechteck quadratisch ist, fügen wir eine Methode `isSquare()` ein. Sofern die Klasse `Rectangle` als unveränderliche Klasse realisiert ist und ihre Breite sowie Höhe zum Konstruktionszeitpunkt erhält, kann man die Quadrateigenschaft auch schon dann bestimmen. Aber selbst wenn die Klasse `Rectangle` veränderlich gestaltet ist, lässt sich die Eigenschaft problemlos dynamisch über eine Abfrage `getWidth() == getHeight()` ermitteln – jedoch ist das nur für Ganzzahlen zuverlässig, für Gleitkommazahlen gibt es höchstwahrscheinlich falsche Aussagen aufgrund von Rundungsproblemen.

Bei der gezeigten Umsetzung liegt keine Verletzung von LSP vor, weil erstens nicht mehr mit Vererbung gearbeitet und zweitens eine konsistente Realisierung der Quadrateigenschaft umgesetzt wird.

Interface Segregation Principle

Die Kernaussage des ***Interface Segregation Principle*** (ISP) ist, dass Benutzer einer Klasse eine ***möglichst spezifische, auf die jeweilige Aufgabe oder auf ihn als Klienten zugeschnittene Schnittstelle bereitgestellt bekommen sollten***. Das impliziert, dass keine zu breite Schnittstelle (d. h. mit zu vielen oder auch semantisch nicht zusammengehörenden Methoden) angeboten wird, sondern möglichst eine solche, die genau den Anforderungen eines möglichen Benutzers entspricht.

Oftmals sieht man in der Praxis eher zu breite oder zu unspezifische Interfaces,[35] die folglich fast immer auch Funktionalität anbieten, die ein Klient nicht benötigt, etwa wie folgt:

[35] Der Einsatz des Entwurfsmusters FASSADE (vgl. Abschnitt 18.2.1) führt auch recht leicht zu derartigen Interfaces. Hier ist das aber in Ordnung, weil eine Fassade eine Schnittstellensammlung und einen externen Zugriffspunkt als Abstraktion vieler interner Schnittstellen darstellt.

```
public interface IUniversalFileCustomerAndPizzaService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;

    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);

    void orderPizza(final long customerId, final Pizza pizza);
}
```

Als Abhilfe kann man die Schnittstelle einer Klasse über mehrere Interfaces beschreiben und so der Überfrachtung und einer schwierigeren Benutzbarkeit entgegenwirken. Für das vorherige Beispiel ließe sich das Interface etwa in folgende drei semantisch zusammengehörende Einzelinterfaces aufsplitten:

```
public interface IFileService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;
}

public interface ICustomerService
{
    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);
}

public interface IPizzaService
{
    void orderPizza(final long customerId, final Pizza pizza);
}
```

Bei einer Aufteilung sollte das Ziel jedoch nicht alleine darin bestehen, lediglich eine sehr feine Granularität der entstehenden Schnittstellen zu erreichen. Denn ansonsten bestünde eine Schnittstelle im Extremfall nur noch aus einer Methode. Das ist sogar möglicherweise in Ordnung, sofern diese eine Methode tatsächlich auch eine semantische Einheit für sich bildet. Im Beispiel ist das für die Methode `orderPizza()` und das Interface `IPizzaService` gegeben. Gleiches gilt etwa für diverse Interfaces aus dem JDK, die eine spezifische Funktionalität anbieten, etwa `Runnable` bei Multithreading oder `ActionListener` zur Ereignisbehandlung in GUIs. Normalerweise sollte man bei Interfaces mit nur einer Methode jedoch genauer hinschauen, ob nicht eine Kombination thematisch zusammengehörender Interfaces für mehr Zusammenhalt und Klarheit sorgen kann.

Schlussfolgernd lässt sich festhalten, dass der Entwurf einer gelungenen Schnittstelle gar nicht so leicht ist. Es gilt, die »richtige« Granularität zu finden. Dazu gehört etwas Erfahrung, Fingerspitzengefühl und ein wenig Ausprobieren – insbesondere auch eine Betrachtung aus Sicht möglicher Nutzer.

Dependency Inversion Principle

Das ***Dependency Inversion Principle*** (DIP) empfiehlt, dass konkrete Klassen möglichst unabhängig von anderen konkreten Klassen sein sollen, indem eine Abhängigkeit von einer konkreten Klasse durch eine Referenz auf deren Schnittstelle aufgelöst wird. Oder kurz: ***Verwende möglichst Interfaces (bzw. abstrakte Klassen), um (konkrete) Klassen voneinander zu entkoppeln***. Bei diesem Prinzip geht es also um die Reduktion der Kopplung.

Wenn Klassen lediglich über Interfaces miteinander interagieren, so kann man bei Bedarf die konkrete Implementierung auswechseln, im Idealfall sogar ohne dass dies Rückwirkungen auf den Aufrufer besitzt. Ein recht einfaches Beispiel wäre etwa, gegen das Interface `List<E>` statt gegen die konkreten Realisierungen `ArrayList<E>` oder `LinkedList<E>` zu entwickeln.

Das Programmieren gegen Interfaces kann die Wartbarkeit von Software erhöhen, weil es zu einer loseren Kopplung führt. ***Jedoch ist ein massiver Einsatz von Interfaces auch nicht sinnvoll***. Generell dürfen sich Klassen innerhalb eines Packages durchaus direkt kennen.[36] Über Package-Grenzen hinweg oder wenn es mehrere Spezialisierungen gibt, sind Interfaces häufig ein Gewinn.

Beispiel Schauen wir uns ein Beispiel an und bemühen dazu wieder den Pizza-Service. Bei der Ausführung des Bestellvorgangs werden verschiedene Daten des Kunden benötigt, dazu erfolgt ein Zugriff auf eine Datenbank. Zum Glück ist dies bereits durch ein sogenanntes Data Access Object (DAO) abstrahiert. Ebenso gibt es hin und wieder Rabattaktionen, die durch die Klasse `Discount` modelliert werden. Nachfolgend ist gezeigt, dass der `PizzaService` beide Abhängigkeiten durch Instanziieren der jeweiligen Klassen selbst auflöst (oder besser gesagt, selbst bereitstellt).

```
public class PizzaService
{
   private final Discount discount;
   private final CustomerDAO customerDAO;

   private final Map<Long, Receipt> customerToReceipt = new HashMap<>();

   public PizzaService()
   {
      // Direkte Abhängigkeiten
      discount = new Discount();
      customerDAO = new CustomerDAO();
   }

   public void orderPizza(final long customerId, final Pizza pizza)
   {
      final Customer customer = customerDAO.findById(customerId);
      customerToReceipt.putIfAbsent(customerId, new Receipt(customer));

      final Receipt receipt = customerToReceipt.get(customerId);
```

[36] Das Problem liegt teilweise darin, wie die Abhängigkeit erzeugt wird. Manchmal ist es auch akzeptabel, auf ein Interface zu verzichten, solange man die Klassen, die man benötigt, nicht selbst instanziiert.

```
        final double price = discount.apply(pizza);
        receipt.addEntry(pizza, price);
    }

    ...
}
```

Durch die Konstruktion der Objekte im `PizzaService`-Konstruktor entstehen direkte Abhängigkeiten zu den beiden Klassen. Erschwerend kommt in der Praxis oft hinzu, dass die Konstruktion von Objekten aufwendig ist, weil an Konstruktoren oft diverse Parameter übergeben und diese Informationen zuvor ermittelt werden müssen.

Analyse Die eben vorgestellte Klasse `PizzaService` besitzt verschiedene Abhängigkeiten. Wenn wir die Funktionalität einer solchen Klasse testen wollen, erschweren (zu viele) Abhängigkeiten, Unit Tests zu schreiben. Dies gilt hier aus folgenden zwei Gründen:

1. Es besteht eine Fixierung auf die Datenquelle `CustomerDAO`, und diese Klasse liest die Kundendaten aus einer Datenbank. Dadurch können wir keine Tests mit einem eng umrissenen und vordefinierten Testdatenbestand vornehmen. Außerdem sollten wir tunlichst keine Testdaten in den sensiblen Kundendatenbestand einspielen, um Missverständnisse, Fehlbestellungen oder sonstige Fallstricke zu vermeiden.
2. Die Auswirkungen verschiedener Rabattaktionen, etwa ein Einführungsangebot oder eine Happy Hour, sind schwierig zu testen, da momentan fix der durch die Klasse `Discount` realisierte Rabatt zur Preisreduktion genutzt wird.

Wie eingangs schon erwähnt, kann man die Abhängigkeiten durch ***Einführen von Interfaces*** lösen. Das ist jedoch nur für zentrale Klassen und Bestandteile sinnvoll, nämlich genau für diejenigen, die ***Sollbruchstellen oder veränderliche, austauschbare Funktionalität anbieten***. Im obigen Beispiel bildet die Rechnungserstellung mit der Klasse `Receipt` eine Ausnahme. Diese Funktionalität bleibt (vermutlich) stabil, daher fügen wir hier keine weitere Indirektion ein. Das könnte jedoch dann notwendig werden, wenn man verschiedene Zahlungs- und Rechnungsvorgänge unterstützen möchte.

Schritt 1: Interfaces einführen Wir wollen hier zunächst die Rabattberechnung und die Zugriffe auf Kunden durch Interfaces unabhängig von konkreten Realisierungen halten. Wir führen dazu die Interfaces `IDiscountStrategy` und `ICustomerRepository` sowie konkrete Realisierungen davon ein.

```
public interface IDiscountStrategy
{
    double apply(final Pizza pizza);
}

public class XL_HalfPrice_Discount implements IDiscountStrategy
{
    // ...
}
```

```
public interface ICustomerRepository
{
    Customer findById(final long customerId);
}

public class CustomerDAO implements ICustomerRepository
{
    // ...
}
```

Mit diesen Interfaces können wir in der Klasse `PizzaService` die Abhängigkeiten reduzieren, indem wir die Referenzen auf konkrete Klassen durch Interfaces ersetzen und so eine losere Kopplung erzielen.

Weitere Schritte Insgesamt sind verschiedene weitere Transformationsschritte notwendig, die hier nicht gezeigt, aber nachfolgend kurz beschrieben werden. Als Erstes ersetzt man die direkten Abhängigkeiten durch Interfaces, konstruiert die Objekte aber weiterhin innerhalb der Klasse selbst. Damit ist noch nicht allzu viel erreicht, allerdings wird dadurch der Grundstein dafür gelegt, dass man Abhängigkeiten nun als Parameter »injizieren« kann. Als zweiten Schritt konstruiert der `PizzaService` die benötigten Objekte nicht mehr selbst, sondern bekommt diese als Konstruktor- bzw. Methodenparameter übergeben. Man spricht in diesem Zusammenhang auch von ***Dependency Injection***, da Abhängigkeiten in die nutzende Klasse als Parameter übergeben werden. Abschließend kann man eine weitere Verbesserung vornehmen, indem die Rabattstrategie vom Typ `IDiscountStrategy` nicht bereits an den Konstruktor, sondern erst später an die Methode übergeben wird, die diese Referenz benötigt und damit die Berechnung durchführt. Mithilfe des beschriebenen Vorgehens lässt sich die Methode `orderPizza()` einfacher testen und Auswirkungen unterschiedlicher Strategien simulieren. Gemäß diesen Überlegungen ergibt sich folgende Implementierung der Klasse `PizzaService`, die nun dem DIP folgt. Zudem müssen wir die ursprüngliche `orderPizza()`-Methode so umgestalten, dass sie die im Listing gezeigte neue Variante nutzt:

```
public class PizzaService
{
   private final ICustomerRepository customerRepository;

   private Map<Long, Receipt> customerToReceipt = new HashMap<>();

   // Konstruktor-Injektion
   public PizzaService(final ICustomerRepository customerRepository)
   {
      this.customerRepository = customerRepository;
   }

   // Methoden-Injektion
   public void orderPizza(final long customerId, final Pizza pizza,
                          final IDiscountStrategy discountStrategy)
   {
      final Customer customer = customerRepository.findById(customerId);
      customerToReceipt.putIfAbsent(customerId, new Receipt(customer));
```

```
        final Receipt receipt = customerToReceipt.get(customerId);
        final double price = discountStrategy.apply(pizza);
        receipt.addEntry(pizza, price);
    }

    ...

}
```

Fazit

Neben den SOLID-Prinzipien haben wir auch noch das Gesetz von Demeter sowie das Geheimnisprinzip nach Parnas kennengelernt. Wenn wir uns beim Programmieren hin und wieder an diese erinnern und auch befolgen, werden wir in Zukunft bessere und leichter erweiter- und wartbare Designs produzieren.

3.6 Formen der Varianz

Zum besseren Verständnis kovarianter Ergebnistypen und von Besonderheiten beim Einsatz generischer Typen stelle ich in Abschnitt 3.6.1 verschiedene Formen der Varianz vor. Anschließend geht Abschnitt 3.6.2 auf kovariante Rückgabewerte ein.

3.6.1 Grundlagen der Varianz

Die »is-a«-Beziehung und das Substitutionsprinzip gemäß dem zuvor besprochenen LSP (vgl. Abschnitt 3.5.3) besagen, dass an allen Stellen in einem Programm, an denen ein Basistyp verwendet wird, es auch möglich sein sollte, jeden Subtyp zu verwenden. Zur Verdeutlichung nutze ich in den folgenden Beispielen eine einfache Klassenhierarchie `Sub extends Base`. Allgemein gilt, dass aufgrund der Vererbungsbeziehung die technische Ersetzbarkeit von Objekten der Basisklasse `Base` durch Objekte der abgeleiteten Klasse `Sub` möglich ist. Eine semantische Ersetzbarkeit erhält man aber nur dann, wenn die »is-a«-Beziehung und das LSP eingehalten werden.

Vorbetrachtungen

Anhand einer einfachen Methodendefinition `method(Base)` und eines Aufrufs dieser Methode mit einer Eingabe vom Typ `Sub` möchte ich einige Vorbetrachtungen anstellen, die das Verständnis der Formen der Varianz erleichtern:

```
// Definition
Base method(final Base in)
{
    return new Sub(in);             // Speziellerer Rückgabetyp -> Kovarianz
}

// Aufruf
final Base out = method(new Sub()); // Speziellerer Eingabetyp -> Kontravarianz
```

Für Methodenparameter gilt: Ein ***Eingabetyp darf spezieller sein*** als der von der Methode in der Signatur geforderte Eingabetyp, da in der Methode lediglich die Attribute und Methoden der Basisklasse zugreifbar sind und diese auch in der Subklasse vorhanden sind. Auch für ***Rückgabetypen darf ein speziellerer Typ zurückgeliefert*** werden, da ein verwendender Aufrufer nur die Attribute und Methoden der Basisklasse kennt.

Mit dem Begriff ***Varianz*** werden unter anderem die beiden gezeigten Typabweichungen Ko- und Kontravarianz beschrieben. Für beide gilt in diesem Beispiel, dass der Eingabeparameter sowie der Rückgabewert zur Kompilierzeit jeweils vom Typ `Base` sind, aber zur Laufzeit der Typ `Sub` verwendet wird. Diese Typabbildung wird durch gestrichelte Pfeile für die Typen des Eingabeparameters bzw. des Rückgabewerts in Abbildung 3-34 visualisiert.

Abbildung 3-34 *Grundlagen der Varianzformen*

Beim Rückgabewert der obigen Methode `Base method(Base)` deutet der gestrichelte Pfeil die Typabweichung an. Hier wird eine Instanz vom Typ `Sub` an eine Referenz vom Basistyp `Base` übergeben. Damit folgt die Typabweichung in ***gleicher*** Richtung wie bei der Typhierarchie. Diesen Sachverhalt drückt die sogenannte ***Kovarianz*** aus. Beim Eingabeparameter wird mit `Sub` ein speziellerer Typ übergeben und der gestrichelte Pfeil deutet an, dass eine Übergabe ***entgegen*** der Typhierarchie erfolgt. Das nennt man ***Kontravarianz***. Ändert sich der Typ nicht, so spricht man von ***Invarianz***.

Kovarianz: Auswirkungen auf Arrays

Arrays besitzen in Java kovariantes Verhalten. Die Kovarianz besagt in diesem Fall das Folgende: Da die Klasse `String` ein Subtyp von `Object` ist, gilt das auch für Arrays dieser Typen: Ein `String[]` stellt in Java somit einen Subtyp von `Object[]` dar. Verdeutlichen wir uns dies anhand des UML-Diagramms in Abbildung 3-35.

Abbildung 3-35 *Kovarianz für Array-Typen*

Aufgrund der Kovarianz ist eine Zuweisung eines `String[]` an ein `Object[]` möglich. Das scheint ganz selbstverständlich und bestimmt haben Sie derartige Zuweisungen von Arrays beliebiger Typen an ein `Object[]` schon des Öfteren in eigenen Programmen genutzt, etwa folgendermaßen:

```
final Object[] messages = new String[] { "Arrays", "sind", "kovariant" };
```

Obwohl diese Art der Zuweisung absolut praktisch ist und natürlich in der Handhabung erscheint, birgt das Ganze zur Laufzeit gewisse Fallstricke, die wir nun betrachten.

Fallstricke durch Kovarianz Die Kovarianz ermöglicht, dass beliebige, nicht primitive Array-Typen an eine Referenz vom Typ `Object[]` zugewiesen werden können, etwa wie oben ein `String[]`. Demnach könnte die Referenz zwischenzeitlich auch auf ein `Integer[]` und später wieder auf ein `String[]` verweisen. Aufgrund des Basistyps `Object` für die einzelnen Elemente können im `Object[]` beliebige Typen gespeichert werden. Ohne einen Blick auf die Zuweisung könnte es etwa zu folgender (versehentlicher) Fehlverwendung kommen, wenn man nur die obige Variable `messages` sieht und dort ein `Message`-Objekt wie folgt speichern möchte:

```
// Typfehler, weil Message nicht kompatibel zu String ist
messages[1] = new Message("Typfehler durch Kovarianz!");
```

Derartige Typinkompatibilitäten lassen sich allein auf Basis von Typinformationen zur Kompilierzeit nicht unterbinden. ***Daher kann für Arrays (insbesondere vom Typ `Object[]`) Typsicherheit zur Kompilierzeit nicht sichergestellt werden und erfordert eine zusätzliche Prüfung zur Laufzeit***. Deshalb speichern Arrays auch zur Laufzeit noch Typinformationen über die in ihnen gespeicherten Elemente. Wird, wie im Listing gezeigt, eine Zuweisung inkompatibler Typen vorgenommen, hier statt eines Werts vom Typ `String` eine Zuweisung eines `Message`-Objekts, so werden `java.lang.ArrayStoreException`s ausgelöst.

Hintergundwissen: Automatische Typerzeugung bei Arrays

Bei der Angabe eines Arrays im Sourcecode wird durch den Compiler dynamisch ein neuer Typ erzeugt, falls es diesen noch nicht gibt. Beim Kompilieren der Anweisung

```
final Object[] infos = new String[] { "Arrays", "sind", "kovariant" };
```

entstehen zwei neue Typen: `Object[].class` und `String[].class`.

Einfluss von Varianz beim Überschreiben von Methoden

Nachdem wir bereits ein gewisses Verständnis für die verschiedenen Formen der Varianz aufgebaut haben, möchte ich darstellen, wie sich Varianz auf die Definition und das Überschreiben von Methoden in einer Klassenhierarchie auswirkt.

Zur Veranschaulichung verwende ich die bereits gezeigte Typhierarchie `Sub extends Base`, die Eingabe- bzw. Rückgabetyp von Methoden einer Klassenhierarchie definiert. Die Klassenhierarchie besteht aus den zwei Klassen `BaseClass` und `SubClass` sowie der Vererbungsbeziehung `SubClass extends BaseClass`. Jede dieser Klassen definiert eine Methode `method()`. Je nach Varianzform nutzt diese einen Eingabeparameter bzw. Rückgabewert vom Typ `Base` bzw. `Sub`. Das kann man sich am besten mithilfe eines kleinen Modells in UML-Schreibweise verdeutlichen. In Abbildung 3-36 ist dies dargestellt.

Abbildung 3-36 *Varianzformen und Vererbung*

Die gezeigten Varianzformen führen nicht immer zum Überschreiben von Methoden. In der Java Language Specification[37] (JLS) ist definiert, dass ein Überschreiben nur möglich ist, wenn eine Methode einer Subklasse dieselbe Signatur wie die Methode der Basisklasse besitzt. Wir erkennen daher Folgendes:

- Beim ***invarianten Überschreiben*** sind in einer Basisklasse und in einer davon abgeleiteten Klasse die eingesetzten Typen eines Methodenparameters ***identisch***.
- Beim ***kovarianten Überschreiben*** folgt die Vererbungshierarchie der Rückgabetypen und die der einsetzenden Klassen der ***gleichen Richtung***. Ab JDK 5 wird dies in Form kovarianter Rückgabewerte unterstützt (vgl. Abschnitt 3.6.2).
- Wenn die Typhierarchie der jeweiligen Methodenparameter ***entgegengesetzt zur Vererbungshierarchie*** der verwendenden Klassen läuft, so spricht man von ***Kontravarianz***. In Java findet hierdurch *kein Überschreiben* statt: Stattdessen wird die Methode *überladen* und existiert zusätzlich zu der geerbten Methode.

[37] Diese ist online unter `http://docs.oracle.com/javase/specs/` verfügbar.

3.6.2 Kovariante Rückgabewerte

Nach diesen Grundlagen stelle ich im Folgenden kovariante Rückgabewerte anhand eines Beispiels vor. Überlegen wir uns dazu wieder ein Programm, das verschiedene grafische Elemente der Typen `CircleFigure`, `LineFigure` und `RectFigure` verwaltet: Diese Klassen erweitern eine gemeinsame Basisklasse `BaseFigure`. Nehmen wir an, diese Basisklasse besäße eine Methode `copy()`, um Elemente zu kopieren:

```
public abstract class BaseFigure
{
    public abstract BaseFigure copy();

    // ...
```

Jede konkrete Subklasse muss diese abstrakte Methode realisieren. Im Folgenden betrachten wir dies für kovariante Rückgabewerte, die seit Java 5 möglich sind. Damit kann im Beispiel die abgeleitete Klasse `CircleFigure` die Methode `copy()` mit einem spezielleren Rückgabetyp überschreiben als in der Basisklasse:

```
class CircleFigure extends BaseFigure
{
    private double radius;

    public CircleFigure(double radius)
    {
        this.radius = radius;
    }

    // Kovariante Rückgabe
    public CircleFigure copy()
    {
        return new CircleFigure(this.radius);
    }

    public double getRadius()
    {
        return radius;
    }
}
```

Abbildung 3-37 zeigt kovariante Rückgabetypen als UML-Klassendiagramm für die Spezialisierungen `CircleFigure`, `LineFigure` und `RectFigure`.

Abbildung 3-37 *Abstrakte Basisklasse und kovariante Rückgabewerte*

Betrachten wir nun den Einsatz für eine Referenzvariable vom Typ `CircleFigure`:

```
public static void main(final String[] args)
{
    final BaseFigure baseFigure = new CircleFigure(13);
    final CircleFigure circleFigure = new CircleFigure(7);

    // #1 Kovariante Rückgabe
    final CircleFigure circleTypeCopy = circleFigure.copy();
    final double radius = circleTypeCopy.getRadius();
    System.out.println("radius of copy: " + radius);

    // #2 Compile-Error
    // final CircleFigure circleTypeCopy = baseFigure.copy();

    // #3 Compile-Typ: BaseFigure / Runtime-Typ: CircleFigure
    final BaseFigure baseCopy = baseFigure.copy();
    if (baseCopy instanceof CircleFigure)
    {
        final CircleFigure circleBaseCopy = (CircleFigure)baseCopy;
        System.out.println("baseCopy compile type BaseFigure");
        System.out.println("baseCopy runtime type CircleFigure");
        System.out.println("radius of copy: " + circleBaseCopy.getRadius());
    }
}
```

Listing 3.4 *Ausführbar als* '**COVARIANTRETURNEXAMPLE**'

Das Programm COVARIANTRETURNEXAMPLE erzeugt folgende Ausgaben:

```
radius of copy: 7.0
baseCopy compile type BaseFigure
baseCopy runtime type CircleFigure
radius of copy: 13.0
```

Analyse Analysieren wir den Sourcecode ein wenig. Wir sehen, dass Aufrufe mit kovarianter Rückgabe nur dann einen Subtyp zurückliefern, wenn die Variable auch zum Zeitpunkt des Kompilierens vom entsprechenden Typ ist – in diesem Beispiel, wenn das Original auch vom Typ `CircleFigure` ist (Variante #1).

Bei Variante #2 besitzt das Objekt `baseFigure` zwar den *Laufzeittyp* `CircleFigure`, aber den *Kompiliertyp* `BaseFigure`. Dafür würde zwar eine polymorphe Wahl der Methode erfolgen, es wird jedoch der kovariante Rückgabetyp zur Kompilierzeit nicht unterstützt. Dies führt zu folgender Fehlermeldung: »`Type mismatch: cannot convert from BaseFigure to CircleFigure`«.

Die Variante #3 mit der Anweisung `baseFigure.copy()` führt zu einer Kopie und einer Zuweisung an die Variable `baseCopy` vom Typ `BaseFigure`. Startet man das Programm COVARIANTRETURNEXAMPLE, so entsteht tatsächlich ein `CircleFigure`-Objekt. Allerdings kann man – ohne expliziten Cast auf diesen Laufzeittyp – nicht auf die spezifische Methode `getRadius()`, sondern nur auf die Methoden des Kompiliertyps `BaseFigure` zugreifen.

3.7 Generische Typen (Generics)

Dieser Abschnitt stellt generische Typen (kurz ***Generics***) so weit vor, wie dies hilfreich ist, um die Beispiele und Erklärungen dieses Buchs nachvollziehen zu können.

3.7.1 Einführung

Bis JDK 5 waren die Containerklassen des Collections-Frameworks (vgl. Abschnitt 6.1) wie Listen, Sets und Maps untypisiert. Somit konnten dort Objekte beliebigen Typs verwaltet werden. Jedoch sind derartige heterogene Container nur für wenige Anwendungsfälle nützlich. Viel öfter ist gewünscht, gleichartige Objekte, also diejenigen eines bestimmten Typs, zu speichern – man spricht von einer homogenen Zusammensetzung. Ohne das Sprachfeature Generics oder eine selbstgeschriebene Containerklasse kann man diese Forderung nur durch eine geeignete Namensgebung, etwa `personList`, ausdrücken, nicht aber vom Compiler sicherstellen lassen. Seit JDK 5 lässt sich die typsichere Definition von Containerklassen mithilfe von Generics ohne weiteren eigenen Implementierungsaufwand erreichen. Es muss lediglich eine Typangabe bei der Definition einer Containerklasse erfolgen. Basierend darauf kann vom Compiler sichergestellt werden, dass dort nur gewünschte Typen verwaltet werden.

Nicht generische Container

Um die Problematik nicht typsicherer Container zu rekapitulieren bzw. besser nachvollziehen zu können, betrachten wir als Beispiel die Datenspeicherung von `Person`-Objekten in einer `ArrayList` ohne Typangabe. Man spricht auch von einem sogenannten ***Raw Type***, weil dieser keine Typinformationen über den Inhalt besitzt. Somit sind die Zugriffsmethoden, etwa `Object get(int)` und `add(Object)`, alle mit Eingabeparametern oder Rückgabewerten des Typs `Object` definiert. Dadurch können Objekte beliebiger Typen verarbeitet werden. Zur Demonstration möglicher Auswirkungen wird in eine Liste von `Person`-Objekten bewusst auch ein `Dog`-Objekt hinzugefügt. Danach wird über die Liste iteriert, die Elemente werden ausgelesen und ausgegeben:

```
public static void main(final String[] args)
{
    // Achtung: Nur zur Demonstration auf Generics verzichtet
    final List personList = new ArrayList();
    personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
    personList.add(new Person("Moritz", LocalDate.now() "Musterstadt"));
    personList.add(new Dog("Sarah vom Auetal"));

    for (int i = 0; i < personList.size(); i++)
    {
        // Explizite Typumwandlung notwendig zum Methodenaufruf
        final Person person = (Person) personList.get(i);
        System.out.println(person.getName() + " aus " + person.getCity());
    }
}
```

Listing 3.5 *Ausführbar als* '**OLDSTYLELIST**'

Der gezeigte Sourcecode kompiliert ohne Fehler, aber zur Laufzeit gibt es Probleme: Als Folge der expliziten Typumwandlung des Eintrags vom Typ `Dog` in eine Referenz vom Typ `Person` wird eine `ClassCastException` ausgelöst: Die Klasse `Dog` ist selbstverständlich keine Subklasse der Klasse `Person`.

Die Typumwandlung von `Object` auf `Person` ist erforderlich, damit ein Zugriff auf die Methoden `getName()` und `getCity()` der Klasse `Person` möglich wird, um den Namen einer Person und deren Wohnort ausgeben zu können. Aufgrund des Namens `personList` erfolgt hier der Cast oder eine Typprüfung, da man davon ausgeht, dass die gespeicherten Elemente vom Typ `Person` sind. Wie das Beispiel zeigt, ist dies aber nicht garantiert und führt zu dem Fehler. Wie kann man es also besser machen?

Typsichere, generische Container

Seit JDK 5 lassen sich praktischerweise Containerklassen durch die Angabe von Typparametern auf einen Typ festlegen. Dies geschieht mithilfe der Spitzen-Klammern-Notation, in der Deklaration z. B. mit `List<Person>`, bei der Definition hier als `ArrayList<Person>`. Basierend auf den Typangaben kann der Compiler bereits zur Kompilierzeit eventuell vorhandene Typinkompatibilitäten aufdecken und dadurch sicherstellen, dass nur Objekte des gewünschten Typs (und auch Subtypen davon), hier `Person`, in den Container aufgenommen werden können.

Schauen wir uns nun die typsichere Definition einer Liste von `Person`-Objekten mithilfe von Generics an:

```
public static void main(final String[] args)
{
    // Typsichere Definition mit Generics
    final List<Person> personList = new ArrayList<Person>();

    personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
    personList.add(new Person("Moritz", LocalDate.now(), "Musterstadt"));
    // personList.add(new Dog(”Sarah vom Auetal”)); // Compile-Error

    for (int i = 0; i < personList.size(); i++)
    {
        // Typsicheres Auslesen
        final Person person = personList.get(i);
        System.out.println(person.getName() + " aus " + person.getCity());
    }
}
```

Listing 3.6 *Ausführbar als* **'NEWSTYLELIST'**

Im Gegensatz zum Programm OLDSTYLELIST, das eine `ClassCastException` auf der Konsole protokolliert, wird im Programm NEWSTYLELIST das Hinzufügen eines `Dog`-Objekts direkt schon durch den Compiler verhindert. Demnach werden folgende zwei Einträge ausgegeben:

```
Max aus Musterstadt
Moritz aus Musterstadt
```

Wie das Beispiel zeigt, lassen sich nunmehr wirklich nur `Person`-Referenzen (oder Subtypen) in der so typisierten `ArrayList<Person>` verwalten. Es entfällt die zuvor notwendige explizite Typumwandlung und die Methodensignaturen nutzen den angegebenen Typ, hier `Person`, statt wie früher den Typ `Object`.

Diamond Operator und Vereinfachungen bei Typangaben mit JDK 7

Der Einsatz von Generics erforderte vor JDK 7 etwas zusätzlichen Schreibaufwand durch die Angabe der Typinformation bei der Deklaration und deren Wiederholung bei der Definition etwa folgendermaßen:

```
final Map<String, Set<String>> typeSafeMap = new HashMap<String, Set<String>>();
```

Dies ist unleserlich und enthält Redundanzen, da die Typparameter bereits bekannt sind. Seit JDK 7 kann man den sogenannten ***Diamond Operator*** '<>' zur Abkürzung der Schreibweise wie folgt verwenden:

```
// Typ muss nicht wiederholt werden
final Map<String, Set<String>> newJDK7StyleMap = new HashMap<>();
```

Auf eine kleine Einschränkung möchte ich noch hinweisen: Bis JDK 9 konnte man den Diamond Operator leider nicht für anonyme innere Klassen nutzen. Seit JDK 9 ist das aber praktischerweise möglich (vgl. Abschnitt 5.4).

Definition eigener generischer Klassen

Wir wissen nun, wie man die Containerklassen des Collections-Frameworks typsicher gestaltet, doch oftmals ist es darüber hinaus wünschenswert, eigene typisierte Klassen erstellen zu können. Betrachten wir als einfaches Beispiel einen Datencontainer für Wertepaare beliebiger Typen.

Realisierung ohne Generics und sich daraus ergebende Probleme Ohne Verwendung von Generics muss dieser Datencontainer zunächst ganz allgemein mit Referenzen vom Typ `Object` folgendermaßen implementiert werden:

```
public final class Pair
{
    private final Object first;
    private final Object second;

    public Pair(final Object first, final Object second)
    {
        this.first = first;
        this.second = second;
    }

    public final Object getFirst()        { return first; }
    public final Object getSecond()       { return second; }
}
```

Diese Realisierung bietet keine Einschränkung, welche Typen eingefügt werden können. Möchte man diese Klasse typsicher für diverse Typkombinationen ohne Einsatz von Generics bereitstellen, so müssten für jede gewünschte Typkombination jeweils eigene Klassen erstellt werden. Dies würde zu Sourcecode-Duplikation und zu einer wachsenden Menge ähnlicher Klassen führen, und zwar abhängig von der Anzahl gewünschter Typkombinationen, wobei die jeweilige Implementierung in den Typen der Attribute `first` und `second` entsprechend variiert. Man erkennt, dass es wenig sinnvoll ist, Datencontainer in dieser Weise auf spezielle Typen der Einzelelemente festzulegen, wenn man eine allgemeingültige Lösung sucht. Wie geht es also besser?

Realisierung mit Generics Anstatt die Klasse `Pair` für jede benötigte Typkombination wieder erneut zu programmieren, wollen wir eine generische, für alle möglichen gewünschten Typkombinationen verwendbare Definition bereitstellen. Durch den Einsatz von Generics wird genau dies möglich.

Nachfolgend definieren wir den Container `Pair` mit zwei formalen Typparametern `T1` und `T2`, die als Platzhalter für konkrete Typen beim Einsatz dienen, wie folgt:[38]

```
public final class Pair<T1, T2>
{
    private final T1 first;
    private final T2 second;

    public Pair(final T1 first, final T2 second)
    {
        this.first = first;
        this.second = second;
    }

    public final T1 getFirst()       { return first; }
    public final T2 getSecond()      { return second; }
}
```

Erst beim Einsatz einer solchen generischen Klasse werden die formalen Typparameter vom Compiler durch die im Sourcecode angegebenen konkreten Typen ersetzt.

Betrachten wir als Anwendungsfall die Speicherung und Verwaltung von Personen und deren Spitznamen. Mithilfe der vorgestellten Klasse `Pair` soll die Kombination aus Spitzname und Person modelliert werden. Dazu substituiert man die beiden formalen Typparameter `T1` und `T2` durch die gewünschten Typen `String` und `Person`:

```
final Person wizard = new Person("Wizard", LocalDate.of(1970, 7, 17), "Kiel");
final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Bremen");

final Pair<String, Person> pair1 = new Pair<>("Dark", wizard);
final Pair<String, Person> pair2 = new Pair<>("Iron", mike);
```

[38] In der Regel werden als Typplatzhalter einzelne Zeichen verwendet, die bei Bedarf, wie hier, eine ergänzende Nummerierung erhalten. Gebräuchlich sind die Kürzel `E` für Elemente von Containerklassen und `T` für beliebige Typen.

Typeinschränkungen für Klassen Teilweise möchte man bei der Definition einer generischen Klasse den erlaubten Typ eines zu ersetzenden Typparameters genauer spezifizieren. Das kann man durch folgende Notation mit `extends` und `&` erreichen:

```
public class GenericClass<T extends BaseType & Interface1 & Interface2>
```

Diese Angabe sichert folgende Eigenschaften für den Typ `T` ab:

- `T` muss die nach `extends` angegebene Basisklasse bzw. den Basistyp `BaseType` besitzen (oder selbst vom Typ `BaseType` sein).
- Optional können mit `&` Interfaces spezifiziert werden: `T` muss im Beispiel dann auch die Interfaces `Interface1` und `Interface2` implementieren. Diese Bedingung ist auch erfüllt, wenn eine beliebige Basisklasse von `T` dies tut.

Vor dem ersten '`&`' können Typangaben verwendet werden, also sowohl einer Klasse als auch eines Interface. Nach dem '`&`' können lediglich Interfaces angegeben werden.

Typeinschränkungen für (statische) Methoden Für Methoden kann auf ähnliche Weise eine Typeinschränkung erfolgen. Nehmen wir an, eine statische Hilfsmethode `doSomething()` soll Listen übergeben bekommen, die auf den Typ `BaseFigure` oder Subtypen davon festgelegt sind, etwa `List<RectFigure>`. Eine mögliche Schreibweise ist folgende:

```
public static void doSomething(final List<? extends BaseFigure> figures)
```

Alternativ kann man die Methode auch explizit als generisch definieren. Dazu wird der formale Typparameter vor dem Rückgabetyp in spitzen Klammern notiert:[39]

```
public static <T extends BaseFigure> void doSomething(final List<T> figures)
```

Analog kann man nicht statische Methoden mit einem Typparameter versehen, etwa wenn dieser abweichend vom Typparameter der Klassendefinition ist. Gebräuchlich ist aber vor allem die gezeigte Form für statische Methoden.

Die hier genutzte spezielle Notation `? extends` bzw. `T extends` mutet etwas merkwürdig an. Sie drückt Kovarianz aus, die vor allem für Containerklassen und Generics von Interesse ist und später im Detail in Abschnitt 6.4 besprochen wird.

[39] Wodurch im Unterschied zur vorherigen Variante in dieser `T`-Variante die `BaseFigure`-Spezialisierung »festgehalten« wird.

3.7.2 Generics und Auswirkungen der Type Erasure

Als Generics mit JDK 5 eingeführt werden sollten, stand man vor der Herausforderung, dies kompatibel zur JVM und den bisherigen Klassen des JDKs, insbesondere des Collections-Frameworks, realisieren zu müssen. Man entschied sich dazu, weder dynamisch neue Klassen zu erzeugen, wie dies für Arrays geschieht, noch den von der JVM generierten und ausgeführten Bytecode anzupassen.

Dies war nur durch einen Trick möglich: Generics sind mithilfe der sogenannten ***Type Erasure*** realisiert. Das bedeutet, dass die im Sourcecode typisierten Klassen durch den Compiler auf untypisierte Klassen bzw. die Parameter auf den allgemeinsten Typ `Object` abgebildet werden. Aus einer `ArrayList<Person>` wird dadurch beim Kompilieren eine `ArrayList` ohne Typinformationen. Als Folge der Type Erasure existieren also zur Laufzeit nur noch untypisierte Klassen.[40] Um das zu verstehen, schauen wir wieder auf eine Liste mit `Person`-Objekten:

```
final List<Person> personList = new ArrayList<>();

personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
final Person firstPerson = personList.get(0);
```

Dieser Sourcecode wird so kompiliert, als ob man Folgendes programmiert hätte:

```
final List personList = new ArrayList();

personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
final Person firstPerson = (Person)personList.get(0);
```

Das Entfernen des Typparameters und das Hinzufügen des Casts sind aber noch die am wenigsten weitreichenden Konsequenzen. Damit die Zuweisungen wie im Sourcecode angegeben funktionieren, muss der Compiler zum einen Casts passend in den Bytecode einfügen und zum anderen muss nun beim Kompilieren extrem streng geprüft werden, dass zur Laufzeit keine Typinkompatibilitäten auftreten können. Mögliche Probleme werden durch Warnungen signalisiert. Nur wenn keine Warnungen auftreten, sind keine Typinkompatibilitäten erkannt worden.

Insgesamt führt die Type Erasure zu einigen Einschränkungen und Merkwürdigkeiten beim Einsatz von Generics, etwa bei dynamischen Typabfragen zur Laufzeit, dem Erzeugen von Objekten mit generischem Typ und vor allem in Kombination mit Collections und Arrays. Im Folgenden stelle ich Auswirkungen sowie potenzielle Fallstricke vor und zeige mögliche Lösungen auf.

[40] Allerdings verbleiben einige Hinweise zu generischen Definitionen von Attributen und Methodenparametern im Bytecode. Darauf gehe ich später in Abschnitt 12.1.4 ein.

Auswirkung 1: Fehlende Typinformationen bei dynamischen Typabfragen

Wie schon in Abschnitt 3.6.1 beschrieben, erzeugt der Compiler bei der Definition eines Arrays dynamisch einen neuen Typ, falls es diesen noch nicht gab. Rekapitulieren wir dies am Beispiel von zwei Arrays für Spitznamen und Personen:

```
final String[] nicknames = { "Dragonman", "Iron Mike", "Lordmaster" };

final LocalDate feb7th = LocalDate.of(1971, 2, 7);
final LocalDate jul17th = LocalDate.of(1970, 7, 17);
final LocalDate aug22th = LocalDate.of(1970, 8, 22);

final Person[] bowlingPeople = { new Person("Sven", aug22th, "Kiel"),
                                 new Person("Michael", feb7th, "Bremen"),
                                 new Person("Andreas", jul17th, "Kiel") };
```

Durch diese Anweisungen entstehen zwei neue Array-Typen: `String[].class` und `Person[].class`. Niemals würde man auf die Idee kommen, dass die Referenzen `nicknames` und `bowlingPeople` vom gleichen Typ wären. Anders ausgedrückt: Nur weil beide Arrays sind, gilt ***nicht*** `String[].class == Person[].class`. Schreiben wir etwas Sourcecode, um das zu überprüfen:

```
// Compile-Error: Incompatible operand types Class<String[]> and Class<Person[]>
// final boolean sameType1 = (String[].class == Person[].class);
// final boolean sameType2 = (nicknames.getClass() == bowlingPeople.getClass());

final boolean sameType3 = (nicknames.getClass().equals(
                           bowlingPeople.getClass()));

System.out.println(sameType3); // false
```

Tatsächlich führen die gezeigten Vergleiche sogar zu Kompilierfehlern: Das ist zunächst verwunderlich, da man hier eher erwarten würde, dass der Vergleich fehlschlagen und der Wert `false` zurückgegeben würde. Ist keine Typkompatibilität gegeben, kommt es jedoch zu einem Kompilierfehler mit der obigen Fehlermeldung. Ein Vergleich der Klassen mit `equals(Object)` liefert den Wert `false` und zeigt damit, dass es sich tatsächlich um unterschiedliche Klassen handelt.

Typabfragen bei Generics Setzen wir das Beispiel fort und verwenden nun Generics. Wir ändern die speichernde Datenstruktur von einem Array zu einer `ArrayList<E>`, um zu schauen, welche Auswirkungen dies für Typabfragen besitzt. Wir nutzen hier die für unsere Zwecke praktische statische Hilfsmethode `Arrays.asList(T...)` aus dem Collections-Framework (vgl. Abschnitt 6.3.1). Diese erzeugt aus einem Array vom Typ `T` eine typisierte, unmodifizierbare Liste. Betrachten wir das Ganze erneut:

```
final List<String> nicknamesList = Arrays.asList(nicknames);
final List<Person> bowlingPeopleList = Arrays.asList(bowlingPeople);

// ACHTUNG: Compile-Error
// final boolean sameType1 = (List<String>.class == List<Person>.class);
final boolean sameType2 = (nicknamesList.getClass() ==
                           bowlingPeopleList.getClass());

System.out.println(sameType2); // true
```

Listing 3.7 *Ausführbar als* '**TYPEERASUREEXAMPLE**'

Das Programm TYPEERASUREEXAMPLE produziert folgende Ausgaben:

```
false
true
```

Daran sieht man Folgendes: Ein direkter Vergleich der generischen Typen scheitert wiederum bereits an Kompilierfehlern. Um die Typinformationen zu den generischen Klassen zu erhalten, nutzen wir einen Aufruf von `getClass()`. Interessanterweise führt der Vergleich nun nicht mehr zu einem Kompilierfehler, sondern die ermittelten Typinformationen der Variablen `nicknamesList` und `bowlingPeopleList` sind ***gleich***. Daraus folgt: ***Durch die Type Erasure existiert tatsächlich nur genau ein Typ für jede mit verschiedenen Typparametern erzeugte Containerklasse***. Demnach gibt es also weder einen Typ `List<Person>.class` noch einen Typ `List<String>.class`, sondern lediglich den Typ `List.class`. Somit lässt sich also weder mit `instanceof` noch mit `getClass()` prüfen, ob eine Collection auf einen gewünschten Typ festgelegt ist. ***Es sind keine Typprüfungen generischer Containerklassen möglich***, um etwa vor dem Einfügen eines `Person`-Objekts in eine Liste abzuprüfen, ob diese tatsächlich auf die Speicherung von Personen ausgelegt ist. Das ist eine recht bedeutende Einschränkung.

Auswirkung 2: Probleme beim Erzeugen von generischen Objekten

Weil beim Kompilieren ein generischer Typparameter `T` durch die Type Erasure in den Typ `Object` umgewandelt wird, stehen zur Laufzeit nicht die benötigten Informationen zum gewünschten Typ `T` bereit, um folgenden Konstruktoraufruf auszuführen:

```
// Achtung: Compile-Error
new T()
```

Wenn man aber Instanzen generischer Klassen dynamisch erzeugen möchte, wäre ein solcher Konstruktoraufruf eine wünschenswerte Funktionalität. Es gibt zwei mögliche Lösungen, die ausnutzen, dass generische Typen als Rückgabe erlaubt sind:

1. Man definiert eine Methode zur Konstruktion gemäß dem Muster ERZEUGUNGSMETHODE bzw. FABRIKMETHODE (vgl. Abschnitt 18.1.1 bzw. 18.1.2).
2. Man definiert ein Interface mit einer oder mehreren Methoden zur Konstruktion gemäß dem Muster ABSTRAKTE FABRIK (siehe Literatur in Abschnitt 18.4).

Lösung 1 Zum Erzeugen definiert man folgende Methode `createNewTypedObject(Class<T>)`, die mit Reflection (vgl. Abschnitt 12.1) arbeitet. Für Klassen eines beliebigen Typs `T` kann man wie folgt eine neue Instanz erzeugen, sofern der Typ `T` einen Defaultkonstruktor besitzt:

```
@SuppressWarnings("unchecked")
public static <T> T createNewTypedObject(final Class<T> clazz)
{
    try
    {
        // Konstruktoraufruf per Reflection
        return (T) clazz.newInstance();
    }
    catch (final InstantiationException e)
    {
        // Keine Instanziierung möglich
        throw new IllegalStateException("Keine Instanziierung möglich", e);
    }
    catch (final IllegalAccessException e)
    {
        // Kein Zugriff möglich
        throw new IllegalStateException("Kein Zugriff möglich", e);
    }
}
```

In diesem Beispiel nutzen wir neben Reflection und `newInstance()` auch die Annotation `@SuppressWarnings("unchecked")`, die Typwarnungen unterdrückt. Diese Warnungen sollten im Normalfall nicht ignoriert werden, da ansonsten Fehler zur Laufzeit drohen. Hier ist der Einsatz der Annotation sinnvoll, da durch die Anweisungen keine Typinkompatibilität entsteht.

Lösung 2 Die zweite Lösung ist etwas komplizierter. Zum Erzeugen von Objekten führt man eine generische, abstrakte Klasse `AbstractFactory<T>` ein, die ebenfalls eine Methode `createNewTypedObject()` wie folgt deklariert:

```
public abstract class AbstractFactory<T>
{
    abstract T createNewTypedObject();
}
```

Zur Konstruktion spezieller Typen werden dann konkrete Realisierungen dieser abstrakten Klasse mit spezifischen Realisierungen der Methode `createNewTypedObject()` bereitgestellt. Wollte man zum Beispiel auf diese Weise `Person`-Objekte konstruieren, so implementiert man dazu folgende Fabrikklasse `PersonFactory`:

```
public final class PersonFactory extends AbstractFactory<Person>
{
    public Person createNewTypedObject()
    {
        return new Person();
    }
}
```

Einsatz der Lösungen Betrachten wir beide Lösungsmöglichkeiten (die Fabrikmethode bzw. die abstrakte Fabrik), um ein `Person`-Objekt zu konstruieren:

```
public static void main(final String[] args)
{
    // Variante 1 - Konstruktion
    final Person newPerson1 = createNewTypedObject(Person.class);

    // Variante 2 - Konstruktion
    final AbstractFactory<Person> factory = new PersonFactory();
    final Person newPerson2 = factory.createNewTypedObject();

    // Initialisierung für beide Varianten
    initPersonAttributes(newPerson1, "Mike1", "Aachen", LocalDate.now());
    System.out.println("Person1: " + newPerson1);

    initPersonAttributes(newPerson2, "Mike2", "Bremen", LocalDate.now());
    System.out.println("Person2: " + newPerson2);
}

private static void initAttributes(final Person newPerson, final String name,
                                   final String city, final LocalDate birthday)
{
    newPerson.setName(name);
    newPerson.setCity(city);
    newPerson.setBirthday(birthday);
}
```

Listing 3.8 *Ausführbar als* **'GENERICSCREATIONEXAMPLE'**

Anhand dieses Beispiels wird allerdings auch ein Nachteil beider Varianten der Konstruktion deutlich: Eine Erzeugung über eine Fabrikmethode bzw. eine abstrakte Fabrik erfordert die Definition eines Defaultkonstruktors in der Klasse. Weil ein solcher aber lediglich eine Defaultinitialisierung vornimmt und anschließend die so erzeugten Objekte mit der gewünschten Wertebelegung zu versehen sind, müssen Schreibzugriffe auf Attribute erfolgen können. Das erfordert die Bereitstellung und den Aufruf diverser (eventuell sogar öffentlicher) `set()`-Methoden. Warum dies für die Kontrolle des Objektzustands problematisch sein kann, wurde in Abschnitt 3.1.5 besprochen.

Auswirkung 3: Keine generischen statischen Variablen möglich

Wir haben bereits einige durch die Type Erasure ausgelöste Fallstricke kennengelernt. Auch die Definition generischer statischer Attribute bereitet Probleme. Folgende Definition ist *nicht* möglich, sondern es kommt zu einem Kompilierfehler:

```
// Achtung: Diese Klasse lässt sich nicht kompilieren!
public class StaticGenericDataStore<T>
{
    // Compile-Error: Cannot make a static reference to the non-static type T
    private static T value;

    public static void setValue(final T newValue)
    {
        value = newValue;
    }
```

```
    public static T getValue()
    {
        return value;
    }
}
```

Klassen können demnach keine Attribute enthalten, die sowohl generisch als auch statisch sind. Der Grund ist folgender: Typparameter gehören, wie oben festgestellt, zu Instanzen und nicht zur Klasse an sich. Statische Attribute und Methoden gehören aber zur Klasse, woraus dann das Gesagte folgt.

Auswirkung 4: Probleme bei der Definition generischer Arrays

Beim Einsatz von Arrays in Kombination mit Generics kommt es aufgrund der Type Erasure mitunter zu Typwarnungen und Fehlern beim Kompilieren.

Schauen wir uns ein einfaches Beispiel an. Es sollen `Pair`-Objekte in einem Array vom Typ `Pair<Integer, String>[]` gespeichert werden. Wie im Listing angedeutet, kommt es dabei jedoch zu folgendem Kompilierfehler:

```
final Pair<Integer, String> PAIR1 = new Pair<>(1, "One");
final Pair<Integer, String> PAIR2 = new Pair<>(2, "Two");

// Compile-Error: Cannot create a generic array of Pair<Integer, String>
// final Pair<Integer, String>[] PAIRS = { PAIR1, PAIR2 };
final Pair[] PAIRS = { PAIR1, PAIR2 };
```

Es ist nur die zweite gezeigte, nicht typisierte Definition `Pair[] PAIRS` erlaubt. Wie lässt sich das erklären? Bekanntermaßen sind Arrays kovariant ausgelegt und können Typsicherheit zur Kompilierzeit nicht zu 100 % garantieren (vgl. Abschnitt 3.6.1). Um Typkompatibilität sicherzustellen, müssen Arrays auch zur Laufzeit Typinformationen besitzen und auswerten. Weil für Generics durch die Type Erasure die Typangaben aus dem Sourcecode entfernt werden, besäße ein wie oben gewünscht definiertes Array `Pair<Integer, String>[]` zur Laufzeit keine Typangaben über die generische Klasse `Pair` mehr, sondern wäre vom Typ `Pair<Object, Object>[]`. Damit fehlen die Typangaben zum Inhalt, die aber zur Laufzeitprüfung benötigt würden.

Nehmen wir kurz einmal an, das Ganze wäre doch erlaubt. Dann könnte man im Array auch `Pair`-Objekte speichern, die vom Typ her inkompatibel sind, etwa statt `Pair<Integer, String>` eine Kombination `Pair<String, LocalDate>` wie folgt:

```
final Object[] objects = intStringPairs = new Pair<Integer, String>[7];

// Typinkompatibilität
objects[0] = new Pair<String, LocalDate>("Now", LocalDate.now());
```

Während nicht generische Arrays bei Typinkompatibilitäten `ArrayStoreException`s auslösen können, wäre dies bei der Speicherung generischer Typen in Arrays nicht möglich, da Konflikte weder vom Compiler noch zur Laufzeit durch die JVM erkannt wer-

den könnten. Um Typinkompatibilitäten also zur Laufzeit ausschließen zu können, ist die Definition generischer Arrays in Java nicht erlaubt.

Auswirkung 5: Kein direktes Erzeugen generischer Arrays möglich

Auch das Erzeugen eines generischen Arrays ist aufgrund der Type Erasure nicht möglich, d. h., folgende Definition führt zu dem gezeigten Kompilierfehler. Da Arrays lediglich Referenzen auf Objekte verwalten, könnte man auf die Idee kommen, ein Array benötigter Größe des Typs `Object[]` zu erzeugen und dieses anschließend auf den Typ `T[]` zu casten. Dies führt beim Kompilieren zu einer Warnung: »`Type safety: Unchecked cast from Object[] to T[]`«, da ein `Object[]` einem Subtyp `T[]` zugewiesen wird:

```
public static <T> T[] createTypedArray(final int size)
{
    // Compile-Error: Cannot create a generic array of T
    //final T[] typedArray = new T[size];
    final T[] typedArray = (T[])new Object[size];
    return typedArray;
}
```

Schlimmer noch: Der nachfolgend gezeigte Aufruf löst zur Laufzeit eine `ClassCastException` mit dem Hinweis »`class [Ljava.lang.Object; cannot be cast to class [Ljava.lang.String;`« aus:

```
public static void main(final String[] args)
{
    final String[] texts = createTypedArray(10);
    texts[0] = "Test";
}
```

Listing 3.9 *Ausführbar als* **'WRONGGENERICARRAYCREATION'**

Rekapitulieren wir noch mal: ***Arrays speichern und besitzen Typinformationen auch zur Laufzeit***. Aufgrund ihrer kovarianten Ausrichtung kann man zwar z. B. eine Referenz auf ein `String[]` einer Referenz an ein `Object[]` zuweisen – umgekehrt führt das natürlich zu Problemen: Man kann einem `String[]` selbstverständlich kein `Object[]` zuweisen. Genau dies wird aber im obigen Beispiel ausgeführt: In der Hilfsmethode `createTypedArray(int)` wird ein Array vom Typ `Object[]` erzeugt und auf den Typ `T[]` gecastet. Durch Type Erasure wird daraus jedoch der Typ `Object[]`. Dieser wird dann in der `main()`-Methode dem Typ `String[]` zugewiesen. Als Folge kommt es zu einer `ClassCastException`, da eine Typinkompatibilität vorliegt.

Hinweis: Typwarnungen bei Generics

Ganz allgemein gilt, dass man ***Typwarnungen*** im Zusammenhang mit Generics beim Kompilieren ***unbedingt Beachtung schenken*** sollte. Nur wenn man sich absolut sicher ist, dass wirklich Typkompatibilität gegeben ist, kann man eine Warnung mit der Annotation `@SuppressWarnings("unchecked")` unterdrücken.

Lösung Ich greife dem in Abschnitt 12.1 eingeführten Thema Reflection etwas vor, indem ich die Klasse `java.lang.reflect.Array` nutze. Ich verwende deren Methode `newInstance(Class<T>, int)`, um dynamisch zur Laufzeit ein Array vom übergebenen Typ und der angegebenen Größe mit der Hilfsmethode `createTypedArray(Class<T>, int)` wie folgt zu erzeugen:

```
@SuppressWarnings("unchecked")
public static <T> T[] createTypedArray(final Class<T> clazz, final int size)
{
    return (T[]) Array.newInstance(clazz, size);
}

public static void main(final String[] args)
{
    final String[] texts = createTypedArray(String.class, 2);

    System.out.println("Type: " + texts.getClass().getSimpleName());
    System.out.println("Size: " + texts.length);
    texts[0] = "Test1";
    texts[1] = "Test2";
    System.out.println("Content: " + Arrays.toString(texts));
}
```

Listing 3.10 *Ausführbar als* **'GENERICARRAYCREATION'**

In diesem Fall ist Typsicherheit gegeben, wie es die folgenden Ausgaben des Programms GENERICARRAYCREATION zeigen. Daher kann man hier die Annotation `@SuppressWarnings("unchecked")` einsetzen, um eine Typwarnung zu verhindern. Allerdings sollte man dabei generell vorsichtig sein und alles penibel prüfen.

```
Type: String[]
Size: 2
Content: [Test1, Test2]
```

3.8 Weiterführende Literatur

Dieses Kapitel hat einen fundierten Einstieg in das Thema Objektorientierung gegeben. Dabei wurde das Thema Generics so weit behandelt, dass ein Einsatz in der Praxis erleichtert wird. Wir haben auch ein erstes Verständnis für Varianzformen gewonnen und mögliche Fallstricke der Type Erasure kennengelernt. Ein paar problematische Details von Generics in Kombination mit den Containerklassen des Collections-Frameworks werden später in Abschnitt 6.4 beleuchtet.

Wissenswertes zur objektorientierten Programmierung und zu Generics finden Sie in den folgenden Büchern und Onlinequellen:

- »**Object-Oriented Software Development Using Java**« von Xiaoping Jia [43]
  Dieses Buch beschäftigt sich intensiv mit den Grundlagen eines gelungenen objektorientierten Designs von Software. Dabei werden viele Sprachelemente sowie diverse weitere interessante Themen wie Collections, Multithreading und verteilte Programme besprochen.
- »**Core Java 2 – Grundlagen**« von Cay S. Horstmann und Gary Cornell [32]
  Der erste Band dieses Buchs bietet einen Einstieg in den objektorientierten Entwurf mit Java und stellt dabei unter anderem auch Generics vor.
- »**Design mit Java**« von Peter Coad und Mark Mayfield [11]
  Coad und Mayfield stellen anhand von Beispielen den objektorientierten Entwurf mit Java dar. Dabei thematisieren sie insbesondere Komposition und Vererbung sowie den Einsatz von Interfaces.
- »**Fortgeschrittene Programmierung mit Java 5**« von Johannes Nowak [56]
  Nowak beschäftigt sich intensiv mit den in JDK 5 hinzugekommenen Sprachfeatures Generics und den Concurrency Utilities.
- »**Java Generics and Collections**« von Maurice Naftalin und Philip Wadler [55]
  Dieses Buch ist eine gute Ergänzung des zuvor genannten Buchs, da es besonders auf die Neuerungen im JDK 5 eingeht – im Speziellen auf Generics und einige Erweiterungen des Collections-Frameworks.
- »**A Programmers's Guide to Java SCJP Certification**« von Khalid A. Mughal und Rolf W. Rasmussen [53]
  Dieses Buch bietet einen präzisen, etwas formalen Einstieg in die SCJP-Zertifizierung und behandelt detailliert Generics.
- »**Generics in the Java Programming Language**« von Gilad Bracha
  `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`
- »**Java Generics FAQ**« von Angelika Langer
  `http://www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html`

4 Lambdas, Methodenreferenzen und Defaultmethoden

Mit Lambda-Ausdrücken (kurz ***Lambdas***) wurde ein von vielen Entwicklern heiß ersehntes Sprachkonstrukt mit Java 8 eingeführt, das bereits in ähnlicher Form in verschiedenen anderen Programmiersprachen wie C#, Groovy, Python und Scala erfolgreich genutzt wird. Der Einsatz von Lambdas erfordert zum Teil eine andere Denkweise und führt zu einem neuen Programmierstil, der dem Paradigma der ***funktionalen Programmierung*** folgt. Mithilfe von Lambdas lassen sich Lösungen oftmals auf elegante Art und Weise formulieren, weshalb diverse Funktionalitäten im Collections-Framework und an anderen Stellen des JDKs auf Lambdas umgestellt wurden.

4.1 Einstieg in Lambdas

Das Sprachkonstrukt Lambda kommt aus der funktionalen Programmierung. Ein ***Lambda*** ist ein Behälter für Sourcecode und ähnelt einer Methode, besitzt im Gegensatz dazu jedoch keinen Namen. Zudem findet sich keine explizite Angabe eines Rückgabetyps oder potenziell ausgelöster Exceptions.

4.1.1 Syntax von Lambdas

Vereinfacht ausgedrückt kann man einen Lambda am ehesten als anonyme Methode mit folgender Syntax und spezieller Kurzschreibweise auffassen:

```
(Parameter-Liste) -> { Ausdruck oder Anweisungen }
```

Ein paar einfache Beispiele für Lambdas sind die Addition von zwei Zahlen vom Typ `int`, die Multiplikation eines `long`-Werts mit dem Faktor 2 oder eine parameterlose Funktion zur Ausgabe eines Textes auf der Konsole. Diese Aktionen kann man als Lambdas wie folgt schreiben:

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
() -> { String msg = "Lambda"; System.out.println("Hello " + msg); }
```

Tatsächlich sehen diese Anweisungen recht unspektakulär aus, und insbesondere wird klar, dass ein Lambda lediglich ein Stück ausführbarer Sourcecode ist, der

- keinen Namen besitzt, sondern lediglich Funktionalität, und dabei
- keine explizite Angabe eines Rückgabetyps und
- keine Deklaration von Exceptions erfordert und erlaubt.[1]

Lambdas im Java-Typsystem

Wir haben bisher gesehen, dass sich einfache Berechnungen mithilfe von Lambdas ausdrücken lassen. Wie können wir diese aber nutzen und aufrufen? Versuchen wir zunächst, einen Lambda einer `java.lang.Object`-Referenz zuzuweisen, so wie es mit jedem anderen Objekt in Java möglich ist:

```
// Compile-Error: incompatible types: Object is not a functional interface
Object greeter = () -> { System.out.println("Hello Lambda"); };
```

Die gezeigte Zuweisung wird nicht unterstützt und führt zu einem Kompilierfehler. Die Fehlermeldung gibt einen Hinweis auf inkompatible Typen und verweist darauf, dass `Object` kein Functional Interface ist. Aber was ist ein Functional Interface?

Besonderheit: Lambdas im Java-Typsystem

Bis JDK 8 konnte in Java jede Referenz auf den Basistyp `Object` abgebildet werden. Mit Lambdas existiert nun ein Sprachelement, das nicht direkt dem Basistyp `Object` zugewiesen werden kann, sondern nur an Functional Interfaces.

4.1.2 Functional Interfaces und SAM-Typen

Ein ***Functional Interface*** ist eine neue Art von Typ, die mit JDK 8 eingeführt wurde, und repräsentiert ein Interface mit genau einer abstrakten Methode. Ein solches wird auch ***SAM-Typ*** genannt, wobei SAM für Single Abstract Method steht. Diese Art von Interfaces gibt es nicht erst seit Java 8 im JDK, sondern schon seit Langem und vielfach – wobei es früher für sie aber keine Bezeichnung gab. Vertreter der SAM-Typen und Functional Interfaces sind etwa `Runnable`, `Callable<V>`, `Comparator<T>`, `FileFilter`, `FilenameFilter`, `ActionListener`, `EventHandler` usw.

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}
```

```
@FunctionalInterface
public interface Comparator<T>
{
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

[1] Das gilt für sogenannte Checked Exceptions (vgl. Abschnitt 5.6.2). Solche vom Basistyp `RuntimeException` sind erlaubt.

Im Listing sehen wir die mit JDK 8 eingeführte Annotation `@FunctionalInterface` aus dem Package `java.lang`. Damit wird ein Interface explizit als Functional Interface gekennzeichnet. Die Angabe der Annotation ist optional: Jedes Interface mit genau nur einer abstrakten Methode (SAM-Typ) stellt auch ohne explizite Kennzeichnung ein Functional Interface dar. Wenn die Annotation angegeben wird, kann der Compiler eine Fehlermeldung produzieren, falls es (versehentlich) mehrere abstrakte Methoden gibt.

Tipp: Besondere Methoden in Functional Interfaces

Wenn wir im obigen Listing genauer hinsehen, könnten wir uns fragen, wieso denn `java.util.Comparator<T>` ein Functional Interface ist, wo es doch zwei Methoden enthält und keine davon abstrakt ist, oder? Als Besonderheit gilt in Functional Interfaces folgende Ausnahme für die Definition von abstrakten Methoden: Alle im Typ `Object` definierten Methoden können zusätzlich zu der abstrakten Methode in einem Functional Interface angegeben werden.

Verbleibt noch die Frage, warum wir in der Definition des Interface `Comparator<T>` keine abstrakte Methode sehen. Mit etwas Java-Basiswissen erinnern wir uns daran, dass alle Methoden in Interfaces automatisch `public` und `abstract` sind, auch wenn dies nicht explizit über Schlüsselwörter angegeben ist.

Basierend auf den Argumentationen ist die Methode `compare(T, T)` abstrakt und die Methode `equals(Object)` entstammt dem Basistyp `Object`. Sie darf damit zusätzlich im Interface zur abstrakten Methode aufgeführt werden.

Implementierung von Functional Interfaces

Ein SAM-Typ bzw. Functional Interface lässt sich durch eine anonyme innere Klasse implementieren. Seit JDK 8 sind Lambdas zu bevorzugen. Voraussetzung dafür ist, dass mit dem Lambda die abstrakte Methode des Functional Interface erfüllt werden kann, d. h., dass die Anzahl der Parameter übereinstimmt sowie deren Typen und der Rückgabetyp kompatibel sind. Betrachten wir zur Verdeutlichung zunächst ein allgemeines, etwas abstraktes Modell zur Transformation von bisherigen Realisierungen eines SAM-Typs mithilfe einer anonymen inneren Klasse in einen Lambda-Ausdruck:

```
// SAM-Typ als anonyme innere Klasse
new SAMTypeAnonymousClass()
{
    public void samTypeMethod(METHOD-PARAMETERS)
    {
        METHOD-BODY
    }
}

// SAM-Typ als Lambda
(METHOD-PARAMETERS) -> { METHOD-BODY }
```

Bei kurzen Methodenimplementierungen, wie sie für SAM-Typen häufig vorkommen, ist das Verhältnis von Nutzcode zu Boilerplate-Code (oder auch Noise genannt) bislang

recht schlecht. Wenn man für derartige Realisierungen Lambdas einsetzt, so kann man mit einer Zeile das ausdrücken, was sonst fünf oder mehr Zeilen benötigt. Nachfolgend wird dies für das Interface `Comparator<T>` verdeutlicht.

Beispiel: `Comparator<T>` Die Vorteile von Lambdas lassen sich für den Typ `Comparator<T>` gut demonstrieren. Wie später in Abschnitt 6.2.3 dargestellt, wird mit einem `Comparator<T>` ein Vergleich von zwei Instanzen vom Typ `T` realisiert, indem man die abstrakte Methode `int compare(T, T)` passend implementiert. Der Rückgabewert bestimmt die Reihenfolge der Werte. Wollte man zwei Strings nach deren Länge sortieren, so entsteht herkömmlicherweise einiges an Sourcecode:

```
Comparator<String> compareByLength = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
};
```

Wenn man Lambdas nutzt, lässt sich der Komparator knackig wie folgt schreiben:

```
Comparator<String> compareByLength = (final String str1, final String str2) ->
{
    return Integer.compare(str1.length(), str2.length());
};
```

Type Inference und Kurzformen der Syntax

Die Syntax von Lambdas besitzt einige Besonderheiten, um den Sourcecode prägnant formulieren zu können. Durch die sogenannte ***Type Inference*** ermittelt der Compiler die passenden Typen aus dem Einsatzkontext und es ist dadurch möglich, auf die Angabe der Typen für die Parameter im Sourcecode zu verzichten. Den vorherigen Komparator schreibt man ohne Typangabe bei den Parametern des Lambdas wie folgt:

```
Comparator<String> compareByLength = (str1, str2) ->
{
    return Integer.compare(str1.length(), str2.length());
};
```

Eine weitere Verkürzung in der Schreibweise eines Lambdas erreicht man durch folgende Regeln: Falls das auszuführende Stück Sourcecode ein Ausdruck ist, können die geschweiften Klammern um die Anweisungen entfallen. Ebenfalls kann dann das Schlüsselwort `return` weggelassen werden und der Rückgabewert entspricht dem Ergebnis des Ausdrucks. Außerdem gilt: Existiert lediglich ein Eingabeparameter, so sind die runden Klammern um den Parameter optional. Damit ergibt sich für die Ausdrücke

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
```

folgende Kurzschreibweise:

```
(x, y) -> x + y
x -> x * 2
```

Neben dem offensichtlichen Vorteil einer kompakten Schreibweise ist etwas anderes viel entscheidender: Lambdas können flexibler als streng typisierte Methoden genutzt werden. Für die gezeigten Berechnungen ist ein Einsatz überall dort möglich, wo für die Parameter die Operatoren + bzw. * definiert sind, also für die Typen `int`, `float`, `double` usw. Anders formuliert: ***Alles, was hergeleitet werden kann (und soll), darf in der Syntax weggelassen werden***. Als Beispiel betrachten wir folgende `ActionListener`-Implementierung, die schrittweise vereinfacht wird:

```
// Alter Stil
button.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(final ActionEvent e)
    {
        System.out.println("button clicked (old way)");
    }
});
```

Diese herkömmliche Realisierung mithilfe einer anonymen inneren Klasse lässt sich als Lambda und mit Type Inference deutlich kürzer schreiben:

```
// Lambda-Variante mit Type Inference
button.addActionListener((event) -> { System.out.println("button clicked!"); });
```

Nutzt man zusätzlich die Regeln zur Schreibweisenabkürzung, so entsteht Folgendes:

```
// Lambda-Kurzschreibweise
button.addActionListener(event -> System.out.println("button clicked!"));
```

Lambdas als Parameter und als Rückgabewerte

Wir haben mittlerweile ein wenig Gespür für Lambdas entwickelt und wissen, dass man Lambdas anstelle einer anonymen inneren Klasse zur Realisierung eines SAM-Typs nutzen kann. Ebenso lassen sich Lambdas als Methodenparameter und als Rückgabe einer Methode verwenden, um Aufrufe lesbar zu gestalten.

Als Beispiel betrachten wir das Sortieren einer Liste von Namen gemäß deren Länge. Das können wir mit folgenden zwei Varianten eines Lambdas für das Interface `Comparator<T>` schreiben:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");

   // Lambda als Methodenparameter
   Collections.sort(names, (str1, str2) -> Integer.compare(str1.length(),
                                                           str2.length()));
```

```
   // Alternative mit Lambda als Rückgabe einer Methode
   Collections.sort(names, compareByLength());
   System.out.println(names);
}

public static Comparator<String> compareByLength()
{
   return (str1, str2) -> Integer.compare(str1.length(), str2.length());
}
```

Listing 4.1 *Ausführbar als* **'LAMBDAASPARAMANDRETURNVALUEEXAMPLE'**

Das Programm LAMBDAASPARAMANDRETURNVALUEEXAMPLE sortiert erwartungskonform die Namen und gibt dann Folgendes aus:

```
[Max, Andy, Stefan, Michael]
```

4.1.3 Exceptions in Lambdas

Wir haben bisher in Bezug auf Lambdas noch nicht thematisiert, was passiert, wenn dort eine Checked Exception ausgelöst wird.[2] Oftmals »umrahmt« man potenziell fehlerträchtigen und Exception auslösenden Sourcecode mit einem `try-catch`-Block, wie Sie es bestimmt schon unzählige Male getan haben. Im Anschluss betrachten wir einige Varianten des Exception Handlings für Lambdas.

Versuch 1: Im Lambda auftretende Exception außerhalb behandeln

Nachfolgend sehen Sie die Definition eines `Comparator<File>` in Form eines Lambdas innerhalb der Methode `createFileComparator()`. Weil hier Dateisystemzugriffe erfolgen, können potenziell `IOException`s ausgelöst werden. Probieren wir eine erste Variante zum Exception Handling aus und umrahmen die Definition des `Comparator<File>` folgendermaßen:

```
private static Comparator<File> createFileComparator()
{
    try
    {
        final Comparator<File> fileComparator = (file1, file2) ->
        {
           final Path path1 = file1.toPath();

           // Achtung: Sourcecode löst potenziell unbehandelte IOExceptions aus
           // Compiler Error: Unhandled exception type IOException
           final BasicFileAttributes attrs = Files.readAttributes(path1,
                                                   BasicFileAttributes.class);

           // ...
        };
        return fileComparator;
    }
```

[2] Auf die unterschiedlichen Typen von Exceptions gehe ich in Abschnitt 5.6.2 genauer ein.

```
    //  Compiler Error: Unreachable catch block ...
    catch (final IOException ioe)
    {
        handleIOException(ioe);
    }
}
```

Der gezeigte Sourcecode führt zu folgenden Fehlern: »`Unhandled exception type IOException`« sowie »`Unreachable catch block for IOException. This exception is never thrown from the try statement body`«.

Überlegen wir kurz: Ein Lambda stellt ein ausführbares Stück Sourcecode dar und entspricht in etwa einer anonymen Methode. Vielfach wird ein Lambda nicht an der Stelle ausgeführt, an der er im Sourcecode definiert ist, sondern erst später, weil er durch das Programm gereicht wird, wie man es von Methodenparametern kennt. Mit diesem Wissen wird klar, dass ein Einfassen der Definitionsstelle eines Lambdas in einen `try-catch`-Block wirkungslos ist. Spannen wir wieder den Bogen zu Methodendefinitionen. Für diese behandeln wir ja auch keine Exceptions, sondern deklarieren potenziell ausgelöste Exceptions in der Methodensignatur über `throws`. Logischerweise verwenden wir einen `try-catch`-Block erst bei einem Aufruf einer solchen Methode.

Nachfolgend ist die beschriebene Abhilfe angedeutet, indem der Aufruf der Lambda definierenden Methode in einen `try-catch`-Block eingeschlossen wird:

```
try
{
    final List<File> files = collectFiles();
    final Comparator<File> fileComparator = createFileComparator();
    files.sort(fileComparator);
}
catch (final IOException ioe)
{
    handleIOException(ioe);
}
```

Versuch 2: Im Lambda die auftretende Exception behandeln

Basierend auf der obigen Herleitung wissen wir prinzipiell, wie man in Lambdas ausgelöste Exceptions korrekt behandelt: Man muss den entsprechenden Sourcecode-Abschnitt im Lambda mit einem `try-catch`-Block umschließen, etwa folgendermaßen:

```
final Comparator<File> fileComparator = (file1, file2) ->
{
    try
    {
        doSomePotentiallyDangerousIO();
    }
    catch (final IOException ioe)
    {
        handleIOException(ioe);
    }
};
```

Nun ist es aber eventuell nicht gewünscht oder nicht möglich, dass wir den Sourcecode des Lambdas, wie gezeigt, mit Exception Handling »verunstalten«. Wie kann man das Dilemma lösen? Fragen wir uns dazu, was die eigentliche Problematik ist: In Java muss man lediglich solche Exceptions behandeln, die als Checked Exceptions gelten, also den Basistyp `Exception` besitzen. Sogenannte Unchecked Exceptions mit der Basis `RuntimeException` betrifft der Zwang zur Behandlung nicht.

Manchmal weiß man sicher, dass eine in der Signatur einer aufgerufenen Methode deklarierte Checked Exception zur Laufzeit niemals ausgelöst wird. Das gilt beispielsweise ohne Thread-Kommunikation für Aufrufe von `Thread.sleep()`, die trotzdem immer die Behandlung einer `InterruptedException` erfordern. Für solche Fälle lässt sich die Compiler-Prüfung, ob eine explizite Behandlung von Exceptions erforderlich ist, verhindern. Dazu müssen wir alle möglicherweise auftretenden Checked Exceptions in Unchecked Exceptions, also solche mit dem Basistyp `RuntimeException`, umwandeln.

Verbesserte Variante mit Exception Wrapping

Kommen wir zu unserem Beispiel zurück. Es wäre wünschenswert, wenn wir uns im Applikationscode möglichst wenig mit den zuvor genannten Details auseinandersetzen müssten. Im Beispiel soll die Implementierung des Functional Interface `Comparator<T>` eine Checked Exception auslösen können. Dazu kann man folgenden Trick nutzen: Wir setzen das im Anschluss in Abschnitt 4.2.1 besprochene JDK-8-Sprachfeature der Defaultmethoden ein, um Functional Interfaces um Implementierungen zu erweitern. Dadurch können wir eine spezielle Variante der Methode des Interface realisieren, in dieser die Exceptions fangen und als `RuntimeException`s weiter propagieren. Weil die Implementierung des Lambdas potenziell eine Exception auslösen können muss, definieren wir ergänzend eine abstrakte Methode, die die erwartete Checked Exception in ihrer Signatur auflistet. Das klingt komplizierter, als es ist. Das zeigt die folgende Erweiterung `ComparatorThatThrows<T>` für das Interface `Comparator<T>`:

```
@FunctionalInterface
public interface ComparatorThatThrows<T> extends Comparator<T>
{
    @Override
    default int compare(final T t1, final T t2)
    {
        try
        {
            return compareThrows(t1, t2);
        }
        catch (final IOException e)
        {
            throw new RuntimeException(e);
        }
    }

    public int compareThrows(final T t1, final T t2) throws IOException;
}
```

Mit dieser Implementierung werden `IOException`s in den Typ `RuntimeException` konvertiert, wodurch wir die aufrufende Stelle wie folgt abwandeln können:

```
final ComparatorThatThrows<File> fileSizeComparator = (file1, file2) ->
{
    final Path path1 = file1.toPath();
    final BasicFileAttributes attrs = Files.readAttributes(path1,
                                            BasicFileAttributes.class);
    // ..
};
```

Zwar wird man so vom expliziten Behandeln einer Exception befreit, jedoch besitzt dieser Ansatz einige Nachteile:

1. Man muss Derartiges für all diejenigen Functional Interfaces duplizieren, für die man diese Exception-Wrapping-Funktionalität benötigt.
2. Es findet mitunter keine adäquate Fehlerbehandlung statt, weil die ausgelösten `RuntimeException`s potenziell unbehandelt an andere Programmteile weitergereicht werden.
3. Es werden eventuell Fehler so behandelt, dass ein Aufrufer dies nicht bemerkt oder nicht mehr steuernd eingreifen kann.

Aufgrund dieser Nachteile sollte man meiner Meinung nach in Lambdas möglichst auf Exceptions verzichten oder diese geeignet behandeln, wie dies zuvor per Aufruf von `handleIOException()` schon gezeigt wurde. Richtig schön ist keine der Varianten.

Hinweis: Exceptions und Lambdas

Bitte beachten Sie, dass es generell keine gute Idee darstellt, zu komplexe Berechnungen in Lambdas auszuführen, weil dies die Kombinierbarkeit und Ausführung mit dem Filter-Map-Reduce-Framework und insbesondere mit Streams erschwert.

Besonderheiten beim Exception-Wrapping Manchmal kann man für mehr Klarheit sorgen und eine leichte Erkennung eventuell zu behandelnder Exceptions bei Aufrufern dadurch erzielen, dass man statt der `RuntimeException` eine spezifischere `WrappedException` wie im folgenden Listing gezeigt selbst definiert. Dann können Aufrufer auf diesen Typ prüfen und speziell behandeln, indem die Exception zunächst entpackt wird und danach eine Prüfung auf die Fehlersituation erfolgt.

```
public class WrappedException extends RuntimeException
{
    public WrappedException(final Exception cause)
    {
        super(cause);
    }
}
```

Im Listing sehen wir eine Besonderheit: Während die Basisklasse `RuntimeException` im Konstruktor den allgemeinsten Typ `Throwable` entgegennimmt, erlauben wir hier

lediglich Checked Exceptions (Basistyp `Exception`) als Eingabe. Auf diese Weise werden wir dem gewählten Einsatzzweck gerecht.

Praktischer Nutzen von Exception-Wrapping Das Wrapping von Exceptions befreit zwar vom Behandeln, birgt damit jedoch auch die Gefahr, dass eine Exception unbehandelt den Programmablauf stört.

Weiterführende Infomationen

Weitere Details sowie eine Utility-Klasse, die Exception-Wrapper für gebräuchliche Functional Interfaces bereitstellt, finden Sie unter: `http://stack-overflow.com/questions/14039995/java-8-mandatory-checked-exceptions-handling-in-lambda-expressions-why-mandato`. Ergänzend ist der Artikel von Dr. Heinz Kabutz nützlich: `http://www.javaspecialists.eu/archive/Issue221.html`.

4.2 Syntaxerweiterungen in Interfaces

Beim Entwurf von Lambdas und deren Integration in das JDK stellte sich heraus, dass für eine sinnvolle Nutzbarkeit auch die bestehenden Klassen und Interfaces erweitert werden mussten. Bis zur Einführung von JDK 8 war es allerdings nicht möglich, ein Interface nach seiner Veröffentlichung zu verändern, ohne dass dies Auswirkungen bei allen einsetzenden Klassen gehabt hätte. Vielmehr führte die Erweiterung eines Interface bis inklusive JDK 7 immer zu einem Kompatibilitätsproblem: Wenn eine Methode einem Interface neu hinzugefügt wurde, musste diese in allen Klassen realisiert werden, die das Interface implementieren. Ansonsten kompilierten einige Klassen so lange nicht mehr, bis die Implementierung der neuen Methode bereitgestellt wurde.

4.2.1 Defaultmethoden

Um dieses Dilemma und vor allem Inkompatibilitäten bei Interface-Erweiterungen zu vermeiden, ist es seit Java 8 möglich, im Sourcecode eines Interface eine sogenannte Defaultimplementierung vorzugeben. Dazu nutzt man das Sprachfeature der ***Defaultmethoden***. Das sind ***spezielle Implementierungen von Methoden, die in Interfaces definiert werden können***. Um sie von den normalen abstrakten Methoden in Interfaces zu unterscheiden, werden Defaultmethoden mit dem Schlüsselwort `default` eingeleitet. Die Defaultmethoden sind eine wichtige Neuerung, um Lambdas für bestehende Funktionalitäten des JDKs gewinnbringend nutzen zu können.

Die Defaultmethoden `sort()` und `forEach()`

Lassen Sie uns zwei Erweiterungen näher betrachten. Die erste ist der Aufruf von `sort()` direkt auf der Instanz einer `List<E>`. Als Zweites betrachten wir eine Funktionalität, die eine Iteration über alle Elemente einer Collection und ein Bearbeiten jedes einzelnen Elements ermöglicht.

Beginnen wir mit der Erweiterung im Interface `List<E>` in Form der Definition von `sort(Comparator<? super E>)` wie folgt (gekürzt):

```
public interface List<E> extends Collection<E>
{
    default void sort(Comparator<? super E> c)
    {
        Collections.sort(this, c);
    }
}
```

Gebräuchlicher und allgemeiner als das Sortieren sind Iterationen über die Elemente einer Collection. Dazu dient die Defaultmethode `forEach(Consumer<? super T>)` im Interface `java.lang.Iterable<T>`. Dieses Interface bildet die Basis sowohl von `java.util.Collection<E>` als auch `List<E>`. Die Defaultmethode ist folgendermaßen implementiert (gekürzt):

```
public interface Iterable<T>
{
    default void forEach(Consumer<? super T> action)
    {
        for (T t : this)
        {
            action.accept(t);
        }
    }
}
```

Werfen wir einen kurzen Blick auf das in der Signatur genutzte Functional Interface `java.util.function.Consumer<T>`. Dort ist die abstrakte Methode `accept(T)` deklariert, deren Implementierung die für ein Element auszuführende Funktionalität festlegt. Darüber hinaus ermöglicht die Defaultmethode `andThen(Consumer<? super T>)` die Hintereinanderausführung mehrerer `Consumer<T>`-Instanzen:

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after)
    {
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

Hintergrundwissen: Functional Interfaces

Neben dem im Listing gezeigten Functional Interface `Consumer<T>` wurde eine Vielzahl solcher Interfaces in JDK 8 integriert. Im Package `java.util.function` finden sich um die 40 Functional Interfaces, oftmals mit sprechendem Namen, unter anderem Folgende:

- **`Consumer<T>`** – Beschreibt eine Aktion auf einem Element vom Typ `T`. Dazu ist eine Methode `void accept(T)` definiert.
- **`Predicate<T>`** – Definiert eine Methode `boolean test(T)`. Diese berechnet für eine Eingabe vom Typ `T` einen booleschen Rückgabewert (z. B. `olderThan()`). Damit lassen sich sehr gut Filterbedingungen ausdrücken.
- **`Function<T,R>`** – Definiert eine Abbildungsfunktion in Form der Methode `R apply(T)`. Damit lassen sich Transformationen beschreiben, als Beispiel etwa die Extraktion eines Attributs aus einem komplexeren Typ.

Zudem gibt es Varianten, die auf die Verarbeitung primitiver Typen ausgerichtet sind, im Speziellen für die Typen `int`, `long` und `double`. Die anderen primitiven Zahlentypen können mithilfe von Widening (vgl. Abschnitt 5.2.1), einer Typerweiterung etwa von `byte` auf `int`, auf die drei unterstützten Typen abgebildet werden.

Defaultmethoden und Lambdas im Einsatz

Als Motivation zur Verwendung von Lambdas habe ich an einem Beispiel gezeigt, wie man eine Liste von Namen ihrer Länge nach sortiert und ausgibt. Durch die Kombination von Defaultmethoden und Lambdas kann man das Ganze noch etwas kürzer und knackiger mit drei Zeilen realisieren – wobei hier sogar zusätzlich eine Transformation der Namen auf deren Längen stattfindet. Allerdings findet man den Schönheitsfehler eines Kommas nach dem letzten Element:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");

   names.sort((str1, str2) -> Integer.compare(str1.length(), str2.length()));
   names.forEach(it -> System.out.print(it.length() + ", "));
}
```

Listing 4.2 *Ausführbar als* **'DEFAULTMETHODANDLAMBDAEXAMPLE'**

Das Programm DEFAULTMETHODANDLAMBDAEXAMPLE gibt Folgendes aus:

```
3, 4, 6, 7,
```

Vielleicht fragen Sie sich, wofür `it` steht. Erinnern wir uns an die Transformation einer anonymen inneren Klasse in einen Lambda. Demzufolge entspricht der im Lambda genutzte Parameter `it` dem Parameter der abstrakten Methode `accept(T)` im Functional

Interface `Consumer<T>`. Der Name `it` ist eine gebräuchliche Abkürzung für Parameter beliebigen Typs bei Iterationen. Hier wäre `name` eine besser lesbare Alternative.

Vorgabe von Standardverhalten

Neben der Erweiterung eines Interface besteht ein Anwendungsfall von Defaultmethoden darin, für bereits existierende Methoden ein Standardverhalten vorgeben zu können.

Vor JDK 8 musste jeder Implementierer eines Interface für alle dort definierten Methoden eine eigene Realisierung bereitstellen. Das war störend, wenn für einzelne Methoden keine sinnvolle Implementierung vorgegeben werden konnte. Beispielsweise gilt dies oftmals für eigene Implementierungen des Interface `Iterator<E>` und dessen Methode `remove()`. Fast immer ist diese so realisiert, dass dort eine `UnsupportedOperationException` ausgelöst wird. Nun ist dieses Standardverhalten im JDK durch die Implementierung der Defaultmethode `remove()` umgesetzt. Dadurch kann man sich ganz auf die Implementierung der Iteration konzentrieren. Wenn man eine Spezialisierung von `Iterator<E>` erstellt, bei der keine eigene Aktion in `remove()` benötigt wird, dann ist die folgende Implementierung des JDKs passend:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();

    default void remove()
    {
        throw new UnsupportedOperationException("remove");
    }
    // ...
}
```

4.2.2 Statische Methoden in Interfaces

Interfaces können mit Java 8 nicht nur Defaultmethoden, sondern auch statische Methoden enthalten. Damit wird es möglich, Hilfsmethoden direkt in Interfaces bereitzustellen. Bis JDK 7 musste man dafür separate Utility-Klassen anbieten. Diese Konstellation kennt man sowohl aus dem JDK als auch aus eigenen Projekten. Beispiele aus dem JDK sind unter anderem die Utility-Klasse `Paths` zum Interface `Path` aus dem Package `java.nio.file` oder die Kombination von der Utility-Klasse `Executors` und dem Interface `Executor` aus dem Package `java.util.concurrent`.

Im JDK 8 finden sich viele Beispiele, in denen die Hilfsmethoden statt in einer eigenen Utility-Klasse direkt im Interface implementiert wurden. Dies gilt etwa für das Interface `Comparator<T>`. Nachfolgendes Listing zeigt nur auszugsweise einige statische Erzeugungsmethoden für spezielle Komparatoren, hier für eine inverse Sortierung (`reverseOrder()`), die natürliche Ordnung (`naturalOrder()`) sowie Sortierungen, die `null`-Werte vorne bzw. hinten einsortieren (`nullsFirst()` bzw. `nullsLast()`):

```
@FunctionalInterface
public interface Comparator<T>
{
    // ...

    public static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
    {
        return Collections.reverseOrder();
    }

    public static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
    {
        return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
    }

    public static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)
    {
        return new Comparators.NullComparator<>(true, comparator);
    }

    public static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)
    {
        return new Comparators.NullComparator<>(false, comparator);
    }

    // ...
}
```

Zwar lassen sich mithilfe statischer Methoden in Interfaces auf einfache Art gewisse Funktionalitäten bereitstellen, allerdings ***sollte man sich bewusst sein, dass man damit das Konzept des Interface zur Definition einer Schnittstelle immer mehr verwässert***. Aber nicht nur das! In diesem Beispiel erkennen wir, dass die eigentliche Schnittstellenbeschreibung nun Abhängigkeiten von diversen speziellen Klassen und Implementierungsdetails besitzt. Für diese Realisierung im JDK ist das wohl nicht so kritisch. ***Für eigene Interfaces sollte man sich jedoch sehr genau überlegen, ob man dort statische Methoden anbieten möchte und welche möglichen Auswirkungen und Abhängigkeiten sich dadurch ergeben***. Deklariert man dagegen lediglich Methoden in Interfaces, nimmt also eine reine Schnittstellenbeschreibung vor, so lassen sich diese Interfaces meistens viel leichter auch in andere Projekte integrieren, ohne eine (unerwartete) Menge von weiteren Abhängigkeiten anzuziehen.

4.3 Methodenreferenzen

Neben Lambdas kann der Einsatz der mit JDK 8 eingeführten Methodenreferenzen dazu beitragen, die Lesbarkeit des Sourcecodes zu erhöhen. Das Sprachfeature der Methodenreferenzen besitzt die Syntax `Klasse::Methodenname` und verweist auf ...

- eine Methode – `System.out::println`, `Person::getName`, ...
- einen Konstruktor – `ArrayList::new`, `Person[]::new`, ...

Das wirkt recht unspektakulär. Eine Methodenreferenz lässt sich aber zur Vereinfachung der Schreibweise anstelle eines Lambdas nutzen:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");

   names.forEach(it -> System.out.println(it)); // Lambda
   names.forEach(System.out::println); // Methodenreferenz
}
```

Wie man sieht, verbessert sich die Lesbarkeit. Allerdings könnte man sich noch folgende Fragen zu der Ersetzung stellen: Methoden erhalten oftmals Parameter – wie auch im Listing. Wie werden diese für Methodenreferenzen übergeben? Wie ist die Reihenfolge bei mehreren? Die Antwort darauf ist, dass diese Informationen vom Compiler ermittelt und automatisch beim jeweiligen Methodenaufruf übergeben werden.

Ergänzend zu dieser Ausführung möchte ich an ein paar Beispielen zeigen, wie sich Methodenreferenzen auf Lambdas bzw. andersherum abbilden lassen. Dabei gibt es vier verschiedene Varianten, die in Tabelle 4-1 dargestellt sind.

Tabelle 4-1 *Methodenreferenzen*

Referenz auf ...	Als Methodenreferenz	Als Lambda
Statische Methode	`String::valueOf`	`obj -> String.valueOf(obj)`

Instanzmethode eines Typs	`Object::toString`
	
	`String::compareTo`
		`obj -> obj.toString()`
		
		`(str1, str2) -> str1.compareTo(str2)`

Instanzmethode eines Objekts	`person::getName`	`() -> person.getName()`
Konstruktor	`ArrayList::new`	`() -> new ArrayList<>()`

4.4 Externe vs. interne Iteration

Mittlerweile haben wir Lambdas und Defaultmethoden ein paarmal in Aktion erlebt. Vor allem beim ***Durchlaufen einer Collection***, auch ***Iteration*** genannt, unterscheidet man seit Java 8 die externe und die interne Iteration. Von ***externer Iteration*** spricht man, wenn der Vorgang der Iteration vom Aufrufer kontrolliert wird. Dagegen wird bei der ***internen Iteration*** das Durchlaufen durch die Collection-Klasse gekapselt und dort intern realisiert. Implementierungsdetails bleiben so verborgen, allerdings sind auch die Möglichkeiten zur Einflussnahme durch den Aufrufer begrenzt. Betrachten wir nachfolgend einige Beispiele für die externe und interne Iteration.

Externe Iteration

Nehmen wir an, wir wollten alle Elemente einer Collection auf der Konsole ausgeben. Herkömmlicherweise könnte man dies wie folgt implementieren:

```
final List<String> names = Arrays.asList("Andi", "Mike", "Ralph", "Stefan" );

// Klassische Variante mit Iterator ...
final Iterator<String> it = names.iterator();
while (it.hasNext())
{
    System.out.println(it.next());
}

// ... oder alternativ mit indiziertem Zugriff
for (int i = 0; i < names.size(); i++)
{
    System.out.println(names.get(i));
}

// JDK-5-Schreibweise mit "for-each"
for (final String name : names)
{
    System.out.println(name);
}
```

Dieses Beispiel verdeutlicht die iterative und sequenzielle Abarbeitung sowohl für die Variante mit Iterator als auch für den danach gezeigten indizierten Zugriff. Die Variante mit der seit JDK 5 verfügbaren sogenannten for-each-Schleife[3] zeigt den sequenziellen Charakter weniger klar.[4] In allen drei Fällen spricht man von ***externer Iteration***, weil die ***Traversierung im Applikationscode programmiert*** wird.

[3] Leider wurde kein neues Schlüsselwort `foreach` eingeführt, sondern das Schlüsselwort `for` etwas missbraucht.

[4] Sie erleichtert lediglich die Schreibweise, stellt sogenannten syntaktischen Zucker dar und wird beim Kompilieren in eine externe Iteration mit Iterator umgewandelt.

Interne Iteration

Mit JDK 8 wurden die Klassen des Collections-Frameworks derart erweitert, dass sie verschiedene Verarbeitungsmethoden anbieten, die man bisher über `for`- oder `while`-Schleifen – wie oben im Listing – selbst programmieren musste. Neu ist, dass man die in der internen Iteration auszuführende Funktionalität übergibt. Dazu sind verschiedene Callback-Interfaces definiert. Seit JDK 8 bieten sich zu deren Implementierung sowohl Lambdas als auch Methodenreferenzen an:

```
// Interne Iteration in zwei Varianten
names.forEach(name -> System.out.println(name));
names.forEach(System.out::println);
```

Die im Listing gezeigte Form wird ***interne Iteration*** genannt, weil die Iteration nicht vom Entwickler selbst programmiert werden muss, sondern diese ***in der Collection realisiert*** wird. Man übergibt nur die auszuführende Aktion.

4.5 Wichtige Functional Interfaces für Collections

Neben der bereits eingesetzten Iteration mit `forEach()` existieren diverse weitere Beispiele für diese Art der internen Iteration im JDK. Einige wichtige finden wir im Collections-Framework, das wir im folgenden Kapitel betrachten. Nachfolgend lernen wir hier zunächst einige dafür grundlegende funktionale Interfaces kennen.

4.5.1 Das Interface `Predicate<T>`

Das funktionale Interface `java.util.function.Predicate<T>` erlaubt es, sogenannte ***Prädikate*** zu formulieren. Das sind boolesche Bedingungen, die durch Aufruf der im Interface definierten Methode `boolean test(T)` ausgewertet werden. Darüber hinaus sind ein paar andere Methoden im Interface `Predicate<T>` wie folgt definiert (gekürzt):

```
@FunctionalInterface
public interface Predicate<T>
{
   boolean test(T t);

   default Predicate<T> and(Predicate<? super T> other) { ... }
   default Predicate<T> negate() { ... }
   default Predicate<T> or(Predicate<? super T> other) { ... }
}
```

Auf die gezeigten Defaultmethoden gehe ich ein, nachdem ich einige Beispiele für Implementierungen des Interface selbst und der `test(T)`-Methode gegeben habe.

Im folgenden Listing sind mithilfe von Lambdas und Methodenreferenzen einfache Prüfungen auf den Wert `null`, einen Leerstring oder ein Mindestalter von 18 Jahren kurz und knackig formuliert:

```
public static void main(final String[] args)
{
   // Prädikate formulieren
    final Predicate<String> isNull = str -> str == null;
    final Predicate<String> isEmpty = String::isEmpty;
    final Predicate<Person> isAdult = person -> person.getAge() >= 18;

    final Person pia = new Person("Pia", LocalDate.of(1950, 1, 1), "London");
    System.out.println("isNull('''):     " + isNull.test(""));
    System.out.println("isEmpty('''):    " + isEmpty.test(""));
    System.out.println("isEmpty('Pia'): " + isEmpty.test(pia.getName()));
    System.out.println("isAdult(Pia):   " + isAdult.test(pia));
}
```

Listing 4.3 *Ausführbar als* '**FIRSTPREDICATESEXAMPLE**'

Das Programm FIRSTPREDICATESEXAMPLE gibt erwartungsgemäß Folgendes aus:

```
isNull('''):     false
isEmpty('''):    true
isEmpty('Pia'): false
isAdult(Pia):   true
```

Komplexere Bedingungen mit Prädikaten formulieren

Zwar kann man mit einfachen Prädikaten schon einige Anwendungsfälle abdecken. Zur Realisierung komplexerer Abfragen wird man jedoch verschiedene Bedingungen miteinander kombinieren wollen. Um boolesche Verknüpfungen auszuführen, bietet sich oftmals der Einsatz der folgenden drei Defaultmethoden an:

- `negate()` – Negiert die Bedingung.
- `and(Predicate<? super T>)` – Verknüpft die aktuelle Bedingung mit einer anderen Bedingung mit logischem UND.
- `or(Predicate<? super T>)` – Verknüpft die aktuelle Bedingung mit einer anderen Bedingung mit logischem ODER.

Mit diesem Wissen bauen wir unser Beispiel ein wenig aus und richten den Fokus dabei auf die Kombination von Prädikaten.

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Michael", LocalDate.of(1971, 2, 7), "Zürich"));
    persons.add(new Person("Barbara", LocalDate.of(1973, 3, 24), "Hamburg"));
    persons.add(new Person("Lili", LocalDate.of(1979, 7, 14), "Zürich"));
    persons.add(new Person("Tom", LocalDate.of(2017, 5, 5), "Kiel"));

    // Einfache Prädikate formulieren
    final Predicate<Person> isAdult = person -> person.getAge() >= 18;
    final Predicate<Person> livesInKiel =
                            person -> person.getCity().equals("Kiel");

    // Negation einzelner Prädikate
   final Predicate<Person> isYoung = isAdult.negate();
```

```
    // Kombination von Prädikaten mit AND
    final Predicate<Person> boysFromKiel = isYoung.and(livesInKiel);

    // Kombination von Prädikaten mit OR
    final Predicate<Person> boysFromKielOrLivingInHamburg =
                            boysFromKiel.or(person ->
                                            person.getCity().equals("Hamburg"));

    persons.removeIf(boysFromKielOrLivingInHamburg);
    System.out.println(persons);
}
```

Listing 4.4 *Ausführbar als* '**COMPLEXPREDICATESEXAMPLE**'

Im Listing sehen wir den Aufruf der später in Abschnitt 6.1.3 vorgestellten, aber intuitiv verständlichen Methode `removeIf(Predicate<E>)`, die Elemente aus einer Collection entfernt, die der übergebenen Bedingung entsprechen. Führt man das Programm COMPLEXPREDICATESEXAMPLE aus, so werden alle männlichen Personen unter 18 (Tom) sowie alle Personen, wohnhaft im Hamburg (Barbara), aus dem Ergebnis entfernt:

```
[Person [name=Michael, birthday=1971-02-07, city=Zürich],
 Person [name=Lili, birthday=1979-07-14, city=Zürich]]
```

Erweiterung im Interface `Predicate<T>` mit JDK 11

Zwar ließ sich mithilfe von `Predicate<T>` eine Negation per `negate()` ausdrücken, aber dies ist nur etwas umständlich und schwieriger lesbar:

```
final Predicate<String> isEmpty = String::isEmpty;
final Predicate<String> notEmptyJdk10 = isEmpty.negate();
```

Beispiel mit Java 11 Zur Vereinfachung hätte man sich eine Methode `not()` gewünscht. Java 11 bietet eine solche, die sich wie folgt einsetzen lässt:

```
final Predicate<String> notEmptyJdk11 = Predicate.not(String::isEmpty);
```

Das ist schon eine deutliche Verbesserung, aber es geht noch besser. Um die Lesbarkeit weiter zu erhöhen, empfiehlt sich ein statischer Import:

```
import static java.util.function.Predicate.not;
```

Dann kann man die Negation prägnanter schreiben:

```
final Predicate<String> notEmptyJdk11 = not(String::isEmpty);
```

In nutzenden Applikationen führt dies zu einer deutlichen Steigerung der Lesbarkeit. Man beachte außerdem, dass man im einführenden Beispiel eine künstliche, sogenannte Explaining Variable einführen musste, damit man eine Negation vornehmen konnte. Das ist mit Java 11 nicht mehr nötig.

4.5.2 Das Interface `UnaryOperator<T>`

Im funktionalen Interface `java.util.function.UnaryOperator<T>` selbst ist lediglich die statische Methode `identity()` definiert. Entscheidender ist, dass es über sein Basisinterface `Function<T,R>` die Methode `R apply(T)` anbietet. Diese bildet ein Element vom Typ `T` auf ein Element vom Typ `R` ab. Für den `UnaryOperator<T>` sind die Typen `T` und `R` gleich:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T>
{
    // Statische Methoden seit JDK 8 in Interfaces erlaubt
    static <T> UnaryOperator<T> identity()
    {
        return t -> t;
    }
}

@FunctionalInterface
public interface Function<T, R>
{
    R apply(T t);
    // ...
}
```

Das Ganze ist noch etwas abstrakt, daher schauen wir uns verschiedene Realisierungen von `UnaryOperator<String>` an: Zunächst verändern wir alle mit »`M`« startenden Namen (`markTextWithM`). Praxisrelevanter sind das Trimmen (`trimmer`) und die Abbildung von `null`-Werten auf gewünschte Defaultwerte (`mapNullToEmpty`). Die korrespondierenden `UnaryOperator<String>`s realisieren wir wie folgt:

```
public static void main(final String[] args)
{
   // Mark
   final UnaryOperator<String> markTextWithM = str -> str.startsWith("M") ?
                                        ">>" + str.toUpperCase() + "<<" : str;

   printResult("Mark 1", "unchanged", markTextWithM);
   printResult("Mark 2", "Michael", markTextWithM);

   // Trim
   final UnaryOperator<String> trimmer = String::trim;
   printResult("Trim 1", "no_trim", trimmer);
   printResult("Trim 2", "  trim me   ", trimmer);

   // Map
   final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" : str;
   printResult("Map same", "same", mapNullToEmpty);
   printResult("Map null", null, mapNullToEmpty);
}

private static void printResult(final String text, final String value,
                                final UnaryOperator<String> op)
{
   System.out.println(text + ": '" + value + "' -> '" + op.apply(value) + "'");
}
```

Listing 4.5 *Ausführbar als* **'UNARYOPERATOREXAMPLE'**

Das Programm UNARYOPERATOREXAMPLE produziert folgende Ausgaben:

```
Mark 1: 'unchanged' -> 'unchanged'
Mark 2: 'Michael' -> '>>MICHAEL<<'
Trim 1: 'no_trim' -> 'no_trim'
Trim 2: '  trim me   ' -> 'trim me'
Map same: 'same' -> 'same'
Map null: 'null' -> ''
```

4.6 Praxiswissen: Definition von Lambdas

Wir wissen zwar schon das eine oder andere über Lambdas, aber es gibt noch ein Detail zum Zugriff auf Variablen kennenzulernen.

Mitunter muss man Lambdas definieren, die auf Variablen zugreifen, etwa ein `Predicate<T>` zur Bereichsprüfung wie folgt:

```
final Predicate<Integer> closedInterval =
                         value -> lowerBound <= value && value <= upperBound;
```

Man kann auf derartige Variablen nur dann zugreifen, wenn diese `final` oder »effectively final« sind. Darunter versteht man, dass die Variablen nicht mehr explizit `final` deklariert werden müssen, sondern es genügt, wenn diese ihren Wert zur Programmlaufzeit nicht ändern. Dieser Sachverhalt wird vom Compiler geprüft und Verstöße werden als Fehler angemahnt.

Zwar lässt sich das ohne Mühe machen, aber das löst das Problem nur für einfache Anwendungsfälle. In der Praxis benötigt man jedoch oftmals mehr Flexibilität. Dazu gibt es eine trickreiche Abhilfe, die so einfach wie elegant ist, nämlich die Definition von Hilfsmethoden:

```
static Predicate<Integer> closedInterval(final int lowerBound,
                                         final int upperBound)
{
    if (upperBound < lowerBound)
        throw new IllegalArgumentException("lowerBound must be <= upperBound");

    return value -> lowerBound <= value && value <= upperBound;
}
```

5 Java-Grundlagen

Dieses Kapitel stellt verschiedene grundlegende Java-Sprachelemente vor, die wir in den folgenden Kapiteln immer wieder verwenden. Nützliches und Wissenswertes rund um die Klasse `java.lang.Object` – die Basis aller anderen Objekte – sowie ihre wichtigsten Methoden sind Thema von Abschnitt 5.1. In Abschnitt 5.2 folgt eine Beschreibung der primitiven Datentypen und der korrespondierenden Wrapper-Klassen. Danach beschäftigen wir uns in Abschnitt 5.3 mit der Verarbeitung textueller Informationen. Anschließend stellt Abschnitt 5.4 Wissenswertes zu inneren Interfaces und inneren Klassen vor. Das Thema Ein- und Ausgabe und das Konzept von Ein- und Ausgabestreams wird in Abschnitt 5.5 besprochen. Schließlich beschreibt Abschnitt 5.6 verschiedene Varianten zur Behandlung von Fehlern. Dort werden sowohl die Stärken als auch die Tücken beim Einsatz von Exceptions und Assertions dargestellt.

5.1 Die Klasse `Object`

Die Klasse `java.lang.Object` ist die Basisklasse aller in Java existierenden Klassen. Jede Referenzvariable besitzt zumindest immer den Typ `Object`.[1] Das ist praktisch, da diese Basisklasse bereits einige Methoden und damit elementare Verhaltensweisen bereitstellt. In der folgenden Aufzählung gebe ich einen kurzen einführenden Überblick über die wichtigsten Methoden und ihre Intentionen:

- `Class<?> getClass()` – Mithilfe dieser Methode kann der Laufzeittyp einer Klasse abgefragt werden. Diese Methode ist `final` und kann daher nicht überschrieben werden. Sie ist wichtig für den Einsatz von ***Reflection***. Damit wird es zur Laufzeit möglich, Klassenbestandteile zu ermitteln, etwa Methoden, Attribute, Oberklassen, implementierte Interfaces usw. Details beschreibt Abschnitt 12.1.
- `String toString()` – Diese Methode dient dazu, eine textuelle Repräsentation für ein Objekt zu erzeugen. Um aussagekräftige Informationen über ein Objekt zur Verfügung zu stellen, ist es ratsam, die Methode `toString()` zu überschreiben. In Abschnitt 5.1.1 werden wir dies genauer betrachten.

[1] Seit Java 8 gilt das nicht mehr uneingeschränkt. Die in Kapitel 4 beschriebenen Lambdas lassen sich nicht dem Typ `Object` zuweisen.

- `boolean equals(Object)` – Diese Methode realisiert einen Vergleich eines Objekts mit einem anderen. Die Defaultimplementierung führt lediglich einen Referenzvergleich durch. ***Deshalb muss diese Methode in der Regel für eigene Klassen überschrieben werden, um einen sinnvollen Vergleich zweier Instanzen zu realisieren***. Abschnitt 5.1.2 beschreibt dies im Detail.
- `int hashCode()` – Die Methode `hashCode()` wird immer dann benötigt, wenn Objekte in hashbasierten Containern verarbeitet werden sollen. Sie bildet den Objektzustand (oder Teile davon) auf eine Zahl ab. ***Wenn man `equals(Object)` überschreibt, ist es dringend anzuraten, auch immer `hashCode()` passend dazu zu überschreiben***.[2] Abschnitt 6.1.7 geht auf die Thematik genauer ein.
- `void wait()`, `void notify()` und `void notifyAll()` – Diese Methoden dienen der Thread-Steuerung und sind alle `final` definiert, können also nicht überschrieben werden. Durch Aufruf der Methode `wait()` eines Objekts kann ein Thread darauf warten, dass eine Bedingung eintritt. Er wird inaktiv, bis diese Bedingung von anderen Threads hergestellt und dies durch Aufruf der Methode `notify()` bzw. `notifyAll()` des Objekts an den wartenden Thread kommuniziert wird. In Abschnitt 10.3.2 werden die genannten Methoden im Detail vorgestellt.
- `void finalize()` – Diese Methode ist für Aufräumarbeiten gedacht und wird vom Garbage Collector aufgerufen, bevor dieser den Speicher eines Objekts freigeben möchte, sobald es nicht mehr benötigt (d. h. durch andere Objekte referenziert) wird. Die Defaultimplementierung ist leer und kann überschrieben werden. Da nicht garantiert ist, dass ein Objekt tatsächlich auch von der Garbage Collection behandelt wird, stellt ***`finalize()` nur eine letzte, wenn auch unsichere Lösung dar, Aufräumarbeiten auszuführen oder Ressourcen freizugeben.***

Im Laufe dieses Kapitels werden Tipps und Hinweise zur Implementierung für die Methoden `toString()` und `equals(Object)` vorgestellt. Das geschieht exemplarisch anhand einer Klasse `Person`. Im Anschluss ist für diese eine Minimalimplementierung angegeben, die sukzessive ausgebaut wird. In Kapitel 6 ergänzen wir dann die Methode `hashCode()`.

Die in der obigen Aufzählung letztgenannten Methoden sind bei der Anwendungsentwicklung in der Regel nicht ganz so häufig von Interesse – mit ihrer Bedeutung und groben Funktionsweise sollte aber jeder Java-Entwickler vertraut sein.

[2] Außer die Klasse wird sicher niemals in Hashcontainern verwaltet.

Beispielklasse Person

Die nachfolgend gezeigte Klasse Person besitzt die drei Attribute name, birthday und city und ist als unveränderliche Klasse folgendermaßen realisiert:

```
public final class Person
{
   private final String name;
   private final LocalDate birthday;
   private final String city;

   public Person(final String name, final LocalDate birthday, final String city)
   {
      Objects.requireNonNull(name, "parameter 'name' must not be null");
      Objects.requireNonNull(birthday, "parameter 'birthday' must not be null");
      Objects.requireNonNull(city, "parameter 'city' must not be null");

      this.name = name;
      this.birthday = birthday;
      this.city = city;
   }

   public final String getName()          { return name; }
   public final LocalDate getBirthday()   { return birthday; }
   public final String getCity()          { return city; }
}
```

Bei dieser Realisierung ist insbesondere der Einsatz der Klasse java.util.Objects erwähnenswert. Diese Utility-Klasse wurde mit JDK 7 eingeführt und bietet einige recht praktische Hilfsmethoden. Im Listing werden unerlaubte Eingaben von null mithilfe der generischen Methode requireNonNull(T, String) behandelt.

5.1.1 Die Methode toString()

Wie erwähnt, dient die Methode toString() dazu, eine möglichst aussagekräftige und lesbare textuelle Repräsentation eines Objekts zu erzeugen. Allerdings gibt die Defaultimplementierung lediglich den Klassennamen gefolgt von einer hexadezimalen Darstellung des über die Methode hashCode() berechneten Werts zurück (standardmäßig die Referenz des Objekts in einen int konvertiert). Das lernen wir an einem Beispiel kennen, das dabei hilft, die von der Defaultimplementierung erzeugten Ausgaben für Objekte der Klasse Person und für ein Object-Array zu analysieren:

```
public static void main(final String[] args)
{
    final Object obj = new Object();
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Bremen");
    final Object[] objectArray = new Object[] { obj, mike };

    System.out.println("Object: " + obj);
    System.out.println("Person: " + mike);
    System.out.println("Object[]: " + objectArray);
}
```

Listing 5.1 *Ausführbar als* 'TOSTRINGEXAMPLE'

Startet man das Programm TOSTRINGEXAMPLE, so wird etwa Folgendes ausgegeben:

```
Object: java.lang.Object@311d617d
Person: ch05_javagrundlagen.Person@7eab70f
Object[]: [Ljava.lang.Object;@7852e922
```

Diese Informationen sind wenig nützlich. Wie bereits gesagt, wird lediglich der über `getClass()` ermittelte Klassenname und der mit der Methode `hashCode()` berechnete Hashwert als hexadezimale Zahl ausgegeben.

Ein Überschreiben der Methode `toString()` ist also immer dann sinnvoll, wenn man eine menschenlesbare Form der Ausgabe erreichen möchte. Wir sehen folgende Verbesserungsmöglichkeiten:

1. Die Repräsentation der Klasse `Person` ist nichtssagend und wenig hilfreich. Besser wäre beispielsweise die Ausgabe der Wertebelegungen der Attribute.
2. Auch die Ausgabe des Arrays entspricht nicht dem, was wir erwarten. Statt der kryptischen Notation `[Ljava.lang.Object;@7852e922` würde man sicher eine kommaseparierte Liste der Stringrepräsentationen enthaltener Objekte vorziehen.

Bevor wir die beiden Probleme behandeln, wollen wir zunächst einen Blick auf die Mechanismen bei der Ausgabe und Umwandlung von Werten mit `toString()` werfen.

Mechanismen bei Ausgaben über `System.out.println()`

In diesem Beispiel sehen wir keinen direkten Aufruf der Methode `toString()`. Tatsächlich wird diese Methode immer dann implizit, d. h. also »unsichtbar«, aufgerufen, wenn ein Objekt in einen String umgewandelt werden muss – hier zur Ausgabe über `System.out.println()`. Diese Umwandlung kann auch explizit durch einen direkten Aufruf von `toString()` für eine Objektreferenz durchgeführt werden. Dann muss allerdings für `null`-Referenzen eine Sonderbehandlung erfolgen, da es ansonsten zu `java.lang.NullPointerException`s kommt. Die Ausgabe von `null`-Werten mit `System.out.println()` wie in der folgenden Zeile

```
System.out.println("println is " + null + " safe");
```

führt dagegen nicht zu Exceptions, sondern zum Text `"null"`. Wie kommt das? Zur Umwandlung von Werten primitiver Datentypen und von Objektreferenzen in eine Stringrepräsentation sind im JDK in der Klasse `String` diverse überladene statische `valueOf()`-Methoden definiert. Diese werden implizit beim Aufruf von `System.out.println()` genutzt. Ein Blick auf die Implementierung der Methode `valueOf(Object)` erklärt das beschriebene Verhalten für Aufrufe mit `null`-Werten:

```
public static String valueOf(Object obj)
{
    return (obj == null) ? "null" : obj.toString();
}
```

Verbesserungen der toString()-Methode für die Klasse Person

Zur Verbesserung der Ausgabe überschreiben wir die toString()-Methode der Klasse Person. Die Realisierung sollte den Klassennamen, die Namen der Attribute sowie deren Werten enthalten. Im einfachsten Fall kann man sich eine Methode von Eclipse durch Aufruf des Menüs SOURCE –> GENERATE TOSTRING()… erzeugen lassen:

```
@Override
public String toString()
{
    return "Person [name=" + name + ", birthday=" + birthday +
           ", city=" + city + "]";
}
```

Das ist eine gute Basis, die sich noch verbessern lässt, indem etwa die Werte der Attribute durch Methodenaufrufe ermittelt werden. Zudem bietet es sich für textuelle Werte mitunter an, diese in Hochkommata (oder spitze Klammern bzw. Anführungszeichen) einzuschließen, wie es im folgenden Listing gezeigt ist. Auf diese Weise lassen sich Fehler erkennen, die durch führende oder nachfolgende Leerzeichen verursacht sind.

```
@Override
public String toString()
{
    return "Person: " +
           "Name='" + getName() + "' " +          // Variante 1
           "Birthday=<" + getBirthday() + "> " + // Variante 2
           "City=\"" + getCity() + "\"";          // Variante 3
}
```

Listing 5.2 *Ausführbar als* **'PERSONTOSTRINGEXAMPLE'**

Der Aufruf des Programms PERSONTOSTRINGEXAMPLE zeigt alle drei Varianten der Stringausgabe und die Ausgabe auf der Konsole ist bereits recht gut lesbar:

```
Person: Name='Mike' Birthday=<1971-02-07> City="Bremen"
```

Tipp: Die Annotation @Override

Die Annotation @Override dient dazu, Methoden zu kennzeichnen, die korrespondierende Methoden einer Basisklasse überschreiben oder eines Basisinterface implementieren (sollen). Dies wird dann durch den Compiler geprüft und sichergestellt. Die Angabe der Annotation hilft, sich vor Flüchtigkeitsfehlern zu schützen, etwa dem versehentlichen Überladen durch Unterschiede in der Methodensignatur. Auch Tippfehler im Methodennamen lassen sich so erkennen. Zudem kann durch die Angabe der Annotation @Override vom Compiler eine Warnmeldung erzeugt werden, wenn in einer Basisklasse Methoden geändert werden. Abgeleitete Klassen würden diesen Sachverhalt ohne Annotation nicht mitbekommen.

Anmerkungen zur Performance Mitunter sieht man in `toString()`-Methoden den Einsatz der Klassen `java.lang.StringBuffer` bzw. `java.lang.StringBuilder` – oftmals mit dem Ziel, die Aufbereitung der Ausgabe bezüglich der Ausführungsgeschwindigkeit zu optimieren. Die gerade gezeigte `toString()`-Methode der Klasse `Person` könnte man unter Nutzung der Methode `append(String)` der Klasse `StringBuilder` folgendermaßen umbauen:

```
@Override
public String toString()
{
    final StringBuilder sb = new StringBuilder();
    sb.append(getClass().getSimpleName()).append(": ");
    sb.append("Name='").append(getName()).append("' ");
    sb.append("Birthday='").append(getBirthday()).append("' ");
    sb.append("City='").append(getCity()).append("' ");
    return sb.toString();
}
```

Anhand des Listings wird unmittelbar die schlechtere Lesbarkeit und umständlichere Schreibweise deutlich. Zudem lässt sich so nur dann ein geringer Performance-Gewinn erzielen, wenn obige Methode extrem häufig aufgerufen wird. ***Sofern also nicht triftige Gründe dagegen sprechen, sollte man immer so lesbar wie möglich programmieren, um die Verständlichkeit und Wartbarkeit zu erhöhen***. Für das Beispiel heißt das, dass die Konkatenation einzelner Stringobjekte mit dem Operator '+' geschehen sollte. Nur für den Fall, dass die Aufbereitung der Ausgaben sehr umfangreich ist und zudem innerhalb von Schleifen erfolgt, kann sich der Einsatz eines `StringBuilder`s und dessen `append(String)`-Methode statt des Operators '+' lohnen. Die Gründe werden in Abschnitt 5.3 und Kapitel 22 im Detail erklärt.

Verbesserungen der `toString()`-Ausgabe für Arrays

Eine menschenlesbare Ausgabe von Arrays ist relativ leicht zu erreichen, indem man statische Hilfsmethoden aus der Utility-Klasse `java.util.Arrays` nutzt. Mit der generischen Methode `asList(T...)` erhält man eine Referenz auf das Interface `java.util.List<E>`. In dessen Realisierungen liefert die `toString()`-Methode die gewünschte kommaseparierte Ausgabe. Als Alternative bietet die Klasse `Arrays` seit Java 5 diverse überladene `toString()`-Methoden, unter anderem die hier genutzte Methode `toString(Object[])`:

```
public static void main(final String[] args)
{
    final Person tom = new Person("Tom", LocalDate.now(), "Hamburg");
    final Person jerry = new Person("Jerry", LocalDate.now(), "Kiel");
    final Object[] persons = new Object[] { tom, jerry };

    System.out.println("Object[]: " + Arrays.asList(persons));   // JDK 1.4
    System.out.println("Object[]: " + Arrays.toString(persons)); // JDK 5.0
}
```

Listing 5.3 *Ausführbar als* '**ARRAYSTOSTRINGEXAMPLE**'

Startet man das Programm ARRAYSTOSTRINGEXAMPLE, so erhält man das gewünschte Resultat (hier gekürzt und leicht umformatiert):

```
Object[]:
[Person: Name='Tom' Birthday='2020-01-26' City='Hamburg',
 Person: Name='Jerry' Birthday='2020-01-26' City='Kiel']
```

5.1.2 Die Methode `equals()`

Dieser Abschnitt stellt die Implementierung der Methode `equals(Object)` ausführlich dar, weil diese Methode eine zentrale Rolle bei der Verwaltung von Objekten in Containerklassen spielt. Wie bereits erwähnt, ist jedes Objekt durch seinen ***Zustand*** (Belegung der Attribute), sein ***Verhalten*** (Methoden) und seine ***Identität*** (Referenz) definiert. Zum Vergleich von Objekten gibt es unter anderem folgende Möglichkeiten:

1. **Operator '==' –** Mit dem Operator '==' werden Objektreferenzen verglichen. Somit wird überprüft, ob es sich um *dieselben* Objekte handelt.
2. **Aufruf der Methode `equals()` –** Wenn zwei Objekte bezüglich ihres Zustands (d. h. der Wertebelegung der Attribute) verglichen werden sollen, dann nutzt man `equals(Object)`. Dazu ist die Methode `equals(Object)` in eigenen Klassen zu überschreiben, weil deren Defaultimplementierung aus der Klasse `Object` lediglich Referenzen mit dem Operator '==' vergleicht. In eigenen Realisierungen muss derjenige Teil des Objektzustands verglichen werden, der für die ***semantische Gleichheit***, also die inhaltliche Gleichheit, relevant ist.

Die Methode `equals(Object)` kann zwar explizit zum Vergleich aufgerufen werden, in der Regel geschieht dies aber implizit und automatisch bei verschiedenen Aktionen.

Die Rolle von `equals()` beim Suchen

Um erste Erkenntnisse zur Implementierung von `equals(Object)` zu gewinnen, betrachten wir ein einfaches Beispiel einer Klasse `Spielkarte`. Die Implementierung nutzt einen Aufzählungstyp `Farbe` und einen `int` als Wert der Karte. Im Konstruktor wird abgesichert, dass nur gültige Objekte vom Typ `Farbe` übergeben werden:

```
public final class Spielkarte
{
    enum Farbe { KARO, HERZ, PIK, KREUZ }

    private final Farbe farbe;
    private final int   wert;

    public Spielkarte(final Farbe farbe, final int wert)
    {
        this.farbe = Objects.requireNonNull(farbe, "farbe must not be null");
        this.wert = wert;
    }
}
```

Nachfolgend wollen wir einige Objekte vom Typ `Spielkarte` in einer `ArrayList<E>` speichern, wobei hier als generischer Typ `Spielkarte` Verwendung findet. Das folgende Listing zeigt das Erzeugen einiger Objekte vom Typ `Spielkarte` und deren Speicherung in einer `ArrayList<Spielkarte>`.[3] Nach dem Hinzufügen von drei Spielkarten prüfen wir mit der Methode `contains(Object)`, ob die »Pik 8« in der Datenstruktur enthalten ist, und speichern den Rückgabewert in der Variablen `gefunden`:

```
public static void main(final String[] args)
{
    // JDK 7: Diamond Operator: ArrayList<> statt ArrayList<Spielkarte>
    final List<Spielkarte> spielkarten = new ArrayList<>();

    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));
    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK, 8));
    System.out.println("gefunden=" + gefunden);
}
```

Listing 5.4 *Ausführbar als* **'SPIELKARTEEXAMPLE'**

Gefunden oder doch nicht? Auf den ersten Blick lautet die Antwort natürlich gefunden, weil ja eine »Pik 8« in die Datenstruktur eingefügt wurde. Starten wir das angegebene Programm SPIELKARTEEXAMPLE und prüfen den Wert der Variablen `gefunden`, so erleben wir – je nach Java-Kenntnisstand – aber eine Überraschung, denn `gefunden` hat den Wert `false`! Die gesuchte »Pik 8« ist laut Rückgabe von `contains(Object)` nicht in der Datenstruktur enthalten. Wie kann das sein?

In der Implementierung von `contains(Object)` werden alle in der Collection gespeicherten Elemente durchlaufen und auf Gleichheit mit dem übergebenen Objekt durch Aufruf von `equals(Object)` geprüft. Die Klasse `Spielkarte` überschreibt diese Methode jedoch nicht. Es wird daher die Methode der Basisklasse `Object` aufgerufen, die nur auf Referenzgleichheit prüft und folgendermaßen implementiert ist:

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

Mit diesem Wissen erklärt sich das Ergebnis der oben gezeigten Suche. Dort wird die Methode `contains(Object)` mit einem neu erzeugten Objekt vom Typ `Spielkarte` aufgerufen. Dieses ist (natürlich) nicht in der Datenstruktur vorhanden.

[3] Details zu Listen und anderen Datenstrukturen beschreibt Kapitel 6 und geht dabei ausführlich auf das Collections-Framework ein.

Der `equals()`-Kontrakt

Vielfach benötigt man eine Prüfung auf inhaltliche Gleichheit. Dazu muss in eigenen Klassen die Methode `equals(Object)` passend überschrieben und dabei deren Kontrakt eingehalten werden. In der JLS [26] ist dazu folgende Signatur festgelegt:

```
public boolean equals(Object obj)
```

Diese Methode muss eine Äquivalenzrelation mit folgenden Eigenschaften realisieren:

- **Null-Akzeptanz** – Für jede Referenz `x` ungleich `null` liefert `x.equals(null)` den Wert `false`.
- **Reflexivität** – Für jede Referenz `x`, die nicht `null` ist, muss `x.equals(x)` den Wert `true` liefern.
- **Symmetrie** – Für alle Referenzen `x` und `y` darf `x.equals(y)` nur den Wert `true` ergeben, wenn `y.equals(x)` dies auch tut.
- **Transitivität** – Für alle Referenzen `x`, `y` und `z` gilt: Sofern `x.equals(y)` und `y.equals(z)` den Wert `true` ergeben, dann muss dies auch `x.equals(z)` tun.
- **Konsistenz** – Für alle Referenzen `x` und `y`, die nicht `null` sind, müssen mehrmalige Aufrufe von `x.equals(y)` konsistent den Wert `true` bzw. `false` liefern.

Mögen diese Forderungen ein wenig kompliziert klingen, so ist eine Umsetzung in der Praxis doch relativ einfach nach folgendem zweistufigen Vorgehen möglich:

1. **Prüfungen** – Vor dem inhaltlichen Vergleich sichern wir ab, dass
   a) keine `null`-Referenzen,
   b) keine identischen Objekte und
   c) nur Objekte des gewünschten Typs verglichen werden.
2. **Objektvergleich** – Anschließend werden diejenigen Attributwerte verglichen, die für die Aussage »Gleichheit« relevant sind.

Prüfungen Die Implementierung der Methode `equals(Object)` in eigenen Klassen sollte mit einer Prüfung des Übergabeparameters auf `null` beginnen (Punkt a). Für eine bessere Performance wird danach auf Identität (Punkt b) geprüft, weil man sich bei Identität alle weiteren, gegebenenfalls aufwendigen Prüfungen sparen kann. Um nicht Äpfel mit Birnen zu vergleichen, erfolgt eine Typprüfung (Punkt c) vor dem eigentlichen Vergleich der Attribute.

Zu dieser Typprüfung findet man kontroverse Meinungen, ob diese mit `getClass()` oder `instanceof` erfolgen sollte. Eine Prüfung mit `instanceof` ist zwar in vielen Fällen ausreichend, beim Einsatz von Vererbung kann es jedoch zu Fehlern kommen: Wenn man sich unsicher ist, sollte man bevorzugt `getClass()` verwenden, da ansonsten schnell die im Kontrakt geforderte Symmetrie verletzt wird. Ausnahmen bilden die Fälle, in denen man explizit Instanzen von Subklassen und Basisklassen

als »gleich« betrachten möchte, falls diese in der Wertebelegung ihrer Attribute übereinstimmen. Man sollte sich bewusst sein, dass es im Zusammenspiel mit Vererbung einiges bei der Typprüfung zu bedenken gibt. Das behandle ich später genauer.

Für die finale, nicht abgeleitete Klasse `Spielkarte` fällt die Wahl leicht: Wir verwenden hier `getClass()`. Aus den Punkten a) bis c) ergeben sich folgende erste Zeilen der Realisierung von `equals(Object)`:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit sicherstellen
        return false;

    // ...
```

Tipp: `equals()` mit `instanceof` vs. `getClass()`

Wie erwähnt, gibt es zur Typprüfung in `equals(Object)` kontroverse Meinungen. Eine Diskussion mit Joshua Bloch finden Sie online unter `http://www.artima.com/intv/bloch17.html`.

Eine Typprüfung mit `getClass()` testet, ob zwei Klassen exakt den gleichen Typ besitzen. Über `instanceof` werden auch alle Subklassen beim Vergleich auf eine Basisklasse akzeptiert. Es wird also eine Subtypbeziehung überprüft. Der Einsatz von `instanceof` ist allerdings nur dann erlaubt, wenn die geprüften Klassen eine gemeinsame Typhierarchie, d. h. eine gemeinsame Basisklasse ungleich `Object`, besitzen. Würde man eine beliebige Referenzvariable vom Typ `JButton` auf Typkonformität mit der zuvor vorgestellten Klasse `Person` prüfen wollen, so führt die Anweisung `okButton instanceof Person` zu einem Kompilierfehler. ***Unterschiedliche Vererbungshierarchien lassen sich mit `instanceof` nicht vergleichen. Praktischerweise ist `instanceof` aber `null`-sicher, d. h., `null instanceof XYZ` liefert immer `false` und keine `NullPointerException`.***

Objektvergleich Wurden die initialen Prüfungen bestanden, so kann nun das übergebene Objekt sicher auf den entsprechenden Typ gecastet werden. Die relevanten Attribute des Objekts werden (z. B. in der Reihenfolge ihrer Definition oder des ersten vermuteten Unterschieds) auf Gleichheit geprüft. Dabei gelten folgende Regeln:

1. Vergleiche alle primitiven Ganzzahltypen per Operator '==‘. Für Gleitkommazahlen sind derartige Vergleiche fehleranfällig und fragil. Aufgrund der systemimmanenten Ungenauigkeit bei Berechnungen mit Gleitkommazahlen müssen wir bei deren Vergleich besondere Vorsicht walten lassen bzw. sie in `equals(Object)` möglichst vermeiden. Nur in Ausnahmefällen kann man eine spezielle Gleichheitsprüfung vornehmen. Beachten Sie dazu bitte die Ausführungen in Abschnitt 5.2 und den folgenden Hinweiskasten.
2. Vergleiche alle Objekttypen mit deren `equals(Object)`-Methode. Für optionale Attribute, bei denen `null` ein gültiger Wert ist, muss eine spezielle Behandlung erfolgen. In den folgenden Abschnitten wird dies genauer betrachtet.

Diese Erkenntnisse nutzen wir, um die `equals(Object)`-Methode für die Klasse `Spielkarte` zu vervollständigen. Dadurch wird auch die gesuchte »Pik 8« korrekt gefunden – prüfen Sie es mit dem Programm SPIELKARTEWITHEQUALSEXAMPLE.

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit sicherstellen
        return false;

    // Enum mit equals, int mit Wertevergleich
    final Spielkarte karte = (Spielkarte) other;
    return this.farbe.equals(karte.farbe) && this.wert == karte.wert;
}
```

Hinweis: Vergleich von Gleitkommatypen `float` und `double`

Zum Vergleich von Gleitkommazahlen kann man sich eine Hilfsmethode `isEqualWithinPrecision(double, double, double)` wie folgt definieren:

```
public static boolean isEqualWithinPrecision(final double value,
                      final double expected, final double epsilon)
{
    return (value > expected - epsilon && value < expected + epsilon);
}
```

Damit werden alle Werte für den erwarteten und den zu prüfenden Wert als »gleich« angesehen, falls deren Differenz kleiner als »Epsilon« ist. ***Selbst damit ist für `equals(Object)` jedoch eine sinnvolle Aussage nicht garantiert!***

Typische Fehler bei der Implementierung von `equals()`

Bis jetzt scheinen die Umsetzung des Kontrakts und die Realisierung der Methode `equals(Object)` relativ einfach zu sein. Bei der unbedachten Implementierung kann man jedoch schnell Fehler machen. Betrachten wir dies anhand einer ersten, naiven Umsetzung für die Klasse `Person`. Es wird hier momentan bewusst auf die bereits kennengelernte Annotation `@Override` verzichtet, um auf ein Problem aufmerksam machen zu können. Eine erste Realisierung, die die Nachnamen vergleicht, könnte wie folgt aussehen:

```
public boolean equals(final Person otherPerson)
{
    return this.getName().equals(otherPerson.getName());
}
```

Zunächst scheint so weit alles in Ordnung zu sein, doch tatsächlich enthält diese Lösung zwei Fehler. Prüfen wir die Signatur und den Kontrakt:

- **Signatur** ✗
  Die gezeigte Methode besitzt die Signatur `equals(Person)`. Bei eigener Implementierung muss die Signatur korrekterweise den Eingabetyp `Object` verwenden, um beliebige Typen von Objekten vergleichen zu können.
- **Null-Akzeptanz** ✗
  Für jede Referenz `x` ungleich `null` sollte der Aufruf `x.equals(null)` den Wert `false` ergeben. Die obige Realisierung löst stattdessen eine `NullPointerException` durch den Aufruf von `getName()` auf einer `null`-Referenz aus.
- **Reflexivität** ✓
- **Symmetrie**[4] ✓
- **Transitivität** ✓
- **Konsistenz** ✓

Wir erkennen, dass die Methode eine falsche Signatur besitzt und darüber hinaus den Aspekt der Null-Akzeptanz des Kontrakts verletzt. Allerdings betreibt man hier ungewollt Overloading statt Overriding. Dieser Fehler fällt nicht auf, solange man nur Objekte gleichen Typs `T` explizit über `equals(T)` vergleicht. Zu Problemen kommt es erst, wenn `Person`-Objekte in Containerklassen des JDKs gespeichert werden oder Vergleiche mit `null` erfolgen. Die in der Klasse `Person` definierte Methode `equals(Person)` wird von den Containerklassen nicht aufgerufen, da sie *nicht* die benötigte Signatur besitzt. Stattdessen wird die Methode `equals(Object)` aus der Klasse `Object` aufgerufen, die einen Referenzvergleich von `Person`-Objekten statt des gewünschten Vergleichs von Attributwerten durchführt.

Neben diesen eher formalen Fehlern enthält diese Implementierung aber auch einen inhaltlichen Fehler. Zwei `Person`-Objekte werden bereits bei gleichem Namen als gleich angesehen. Herr Müller aus Kiel ist sicherlich nicht Herr Müller aus Hamburg.

[4] Weil sichergestellt ist, dass das zum Vergleich benutzte Attribut `name` nicht `null` ist.

Korrekturen Zur Korrektur folgen wir den Forderungen des Kontrakts und definieren die Methode mit der Signatur `equals(Object)`. Eine Realisierung sichert zunächst die Forderungen Null-Akzeptanz, Reflexivität und Typgleichheit ab. Anschließend werden alle diejenigen Attribute verglichen, die notwendig sind, um `Person`-Objekte eindeutig voneinander zu unterscheiden bzw. als inhaltlich gleich zu erkennen. Dadurch ergibt sich folgende, den Kontrakt erfüllende Methode:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit prüfen
        return false;

    final Person otherPerson = (Person) other;  // Attribute prüfen
    return this.getName().equals(otherPerson.getName()) &&
           this.getCity().equals(otherPerson.getCity()) &&
           this.getBirthday().equals(otherPerson.getBirthday());
}
```

Das ist bereits ein guter Schritt in die richtige Richtung. Wir können eine weitere Verbesserung erzielen, indem wir den Sourcecode zum Prüfen der Attribute in eine typsichere Hilfsmethode `compareAttributes(Person)` auslagern:

```
private boolean compareAttributes(final Person otherPerson)
{
    return this.getName().equals(otherPerson.getName()) &&
           this.getCity().equals(otherPerson.getCity()) &&
           this.getBirthday().equals(otherPerson.getBirthday());
}
```

Erfolgt die Extraktion von Attributprüfungen in eine separate Hilfsmethode, so lässt sich die Methode `equals(Object)` in eigenen Klassen einheitlich erstellen. Dort werden zunächst die formalen Kriterien abgesichert und dann jeweils die private, typspezifische und damit typsichere Hilfsmethode `compareAttributes()` aufgerufen:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit prüfen
        return false;

    final Person otherPerson = (Person) other;  // Attribute prüfen
    return compareAttributes(otherPerson);
}
```

Behandlung optionaler Attribute in `equals()`

Manchmal sind einige Attribute eines Objekts optional und dürfen daher den Wert `null` besitzen. Nehmen wir an, die Klasse `Person` wäre um die optionalen Attribute `street` und `zipcode` erweitert worden:

```
public final class Person
{
   private final String name;
   private final LocalDate birthday;
   private final String city;
   private String       street;      // Optional => null-Wert erlaubt
   private String       zipcode;     // Optional => null-Wert erlaubt

   public Person(final String name, final LocalDate birthday, final String city,
                 // Optionale Angaben
                 final String street, final String zipcode)
   {
      Objects.requireNonNull(name, "parameter 'name' must not be null");
      Objects.requireNonNull(birthday, "parameter 'birthday' must not be null");
      Objects.requireNonNull(city, "parameter 'city' must not be null");

      this.name = name;
      this.birthday = birthday;
      this.city = city;
      this.street = street;
      this.zipcode = zipcode;
   }

   @Override
   public boolean equals(final Object other)
   {
      if (other == null)
         return false;
      if (this == other)
         return true;
      if (this.getClass() != other.getClass())
         return false;

      final Person otherPerson = (Person) other;
      return compareAttributes(otherPerson);
   }

   // ...
```

Bis dato haben wir `null`-Werte für Attribute nicht betrachtet. Bei der Implementierung von `equals(Object)` ist dies aber speziell zu berücksichtigen. Man ist versucht, die Behandlung optionaler Attribute über eine `null`-Prüfung wie folgt zu realisieren:

```
private boolean compareAttributes(final Person otherPerson)
{
    return getName().equals(otherPerson.getName()) &&
           getCity().equals(otherPerson.getCity()) &&
           getBirthday().equals(otherPerson.getBirthday()) &&
       // ACHTUNG: falsche Behandlung optionaler Attribute
       (getStreet() != null && getStreet().equals(otherPerson.getStreet())) &&
       (getZipCode() != null && getZipCode().equals(otherPerson.getZipCode()));
}
```

Eine derartige Realisierung ist jedoch weder ausreichend noch korrekt: ***Zwei optionale Attribute, die jeweils den Wert `null` haben, werden nämlich fälschlicherweise nicht als gleich betrachtet!*** Wir lösen dies, indem wir zunächst einen Referenzvergleich durchführen, der sowohl `null`-Werte als auch eine mögliche Referenzgleichheit korrekt behandelt. Zur Überprüfung der inhaltlichen Gleichheit optionaler Attribute wird anschließend auf ungleich `null` verglichen und dann die Methode `equals(Object)` für das jeweilige Attribut aufgerufen:

```
private boolean compareAttributes(final Person otherPerson)
{
   return getName().equals(otherPerson.getName()) &&
          getCity().equals(otherPerson.getCity()) &&
          getBirthday().equals(otherPerson.getBirthday()) &&
       // korrigierte Behandlung optionaler Attribute
       (getStreet() == otherPerson.getStreet() ||
       (getStreet() != null && getStreet().equals(otherPerson.getStreet()))) &&
       (getZipCode() == otherPerson.getZipCode() ||
       (getZipCode() != null && getZipCode().equals(otherPerson.getZipCode())));
}
```

Ganz abgesehen davon, dass die auf diese Weise implementierten Vergleiche der optionalen Attribute schlecht lesbar sind, lässt die Ähnlichkeit der jeweiligen Abfragen ahnen, dass es eine elegantere Umsetzung geben sollte. Tatsächlich ist dem so, wenn man eine Hilfsmethode `nullSafeEquals(Object, Object)` in einer Utility-Klasse `EqualsUtils` wie folgt implementiert:

```
public static boolean nullSafeEquals(final Object obj1, final Object obj2)
{
    return (obj1 == obj2) || (obj1 != null && obj1.equals(obj2));
}
```

Durch den Einsatz dieser Methode können wir `null`-sichere Attributvergleiche klarer gestalten:

```
private boolean compareAttributes(final Person otherPerson)
{
    return getName().equals(otherPerson.getName()) &&
           getCity().equals(otherPerson.getCity()) &&
           getBirthday().equals(otherPerson.getBirthday()) &&
           // lesbare Behandlung optionaler Attribute
           EqualsUtils.nullSafeEquals(getStreet(), otherPerson.getStreet()) &&
           EqualsUtils.nullSafeEquals(getZipCode(), otherPerson.getZipCode());
}
```

IDIOM: NULL-SICHERE VERGLEICHE MIT DER UTILITY-KLASSE `Objects` Seit JDK 7 kann man die obigen Vergleiche durch den Einsatz der Utility-Klasse `Objects` noch kompakter schreiben. Diese Klasse haben wir bereits bei der Implementierung der Prüfung der Konstruktorargumente der Klasse `Person` kennengelernt. Zur Realisierung eigener `equals(Object)`-Methoden, die Attribute `null`-sicher vergleichen können sollen, ist in `Objects` eine statische Methode `equals(Object, Object)`

definiert, die diese Funktionalität bereitstellt. Mit diesem Wissen vereinfachen wir die Implementierung. Wir nutzen dazu folgendes Vorgehen (auch Idiom[5] genannt):

```
private boolean compareAttributes(final Person otherPerson)
{
    return getName().equals(otherPerson.getName()) &&
           getCity().equals(otherPerson.getCity()) &&
           getBirthday().equals(otherPerson.getBirthday()) &&
           // Einsatz der Utility-Klasse Objects
           Objects.equals(getStreet(), otherPerson.getStreet()) &&
           Objects.equals(getZipCode(), otherPerson.getZipCode());
}
```

Mit diesen letzten Schritten und Hinweisen haben wir das notwendige Rüstzeug für stabile Implementierungen von `equals(Object)` komplettiert – allerdings bisher noch ohne die Betrachtung der Auswirkungen von Vererbung. Das stellt der folgende Abschnitt vor.

Erweiterungen für `equals()` in Subklassen

Der Vergleich von Objekten nicht abgeleiteter Klassen war noch relativ einfach zu realisieren, da nach einer Typprüfung lediglich einige Attributwerte verglichen werden mussten. Das Ganze wird komplexer, wenn Objekte aus einer Klassenhierarchie zu vergleichen sind. Besonders die Prüfung auf Typgleichheit erfordert einige Überlegungen. Betrachten wir dies schrittweise anhand einer Klassenhierarchie, bestehend aus einer Basisklasse `BaseClass` und einer Subklasse `SubClass`.

Prüfung auf Typgleichheit Beginnen wir mit einigen Vorüberlegungen zur Implementierung der Typprüfung, da diese den Knackpunkt darstellt und sich teilweise nicht mehr konform zum `equals()`-Kontrakt erstellen lässt, wenn Subklassen die Basisklasse um Attribute und eine eigene `equals(Object)`-Methode erweitern.

Zunächst einmal besteht die Forderung nach Symmetrie, d. h., für beliebige Objekte `x` und `y` muss Folgendes gelten: Wenn `x.equals(y)` gilt, so gilt auch `y.equals(x)`. Nehmen wir dazu an, die Referenzvariable `x` wäre vom Typ `BaseClass` und `y` vom Typ `SubClass`:

- Ein Vergleich mit `instanceof` ist nicht symmetrisch, denn es gilt hierbei `y instanceof BaseClass`, aber nicht `x instanceof SubClass`.
- Ein Vergleich per `getClass()` liefert für beide Fälle `false` und ist symmetrisch.

Nutzt man `getClass()`, so ist man auf der sicheren Seite, wenn man die Methode `equals(Object)` konform zum Kontrakt implementieren möchte. Anhand des nachfolgenden Beispiels erkennt man, dass durch Einsatz von `instanceof` schnell die Gefahr besteht, die Symmetrie zu verletzen.

[5] Unter einem Idiom versteht man eine beispielhafte Umsetzung eines speziellen Problems auf Ebene des Sourcecodes. Idiome sind also »Entwurfsmuster« auf Programmzeilenebene.

```
public static void main(final String[] args)
{
  final BaseClass x = new BaseClass();
  final SubClass y = new SubClass();

  // instanceof prüfen
  System.out.println("y is-a BaseClass: " + (y instanceof BaseClass)); // true
  System.out.println("x is-a SubClass: " + (x instanceof SubClass));   // false

  // getClass() prüfen, Ergebnis: false
  System.out.println("getClass(): " + (x.getClass() == y.getClass()));
}
```

Listing 5.5 *Ausführbar als* **'EQUALSSYMMETRY'**

Für den Fall, dass nur Objekte des *exakt* gleichen Typs in `equals(Object)` als gleich angesehen werden sollen, empfiehlt es sich, die Typprüfungen über einen Aufruf der Methode `getClass()` durchzuführen. Instanzen von `BaseClass` und `SubClass` können so niemals gleich sein. Für diverse Anwendungsfälle ist das genau so auch gewünscht.

Es gibt jedoch einige Ausnahmen, nämlich falls Objekte von Basis- und Subklassen als gleich betrachtet werden sollen, wenn deren Wertebelegung der Attribute bzw. ihres Inhalts (semantisch) übereinstimmen. Das ist im JDK beispielsweise für verschiedene Implementierungen von Listen der Fall: Von diesen erwartet man intuitiv, dass sie als gleich angesehen werden, solange sie den Typ `java.util.List<E>` erfüllen und den gleichen Inhalt besitzen. Damit die Typprüfung für unterschiedliche Implementierungen von Listen nicht `false` liefert, nutzt man `instanceof`. Da dies, wie bereits bekannt, leicht die Forderung nach Symmetrie verletzt, darf ***lediglich die Basisklasse die Methode `equals(Object)` implementieren und darüber bestimmen, wann Subklassen als gleich angesehen werden sollen.*** Im JDK erfolgt daher für Listen die Implementierung in der abstrakten Basisklasse `java.util.AbstractList<E>`. Vereinfacht auf die Klassenhierarchie des Beispiels übertragen ergibt sich dann Folgendes:

```
class BaseClass
{
    private String attr1;
    private String attr2;

    @Override
    public boolean equals(final Object obj)
    {
        if (obj == this)
            return true;

        if (!(obj instanceof BaseClass))  // null-Prüfung + Typprüfung
            return false;

        final BaseClass otherBaseClass = (BaseClass) obj;
        return Objects.equals(this.attr1, otherBaseClass.attr1) &&
               Objects.equals(this.attr2, otherBaseClass.attr2);
    }
}
```

Die geführte Argumentation verdeutlicht Folgendes: Wird die Subklasse `SubClass` um eine `equals(Object)`-Methode erweitert, so wird die Symmetrie verletzt.

Folglich lassen sich Objekte einer Klassenhierarchie entweder nur typrein vergleichen oder erfordern die ausschließliche Implementierung von `equals(Object)` in der Basisklasse mit `instanceof`. Letzteres wird bei mehrstufigen Ableitungshierarchien unpraktikabel, da lediglich die Attribute der untersten Basisklasse ausschlaggebend für den Vergleich sein können.

Tipp: Schwierigkeiten beim Implementieren von `equals(Object)`

Falls Sie sich zunächst etwas schwer mit der korrekten Implementierung der Typprüfung von `equals(Object)` tun, so sind Sie in bester Gesellschaft: Cay Horstmann und Gary Cornell weisen in ihrem Buch »Core Java – Band 1 Grundlagen« [32] darauf hin, dass man auch im JDK verschiedenste Realisierungen von `equals(Object)` findet. Zum Teil wird mit `getClass()` geprüft, teilweise mit `instanceof` und manchmal wird sogar ganz auf die Typprüfung verzichtet. Dadurch kommt es schnell mal zu Verletzungen des `equals()`-Kontrakts.

Prüfung auf semantische Gleichheit Für Subklassen müssen in der Regel nicht nur die eigenen Attribute, sondern auch alle relevanten Bestandteile der Klassenhierarchie geprüft werden. Das kann man wie folgt realisieren:

1. **Zugriff auf Attribute der Basisklasse** – Wenn eine Subklasse Zugriff auf die Attribute der Basisklasse (direkt oder indirekt über `get()`-Methoden) besitzt, so könnte man die bisher gezeigte `equals(Object)`-Implementierung für die Subklasse derart erweitern, dass sie alle Attribute (auch die der Basisklassen) vergleicht.
   Bewertung: Diese Art der Umsetzung ist nicht OO-gemäß. Sie verstößt gegen die Kapselung und die Trennung von Zuständigkeiten. Neben diesen Designaspekten ist diese Art der Realisierung fragil, denn Änderungen in der Basisklasse bedingen Änderungen in Subklassen. Wird die Basisklasse um Attribute ergänzt, so muss dies beim Vergleich in der Subklasse berücksichtigt werden. Die Auswirkungen steigen mit der Anzahl an Subklassen. Daraus ergibt sich folgende Regel: »***Vermeide in `equals(Object)` direkte Zugriffe auf Attribute der Basisklasse!***«
2. **Aufruf von `equals()` der Basisklasse** – Ein Vergleich der Basisklassenbestandteile lässt sich elegant durch einen Aufruf von `equals(Object)` der Basisklasse regeln, der gegebenenfalls wiederum Aufrufe von `equals(Object)` an weitere Basisklassen delegiert. Direkte Subklassen von `Object` dürfen dies jedoch nicht tun, da ansonsten ein Referenzvergleich ausgeführt würde. Dieser gibt nahezu immer `false` zurück – außer wenn beide Objekte identisch sind.

Zur Realisierung von `equals(Object)` in Subklassen wird die zweite Variante eingesetzt. Will man sich nicht darauf verlassen, dass die drei bereits bekannten Prüfungen auf Null-Akzeptanz, Reflexivität und korrekten Typ schon in der Basisklasse durchgeführt werden, ergänzt man diese Prüfungen vorsichtshalber vor dem Vergleich der Bestandteile der Basisklassen. Anschließend kann ein Aufruf an eine Methode `compareAttributes()` erfolgen, die den Vergleich der Attribute ausführt:

```
public final class SubClass extends BaseClass
{
    // ...
    public boolean equals(final Object other)
    {
        // Drei Prüfungen nur vorsichtshalber
        if (other == null)                        // Null-Akzeptanz
            return false;
        if (this == other)                        // Reflexivität
            return true;
        if (this.getClass() != other.getClass()) // Typgleichheit prüfen
            return false;

        // Delegation an Basisklasse
        if (!super.equals(other))
            return false;

        final SubClass subClass = (SubClass)other;
        return compareAttributes(subClass);
    }
    // ...
```

Fazit

Dieser Abschnitt ist ausführlich auf die Grundlagen und auf mögliche Fallstricke bei der Implementierung der Methode `equals(Object)` eingegangen, weil sich dabei recht schnell Fehler mit großen Nebenwirkungen einschleichen können. Daher ist es extrem wichtig, diese Methode konform zu ihrem Kontrakt zu implementieren: Insbesondere die korrekte Funktionsweise der Containerklassen des JDKs basiert darauf. Ergänzend sei auch nochmals auf Folgendes hingewiesen: Wenn man `equals(Object)` überschreibt, sollte man ebenfalls die Methode `hashCode()` konform dazu implementieren. Dies wird später in Abschnitt 6.1.7 besprochen. Auf mögliche Fallstricke geht dann Abschnitt 6.1.9 genauer ein.

5.2 Primitive Typen und Wrapper-Klassen

In diesem Unterkapitel beschäftigen wir uns mit primitiven Datentypen sowie den dazu korrespondierenden Wrapper-Klassen. Einführend stelle ich grundlegende Eigenschaften vor. Danach lernen wir die Konvertierung und Verarbeitung von Werten kennen. Insbesondere gehe ich dabei auf Besonderheiten des sogenannten Auto-Boxings und Auto-Unboxings ein, das eine automatische Konvertierung von einem primitiven Typ in den korrespondierenden Wrapper-Typ durchführt und andersherum. Abschließend

betrachten wir dann die Ausgabe und Verarbeitung von Zahlen und lernen neben der Dezimaldarstellung die Binär-, Oktal- und Hexadezimaldarstellung sowie eine Repräsentation von Zahlen als Bitdarstellung und Operationen dafür kennen.

5.2.1 Grundlagen

Fast alles ist in Java ein Objekt. Zusätzlich existieren verschiedene primitive Datentypen, die Werte ohne Verhalten repräsentieren. In Java sind das die Typen `byte`, `short`, `int`, `long`, `float` und `double` für Zahlen, `boolean` für boolesche Werte sowie der Typ `char` für einzelne Zeichen. Für die primitiven Datentypen gibt es ***Wrapper-Klassen***, die die primitiven Datentypen als unveränderliches Objekt darstellen und mit etwas Funktionalität »umhüllen«, z. B. das Parsen eines Werts aus einem String.

Tabelle 5-1 listet die primitiven Typen und korrespondierende Wrapper-Klassen auf, nennt deren Anzahl an Bits sowie deren Wertebereich. Für die Wrapper-Klassen sind die Wertebereiche über jeweils eigene Konstanten `MIN_VALUE` und `MAX_VALUE` definiert. Darüber hinaus wissenswert ist, dass die Wrapper-Klassen alle aus dem Package `java.lang` stammen und die Zahlentypen `Short`, `Integer`, `Long`, `Float` und `Double` die gemeinsame Basisklasse `java.lang.Number` besitzen.

Tabelle 5-1 *Primitive Datentypen und korrespondierende Wrapper-Klassen*

Primitiver Typ	Wrapper	Bits	Wertebereich
`byte`	`Byte`	8	-2^7 .. 2^7-1 (-128 .. 127)
`short`	`Short`	16	-2^{15} .. $2^{15}-1$ (-32.768 .. 32.767)
`int`	`Integer`	32	-2^{31} .. $2^{31}-1$ (-2.147.483.648 .. 2.147.483.647)
`long`	`Long`	64	-2^{63} .. $2^{63}-1$ (-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807)
`float`	`Float`	32	$\pm 1.4e-45$.. $\pm 3.4e+38$
`double`	`Double`	64	$\pm 4.9e-324$.. $\pm 1.8e+308$
`boolean`	`Boolean`	-	`false` und `true`
`char`	`Character`	16	0 .. $2^{16}-1$ (0 (`\u0000`) .. 65535 (`\uffff`))

Info: Motivation für die Existenz primitiver Datentypen

Die Gründe für die Existenz primitiver Datentypen in einer ansonsten objektorientierten Sprache sind, dass nur so eine Ansteuerung von Hardware und eine Kompatibilität zu C++-Aufrufen über JNI (Java Native Interface) möglich wird. Weiterhin verbrauchen primitive Datentypen weniger Speicher und müssen nicht per `new` erzeugt werden. Dadurch wird eine Optimierung der Programmlaufzeit und des benötigten Speicherplatzes erreicht (vgl. Kapitel 22 »Optimierungen«).

Tipp: Wahl von primitiven Datentypen

Bei Berechnungen mit Ganzzahlen bietet es sich an, bevorzugt mit `int` und `long` zu arbeiten. Nur selten ist der Einsatz von `byte` oder `short` sinnvoll. Bei Gleitkommazahlen sollte man möglichst `double` statt `float` nutzen, da Letzteres eine geringe Genauigkeit bei Berechnungen mit mehreren Nachkommastellen aufweist. Generell sollte man bei Gleitkommatypen immer Rundungsprobleme bedenken.

Interne Darstellung und das Zweierkomplement

Die Interpretation der Zahlen erfolgt gemäß dem sogenannten ***Zweierkomplement***, das beschreibt, wie Zahlen in ein Bitmuster umgerechnet werden. Mit n als Anzahl der Bits ergibt sich ein Wertebereich von $-2^{n-1}, \ldots, +2^{n-1} - 1$. Für ein Byte entspricht dies dem Wertebereich $-2^7, \ldots, +2^7 - 1 = -128 \ldots +127$. Das Zweierkomplement bildet vorzeichenbehaftete Zahlen wie folgt ab: Jede *positive* Zahl wird in ein Bitmuster mit einer führenden Null umgewandelt. Jede *negative* Zahl wird zunächst in das Bitmuster der entsprechenden positiven Zahl umgerechnet, danach wird dieses invertiert und der Wert 1 addiert. Damit ergeben sich folgende Bitmuster für den Typ `byte`:

```
+127 = 01111111
 ...
+  1 = 00000001
   0 = 00000000
-  1 = 11111111
 ...
-127 = 10000001
-128 = 10000000
```

Wissenswertes zu primitiven Datentypen

Wie aus der vorherigen Tabelle ersichtlich, besitzen die primitiven Datentypen `byte`, `short`, `int` und `long` unterschiedliche Wertebereiche und repräsentieren Ganzzahlen. Gleitkommazahlen werden durch die Typen `float` und `double` modelliert. Für die verschiedenen primitiven Datentypen gibt es spezielle Literale (Zeichenfolgen), die Repräsentanten oder Werte des primitiven Typs sind. Ganzzahlenliterale sind ohne weitere Angabe vom Typ `int`:

```
int nr = 4711;
int second = 2;

// Achtung: 'l' ist sehr leicht mit der Ziffer '1' zu verwechseln
long longNumber1 = 4711l;
long longNumber2 = 4711L;
```

Im Listing ist gezeigt, dass man durch ein nachgestelltes kleines `l` bzw. großes `L` eine Zahl vom Typ `long` definieren kann. Aufgrund der besseren Lesbarkeit und der leichteren Unterscheidbarkeit von der Ziffer `1` empfiehlt sich das Zeichen `L` statt `l`.

Bei der Verarbeitung von Gleitkommazahlen (auch Floating-Point genannt) sind die Werte im Gegensatz zu den Ganzzahlen standardmäßig immer vom »größeren« Datentyp `double`. Optional können `double`s durch Angabe von `d` bzw. `D` explizit gekennzeichnet werden. Dies ist aber eigentlich nur für Werte vom Typ `float` notwendig: Ein nachgestelltes `f` bzw. `F` »verkleinert« Werte auf den Typ `float`. Das Gesagte wird durch folgendes Listing verdeutlicht:

```
double pi = 3.14159;
float  one_quarter = 1/4F;
double pi_2 = 3.14159D;
```

Merkwürdigkeiten bei Berechnungen Um Sie für mögliche Probleme bei Berechnungen zu sensibilisieren, zeige ich ein paar einfache Divisionen, die Sie durch Start des Programms DIVISIONROUNDINGPROBLEMS nachvollziehen können:

```
System.out.println(7 / 2);      // 3
System.out.println(7 / 2.0);    // 3.5
System.out.println(7.0 / 2);    // 3.5
System.out.println(7.0 / 2.0);  // 3.5
```

Man kann sich bei der ersten Division fragen, wieso denn das Ergebnis 3 und nicht 3.5 ist, wie man es erwarten würde. Das liegt ganz einfach daran, dass die erste Division eine Operation auf zwei `int`-Werten ist und diese besitzen nun mal keine Nachkommastellen. Erst wenn einer der Operanden eine Gleitkommazahl ist, wird auch das Ergebnis zu einer Gleitkommazahl und man erhält das mathematisch korrekte Ergebnis.

Selbst einfache Berechnungen mit den Gleitkommatypen `float` und `double` können bereits zu Ungenauigkeiten führen, was durch deren Repräsentation gemäß dem Format IEEE 754[6] bedingt ist. Folgendes Beispiel einer `for`-Schleife, die zehnmal den Wert `0.1` addiert und anschließend die Abweichung der berechneten Summe mit dem erwarteten Wert `1` ausgibt, macht mögliche Probleme eindrucksvoll deutlich. Außerdem werden merkwürdige Zwischenstände sowie eine Abweichung zwischen Literalen mit gleichem Zahlenwert für die Typen `float` und `double` gezeigt:

```
public static void main(final String[] args)
{
   float sum = 0.0f;
   for (int i = 0; i < 10; i++)
      sum += 0.1;

   System.out.println("sum:  1 != " + sum);
   System.out.println("3 * add  = " + (0.1 + 0.1 + 0.1));
   System.out.println("3 * 0.1  = " + 3 * 0.1);
   System.out.println("7 * 0.1  = " + 7 * 0.1);
   System.out.println("compare  = " + (0.3f == 0.3d)); // false, Float != Double
}
```

Listing 5.6 *Ausführbar als* **'FLOATPRECISIONEXAMPLE'**

[6] http://de.wikipedia.org/wiki/IEEE_754

Das Programm FLOATPRECISIONEXAMPLE produziert folgende Ausgaben:

```
sum:  1 != 1.0000001
3 * add  = 0.30000000000000004
3 * 0.1  = 0.30000000000000004
7 * 0.1  = 0.7000000000000001
compare  = false
```

Anhand der Ausgaben erkennt man mögliche Überraschungen durch Rundungsfehler ziemlich gut. Insbesondere erwähnenswert ist, dass man zunächst darüber verwundert sein könnte, dass gleiche Zahlenliterale für `float` und `double` verschiedene Werte besitzen. Nach kurzem Nachdenken wird dies durch die unterschiedliche Anzahl an Nachkommastellen und die dort auftretenden Rundungsfehler verständlich.

Wissenswertes zu Modulo und negativen Zahlen Neben Multiplikation und Division wird auch die Modulo-Operation (%) recht häufig verwendet. Sie dient dazu, den Rest einer Division zu ermitteln. Dies kann man beispielsweise dazu nutzen, um zu prüfen, ob eine Zahl gerade ist. Dazu vergleicht man den Rest mit dem Wert 0. Wenn man nun feststellen möchte, ob eine Zahl ungerade ist, könnte man auf die Idee kommen, zu prüfen, ob der Rest 1 ist. Dabei kann man Überraschungen erleben. Folgende Zeilen zeigen einige Beispiele:

```
public void evenAndOddChecks()
{
    // hier scheint noch alles okay zu sein ...
    System.out.println("2 is even? " + (2 % 2 == 0));
    System.out.println("-2 is even? " + (-2 % 2 == 0));

    // hier kommen die Merkwürdigkeiten ...
    System.out.println("3 is odd? " + (3 % 2 == 1));
    System.out.println("-3 is odd? " + (-3 % 2 == 1));
    System.out.println("-3 is odd? " + (-3 % 2 == -1));
}
```

Diese Methode gibt Folgendes aus:

```
2 is even? true
-2 is even? true
3 is odd? true
-3 is odd? false
-3 is odd? true
```

Wie man sieht, wird für ungerade negative Zahlen überraschenderweise das falsche Ergebnis ausgegeben. Die zweite Prüfung zeigt, dass die Modulo-Operation für negative Zahlen negative Werte liefert. Somit ist folgende Methode leider nicht ganz korrekt:

```
private static boolean isOdd(final int val)
{
    return (val % 2) == 1;
}
```

Diese Methode liefert nämlich für negative Eingaben falsche Ergebnisse. Eine Korrektur fällt nicht schwer, wenn man sich des Problems bewusst ist. Statt auf gleich 1 prüft man auf ungleich 0 und erhält damit folgende korrigierte und korrekte Realisierung:

```
private static boolean isOddCorrected(final int val)
{
    return (val % 2) != 0;
}
```

Binär-, Oktal- und Hexadezimaldarstellung

Neben der gebräuchlichen Darstellung von Zahlen in der Dezimalschreibweise wird in der Informatik recht häufig auch eine Binär-, Oktal- und Hexadezimaldarstellung benötigt (also Zahlen mit der Basis 2, 8 bzw. 16). Zur Unterscheidung der Zugehörigkeit von Literalen zu Zahlensystemen gibt es Präfixe: '`0x`' bzw. '`OX`' steht für hexadezimale Zahlen, eine führende `0` für Oktalzahlen. Für Binärzahlen dient das Präfix '`0b`' bzw. '`0B`', gefolgt von einer Folge von Nullen und Einsen. Für Oktalzahlen dürfen nur die Ziffern 0 – 7 genutzt werden. Für hexadezimale Zahlen sind neben den Ziffern 0 – 9 auch die Buchstaben a – f bzw. A – F erlaubt, die die Werte 10 – 15 repräsentieren. Mit diesem Wissen betrachten wir Zahlenliterale in den verschiedenen Formaten:

```
public static void main(final String[] args)
{
    final int octalLiteral = 0567;
    final int hexLiteral = 0xABC;
    final int binaryLiteral = 0b01101001;   // Seit JDK 7

    System.out.println("octalLiteral " + octalLiteral);
    System.out.println("hexLiteral " + hexLiteral);
    System.out.println("binaryLiteral " + binaryLiteral);
}
```

Listing 5.7 *Ausführbar als* '**BINARYOCTALHEXADECIMALEXAMPLE**'

Die Notation für Oktalliterale rein mit führender Null und ohne Buchstabenkürzel birgt den Fallstrick, dass jede Zahl mit einer führenden Null diese automatisch zu einer Zahl im Oktalsystem macht, auch wenn dies eventuell gar nicht beabsichtigt war.

Unterstriche in numerischen Literalen

Zur übersichtlicheren Notation von Konstanten ist es erlaubt, Ziffern in Ganzzahlenliteralen mit einem Unterstrich zu separieren. Dabei sind auch mehrere Unterstriche zur Separation erlaubt, allerdings nicht direkt am Anfang oder Ende eines Zahlenliterals – aber es sind durchaus auch mehrere Unterstriche direkt nacheinander erlaubt. Das Ganze kann man unter anderem zur Trennung von Nibbles (4 Bits) bei Bytewerten oder zur Simulation von Tausenderpunkten folgendermaßen einsetzen:

```
final byte binaryLiteral = 0b0110_1001;
final int oneMillion = 1_000__000;
```

Typ- und Wertebereichserweiterungen sowie -verkleinerungen

Variablen können unterschiedliche Typen besitzen und müssen teilweise aufeinander abgebildet werden. Dabei sind verschiedene Dinge zu beachten.

Widening Eine Wertebereichserweiterung ist in jedem Fall ungefährlich, weil etwa ein `long` einen größeren Wertebereich besitzt als ein `int` und somit alle Zahlen darstellen kann, die auch der kleinere Datentyp enthält. Dabei findet ein sogenanntes ***Widening*** statt, das folgende Kette durchläuft:

$$\texttt{byte} \Rightarrow \texttt{short} \Rightarrow \texttt{int} \Rightarrow \texttt{long} \Rightarrow \texttt{float} \Rightarrow \texttt{double}$$

Narrowing Was passiert aber, wenn man den Wertebereich verkleinert? Das stellt potenziell eine gefährliche Operation dar, weil eventuell Informationen verloren gehen, etwa für den Fall, dass man den Wert 1.000.000 einer Variablen vom Typ `short` zuweist, die maximal Werte bis 32767 darstellen kann, z. B. folgendermaßen:

```
final short truncated = (short)1_000_000;
System.out.println("truncated: " + truncated);
```

Diese Zeilen lassen sich als Programm SHORTTRUNCATIONEXAMPLE starten. Als Ausgabe erhält man dann Folgendes:

```
truncated: 16960
```

Derartige Typverkleinerungen (auch ***Narrowing*** genannt) bergen ganz offensichtlich die Gefahr für einen Informationsverlust und eine falsche Berechnung. Daher sind die Casts explizit zu notieren. Es gibt eine Ausnahme, die wir nun anschauen.

Besonderheit: Auto-Narrowing So praktisch eine Fehlermeldung bei Narrowing in der Regel ist, so gilt dies nicht, wenn ein Zahlenliteral einen Wert darstellt, der kompatibel zu dem kleineren Typ ist. Folgendes Beispiel macht dies deutlich. Hier sehen wir die Zahl 4711, die offensichtlich zuweisungskompatibel zu einem `short` ist, weil sie in dessen Wertebereich liegt:

```
final short validShort = (short)4711;
```

Der Cast war bis JDK 6 notwendig, weil, wie zuvor erwähnt, ja alle Ganzzahlenliterale – wenn nicht anders spezifiziert – vom Typ `int` sind. Der Compiler hat hier einen Verstoß gesehen, obwohl eigentlich kein Problem vorlag. Das war für uns als Entwickler unpraktisch. Mittlerweile werden die Wertebereiche der Literale auf Gültigkeit geprüft und es findet ein sogenanntes Auto-Narrowing statt. Diese automatische Typverkleinerung erlaubt es, auf den Cast zu verzichten, wodurch sich die Lesbarkeit verbessert:

```
final short validShort = 4711;   // Auto-Narrowing seit JDK 6
// final short invalid = 65535;  // Compile-Error
```

Wertebereichsüberläufe und Silent Fail Für Berechnungen mit primitiven Datentypen ist explizit zu beachten, dass sie »überlaufen« können! Was bedeutet das? Nehmen wir an, einer Variablen ist ein Wert zugewiesen, der nahe des Maximalwerts dieses Datentyps ist, beispielsweise `MAX_VALUE - 7`. Addiert man dazu den Wert 10, dann führt dies nicht etwa zu einer Fehlermeldung, sondern zu einem Überlauf im Wertebereich. Sehen wir uns folgendes Programm an:

```
public static void main(final String[] args)
{
    final int value = Integer.MAX_VALUE - 7;
    System.out.println("Integer.MAX_VALUE - 7 + 10 = " + (value + 10));
}
```

Listing 5.8 *Ausführbar als* '**INTEGEROVERFLOWEXAMPLE**'

Das Programm INTEGEROVERFLOWEXAMPLE produziert folgende Ausgabe:

```
MAX_VALUE - 7 + 10 = -2147483646
```

Die merkwürdige Ausgabe liegt an der internen Repräsentation der Ganzzahlen durch das Zweierkomplement. Bei der oben erwähnten Addition bewegt man sich (vermutlich unerwartet) in den negativen Wertebereich, weil Folgendes gilt:

```
Integer.MAX_VALUE + 1 == Integer.MIN_VALUE
       2147483647 + 1 == -2147483648
```

Mit diesem Wissen und den Anmerkungen aus dem nachfolgenden Praxistipp sind Sie besser für Merkwürdigkeiten bei Berechnungen gewappnet und können mögliche Fehler bereits beim Programmieren vermeiden, wenn Sie an diese Dinge denken.

Verbesserungen in JDK 8 Wertebereichsüberläufe in Kombination mit Silent Fail und daraus resultierende Probleme haben in JDK 8 zu einer Erweiterung in der Klasse `java.lang.Math` geführt. Dort sind für gebräuchliche mathematische Operationen wie +, -, * usw. korrespondierende Methoden wie `addExact()`, `subtractExact()` oder `multiplyExact()` definiert, die bei den entsprechenden Operationen auf Wertebereichsüberläufe prüfen und dafür Exceptions auslösen.

Hinweis: Anomalien bezüglich `MIN_VALUE`

Für die beiden Datentypen `int` und `long` gilt die Anomalie `MIN_VALUE == -MIN_VALUE`. Interessanterweise gilt dies nicht für die Typen `byte` und `short`:

```
public static void main(final String[] args)
{
    System.out.println(Byte.MIN_VALUE == -Byte.MIN_VALUE);        // false
    System.out.println(Short.MIN_VALUE == -Short.MIN_VALUE);      // false
    System.out.println(Integer.MIN_VALUE == -Integer.MIN_VALUE);  // true
    System.out.println(Long.MIN_VALUE == -Long.MIN_VALUE);        // true
}
```

Listing 5.9 *Ausführbar als* '**MINVALUETEST**'

Konstruktion von Wrapper-Klassen und weitere Konvertierungen

Wrapper-Objekte kann man durch einen Konstruktoraufruf mit einem String oder einem Wert eines primitiven Datentyps erzeugen – allerdings empfiehlt sich eher die Nutzung einer der überladenen `valueOf()`-Methode, weil so unnötige Objektkonstruktionen verhindert werden können. Dieser Umstand führt auch dazu, dass die Konstruktoren in Java 9 als deprecated markiert sind:

```
// nicht empfohlene Variante
final Integer valueByLiteral = new Integer(7);
final Integer valueByString = new Integer("14");

// bevorzugter Weg
final Integer valueByLiteral2 = Integer.valueOf(7);
final Integer valueByString2 = Integer.valueOf("14");
```

Natürlich entspricht nicht jeder beliebige String einer Zahl. Um Fehler beim Konvertieren auszudrücken, kann für Parameter vom Typ `String` eine `java.lang.NumberFormatException` ausgelöst werden. Auch die beiden Wrapper-Klassen `Boolean` und `Character` weisen einige Besonderheiten auf: Für `Boolean`s wird nur die textuelle Eingabe `"true"` – unabhängig von Groß- und Kleinschreibung – in den Wert `Boolean.TRUE` gewandelt. Jeder andere Wert, auch `null`, wird akzeptiert und ergibt `Boolean.FALSE`. Zudem existiert für den `Character`-Wrapper kein Konstruktor, der einen String entgegennimmt, da ein `char` nur ein einzelnes Zeichen speichert.

Die Wrapper-Klassen ermöglichen, Werte primitiver Datentypen aus verschiedenen Repräsentationsformen (String, Typ-Literal) in Objekte umzuwandeln (***Boxing***). Aus einem solchen Wrapper-Objekt kann wieder ein primitiver Datentyp ermittelt werden (***Unboxing***). Dadurch können primitive Datentypen und Wrapper-Klassen nahezu ohne Unterschied verwendet werden. Sofern benötigt, findet eine Umwandlung automatisch statt. Man spricht daher auch von ***Auto-Boxing*** und ***Auto-Unboxing***.

```
final Integer autoBoxing = 7;               // int -> Integer
final int autoUnboxing = new Integer(4711); // Integer -> int
```

Hierbei gibt es aber folgende Einschränkung: Es können keine Methoden auf primitiven Typen aufgerufen werden: Beispielsweise ist `5.intValue()` nicht möglich.

5.2.2 Konvertierung von Werten

Nachdem wir einiges an Wissen über primitive Datentypen erworben haben, wollen wir uns nun mit der Konvertierung von Werten beschäftigen, etwa um Strings in Wrapper-Instanzen und Wrapper-Instanzen in Werte primitiver Typen sowie umgekehrt umzuwandeln. Abbildung 5-1 zeigt die Konvertierungen.

- **String ⇒ Wrapper** – Die Umwandlung eines Strings in einen Wrapper erfolgt durch statische `valueOf()`-Methoden der jeweiligen Wrapper-Klasse bzw. durch Übergabe eines Strings an deren Konstruktor.

Abbildung 5-1 *Konvertierung von Wrappern, Strings und primitiven Werten*

- **Wrapper ⇒ primitiver Typ** – Objekte der Wrapper-Klassen kann man über deren Objektmethoden `xyzValue()`, etwa `integerObj.intValue()`, in Werte eines primitiven Typs umwandeln. `xyz` steht dabei für alle möglichen primitiven Typen, also `byte`, `short`, `int`, `long`, `float` oder `double`. Zusätzlich kann immer noch in andere primitive Zahlentypen umgewandelt werden, etwa `byte`, `short` oder `int`.
- **String ⇒ primitiver Typ** – Mithilfe von statischen `parseXYZ()`-Methoden der jeweiligen Wrapper-Klasse kann man einen Zahlen repräsentierenden String direkt in einen Wert eines primitiven Typs umwandeln. `XYZ` steht dabei für alle möglichen primitiven Typen, also `byte`, `short`, `int`, `long`, `float` oder `double`, allerdings wird hier der erste Buchstabe großgeschrieben, also etwa `Integer.parseInt()`. Diese Methoden sind eine Abkürzung der Hintereinanderausführung der Methoden `valueOf()` und `xyzValue()`. Die bislang beschriebenen Methoden nehmen eine Dezimaldarstellung an, wie behandelt man aber oktale oder hexadezimale Angaben? Falls die Angabe im String nicht im Dezimalformat, sondern oktal oder hexadezimal vorliegt, kann man die Methode `decode()` der Klasse `Integer` bzw. `Long` nutzen.
- **Primitiver Typ ⇒ String** – Um einen primitiven Typ in einen String umzuwandeln, gibt es in der Klasse `String` überladene statische `valueOf()`-Methoden. Diese werden u. a. zur Umwandlung bei Ausgaben mit `toString()` eingesetzt. Das haben wir bereits in Abschnitt 5.1.1 kennengelernt.

Fallstricke bei der Umwandlung mit `decode(String)` Seit JDK 7 kann man zwar Binärdarstellungen und Unterstriche in Zahlenliteralen nutzen, jedoch wurden die statischen Methoden `decode(String)` der Klasse `Integer` bzw. `Long` leider auch bis einschließlich JDK 14 noch nicht derart erweitert, dass sie diese beiden Besonderheiten verarbeiten können. Das Programm INTEGERDECODINGPROBLEMSEXAMPLE macht dies deutlich. Bei dessen Ausführung kommt es zu einer `java.lang.NumberFormatException: For input string: "1_000_000"`.

Tipp: Fallstricke im API

Ein API sollte möglichst so gestaltet sein, dass es Fehler ausschließt. Leider existieren in den Wrapper-Klassen drei Methoden, die Fehler provozieren. In den Klassen `Boolean`, `Integer` und `Long` sind jeweils Methoden mit den Namen `getBoolean(String)`, `getInteger(String)` bzw. `getLong(String)` definiert. Diese Methoden nehmen einen Parameter vom Typ `String` entgegen. Intuitiv würde man ein Parsing und eine Umwandlung in eine Zahl erwarten, wie dies beim Aufruf von `valueOf(String)` geschieht. Tatsächlich erfolgt durch die zuvor genannten Methoden jedoch ein Zugriff auf die System-Properties. Zu dem übergebenen String wird der Wert des Parameters gleichen Namens ermittelt. Anschließend erfolgt eine Konvertierung in ein Objekt der jeweiligen Wrapper-Klasse. ***Diese Funktionalität ist dort nicht nur überflüssig, sondern führt auch zu einer Kopplung an die Klasse `System`.*** Das ist unerwartet und unnatürlich.

Beispiel: Parsen von Zahlen

Möchte man prüfen, ob ein beliebiger Eingabestring eine gültige Ganzzahl darstellt, so kann man dafür die statische Methode `Integer.parseInt(String)` verwenden. Aufgrund des beschränkten Wertebereichs eines `Integer`s ist diese jedoch nur für Eingabewerte bis etwas über 2 Milliarden geeignet. Für größere Wertebereiche lässt sich die statische Methode `Long.parseLong(String)` nutzen. Für Gleitkommazahlen sollte das Parsing mithilfe der statischen Methoden `parseFloat(String)` bzw. `parseDouble(String)` der Klassen `Float` bzw. `Double` erfolgen.

Folgendes Beispiel zeigt eine Prüffunktionalität, um festzustellen, ob ein übergebener String `numberAsText` in eine Gleitkommazahl umgewandelt werden kann:

```
private static boolean isFloatingNumber(final String numberAsText)
{
    try
    {
        Double.parseDouble(numberAsText);
        return true;
    }
    catch (final NumberFormatException ex)
    {
        return false;
    }
}

public static void main(final String[] args)
{
    System.out.println("isNumber(47) " + isFloatingNumber("47"));       // true
    System.out.println("isNumber(47a) " + isFloatingNumber("47a"));     // false
    System.out.println("isNumber(47.11) " + isFloatingNumber("47.11")); // true
    System.out.println("isNumber(47,11) " + isFloatingNumber("47,11")); // false
}
```

Listing 5.10 *Ausführbar als* '**PARSENUMBEREXAMPLE**'

Für Nachkommastellen wird nur die amerikanische Notation mit einem Punkt statt eines Kommas korrekt ausgewertet. Benötigt man mehr Flexibilität, so müssen landesspezifische Formatierungen genutzt werden. Das wird in Abschnitt 13.1.4 besprochen.

5.2.3 Wissenswertes zu Auto-Boxing und Auto-Unboxing

Wie bereits erwähnt, stellen Auto-Boxing und Auto-Unboxing eine automatische Konvertierung zwischen primitiven Datentypen und deren korrespondierenden Wrapper-Klassen dar. Dies erleichtert zum Teil die Handhabung sowie die Lesbarkeit. Diese Vereinfachungen haben aber einige unangenehme Nebeneffekte, die man kennen sollte.

Vergleich von primitiven Werten und Wrappern

Bis einschließlich JDK 1.4 konnte man primitive Typen nicht direkt mit den korrespondierenden Wrapper-Klassen vergleichen. Durch die mit JDK 5 eingeführte automatische Konvertierung sind nachfolgend gezeigte Vergleiche syntaktisch möglich:

```
// Auto-Unboxing von Integer: 7 == new Integer(7)
System.out.println(7 == new Integer(7));                    // true
System.out.println(7777 == new Integer(7777));              // true

// Auto-Unboxing (ebenfalls)
System.out.println(new Integer(7) == 7);                    // true
System.out.println(new Integer(7777) == 7777);              // true
System.out.println(new Integer(7777) == new Integer(7777)); // false
```

Listing 5.11 *Ausführbar als* **'PRIMITIVESANDWRAPPERCOMPAREEXAMPLE'**

Beim Vergleich von primitiven Werten mit deren Wrapper findet immer ein Auto-Unboxing statt, wodurch der zweitletzte Vergleich `true` liefert. Das mag überraschen, wenn man denkt, dass der rechte Operand durch Auto-Boxing in einen `Integer` gewandelt wird. Dass dem nicht so ist, zeigt der letzte Vergleich zweier `Integer`-Instanzen. Bevor wir uns gleich weitere Fallstricke für die Wrapper-Klassen anschauen, möchte ich kurz erwähnen, dass die Wrapper-Klassen die `equals(Object)`-Methode überschreiben und dort den Wert des ummantelten primitiven Typs vergleichen. Dadurch liefern auch die neu konstruierten Objekte für den Wert 7777 Gleichheit:

```
System.out.println(new Integer(7777).equals(7777));              // true
System.out.println(new Integer(7777).equals(new Integer(7777))); // true
```

Nebenwirkungen und Fallstricke beim Auto-Boxing

Auto-Boxing führt zu einem geringfügig höheren Speicherverbrauch (z. B. 16 Bytes für ein `Integer`-Objekt statt 4 Bytes für einen `int`) und geht mit einer minimal schlechteren Ausführungsgeschwindigkeit einher, wenn extrem viele solcher Konvertierungen erfolgen. Um dem entgegenzuwirken, wurden in den Klassen `Integer` und `Long` je-

weils Caches für Werte im Bereich von `-128` bis `127`[7] eingeführt. Genau diese Optimierung durch das Caching in Kombination mit Auto-Boxing führt aber zu Merkwürdigkeiten beim Vergleich von `Integer`- bzw. `Long`-Objekten.

Die Umwandlung von Zahlenliteralen durch Auto-Boxing verwendet zum Ermitteln des korrespondierenden Wrapper-Objekts implizit die statische Methode `valueOf()`. Dort wird zunächst der Zahlenwert geprüft. Stammt der angeforderte Wert aus dem Wertebereich des Caches, so werden keine neuen Objekte erzeugt, sondern Referenzen auf im Cache gespeicherte Werte zurückgeliefert. Dies vermeidet zwar unter Umständen die Erzeugung vieler temporärer Objekte, löst aber Merkwürdigkeiten beim Referenzvergleich aus, wie dies nachfolgendes Beispiel zeigt:

```
public static void main(final String[] args)
{
    // Cache = Objekt aus dem Cache, New = Neues Objekt
    final Integer i1 = 7;                            // Auto-Boxing => Cache
    final Integer i2 = 4711;                         // Auto-Boxing => New

    System.out.println(i1 == new Integer(7));        // false, Cache != New
    System.out.println(i2 == new Integer(4711));     // false, New != New
    System.out.println(i1 == Integer.valueOf(7));    // TRUE !!!, Cache == Cache
    System.out.println(i2 == Integer.valueOf(4711)); // false, New != New
}
```

Listing 5.12 *Ausführbar als* '**AUTOBOXINGCACHEEXAMPLE**'

Die dargestellten Vergleiche sind allesamt Referenzvergleiche auf `Integer`-Objekten. Die Umwandlung der Zahlenliterale `7` und `4711` erfolgt durch die statische Methode `Integer.valueOf(int)`. Der dritte Vergleich `i1 == Integer.valueOf(7)` liefert daher (und nur zunächst überraschend) den Wert `true`, weil identische Referenzen aus dem Cache verglichen werden. Die restlichen Vergleiche liefern erwartungsgemäß den Wert `false`, da es sich jeweils um unterschiedliche Referenzen handelt.

Nebenwirkungen und Fallstricke beim Auto-Unboxing

Nicht nur beim Auto-Boxing, sondern auch beim Auto-Unboxing gibt es Merkwürdigkeiten. Im folgenden Beispiel werden zwei `Integer`-Objekte `i1` und `i2` per Konstruktor erzeugt. Danach werden scheinbar drei mathematische Vergleiche durchgeführt:

```
public static void main(final String[] args)
{
    final Integer i1 = new Integer(1);
    final Integer i2 = new Integer(1);

    System.out.println("i1 >= i2: " + (i1 >= i2)); // true
    System.out.println("i1 <= i2: " + (i1 <= i2)); // true
    System.out.println("i1 == i2: " + (i1 == i2)); // false
}
```

Listing 5.13 *Ausführbar als* '**AUTOBOXINGUNBOXINGPROBLEM**'

[7] Der obere Wert lässt sich über die System-Property `java.lang.Integer.IntegerCache.high` ändern. Verschiedene Werte können das Programmverhalten (unerwartet) ändern.

Führt man das Programm AUTOBOXINGUNBOXINGPROBLEM aus, so liefern die Vergleiche mit den Operatoren '>=' und '<=', wie erwartet, jeweils `true`. Der Vergleich mit dem Operator '==' ergibt jedoch (zunächst überraschend) `false`. Wie ist das zu erklären?

Die Operatoren '>=' und '<=' sind nur für primitive Datentypen, nicht aber für `Integer`-Objekte definiert. Für die gezeigten Vergleiche findet zunächst ein Auto-Unboxing statt, um den Vergleich auszuführen. Der Operator '==' ist für Objekte definiert und benötigt kein Auto-Unboxing, sondern arbeitet auf `Integer`-Referenzen. Es findet also ein Referenzvergleich statt, und somit kommt es zur Rückgabe von `false`.

Tipp: Spezialitäten beim Auto-Boxing und bei Methodenaufrufen

Beim Auto-Boxing gibt es neben den bereits kennengelernten Besonderheiten noch zwei weitere Spezialfälle zu betrachten.

Auto-Boxing und Methodensignaturen Um bei einem Methodenaufruf eine passende Methode zu finden, führt die JVM folgende Schritte durch:

1. Test auf exakte Übereinstimmung der Parametertypen, dann auf Subtyp
2. Anpassung durch Widening: `byte` ⇒ `short` ⇒ `int` ⇒ `long` usw.
3. Anpassung durch Auto-Boxing und Auto-Unboxing

Dieses Wissen ist hilfreich, wenn man sich fragt, welche Methoden die JVM für folgenden Sourcecode ausführt:

```
public static void main(final String[] args)
{
    final List<Integer> indexList = new ArrayList<>();
    indexList.add(7);
    indexList.add(8);

    // Auto-Boxing => Integer.valueOf(0) => remove(Object) oder remove(int)?
    final Integer removed = indexList.remove(0);
    // Gemäß Regel 1 => remove(int)
    System.out.println("removed " + removed);

    final Integer value = 8;
    // Auto-Unboxing => 8 => remove(int) ?
    // Subtypbeziehung Integer extends Object => remove(Object)
    System.out.println(indexList.remove(value));
}
```

Listing 5.14 *Ausführbar als* '**AUTOBOXINGANDMETHODSIGNATUREEXAMPLE**'

Für das `int`-Zahlenliteral `0` findet kein Auto-Boxing statt. Gemäß Regel 1 wird die passende Methode `remove(int)` des Typs `List<Integer>` aufgerufen. Für den Aufruf von `remove(value)` mit dem `Integer`-Objekt `value` wird basierend auf Regel 1 und der Subtypbeziehung von `Integer` zur Klasse `Object` die Methode `remove(Object)` ausgeführt. Dadurch kommt es zu folgenden Ausgaben:

```
removed 7
true
```

Auto-Boxing und Arrays Weil Arrays Objekte sind, erfolgt weder Auto-Boxing noch Auto-Unboxing: Deshalb wird beispielsweise ein `int[]` nicht automatisch in ein `Integer[]` umgewandelt. Somit kann eine Methode `processValues(Integer[])` nicht mit einem `int[]` als Eingabe arbeiten. Das Verhalten scheint zunächst unverständlich und wirkt unpraktisch. Nach einiger Überlegung kommt man aber zum Schluss, dass es doch ganz gut ist, um negative Auswirkungen auf den Speicherverbrauch und die Performance zu vermeiden.

Fazit

Wie vorgestellt gibt es einige Merkwürdigkeiten beim Einsatz von Auto-Boxing und Auto-Unboxing. Daher ist beides mit etwas Vorsicht zu genießen. Insbesondere kann es zu einer `NullPointerException` kommen, wenn ein Auto-Unboxing einer `null`-Referenz erfolgt. Um unerwünschte Konvertierungen zu finden, sollte man in Eclipse eine entsprechende Warnung beim Kompilieren aktivieren (vgl. Abschnitt 2.1).

5.2.4 Ausgabe und Verarbeitung von Zahlen

In diesem Abschnitt stelle ich einige nützliche Methoden der Wrapper-Klassen zur Ausgabe und zur Verarbeitung von Zahlen vor.

Ausgabe von Zahlen

Die Objektmethoden `toString()` der Klassen `Integer` und `Long` geben den Wert eines Wrapper-Objekts im Dezimalsystem aus. Ebenso kann man einen beliebigen Wert mithilfe der statischen Methoden `toString(int)` bzw. `toString(long)` dezimal ausgeben. Manchmal soll eine Ausgabe jedoch bezüglich einer anderen Basis erfolgen. Dazu sind in den Klassen `Integer` und `Long` überladene statische `toString()`-Methoden definiert, die die Angabe einer beliebigen Basis erlauben:

```
public static void main(final String[] args)
{
    // Zur Basis 10 - Dezimaldarstellung
    System.out.println("15=" + Integer.toString(15));      // 15=15

    // Zur Basis 2 - Binärdarstellung
    System.out.println("15=" + Integer.toString(15, 2));   // 15=1111

    // Zur Basis 8 - Oktaldarstellung
    System.out.println("15=" + Integer.toString(15, 8));   // 17

    // Zur Basis 16 - Hexadezimaldarstellung
    System.out.println("15=" + Integer.toString(15, 16));  // 15=f
}
```

Listing 5.15 *Ausführbar als* **'NUMBEROUTPUTEXAMPLE'**

Die Ausgabe in einigen Zahlensystemen ist so gebräuchlich, dass dafür spezielle statische Methoden der Klassen `Integer` und `Long` definiert sind. Diese Methoden können

eine binäre, hexadezimale oder oktale Darstellung erzeugen. Dabei gibt es Folgendes zu beachten:

1. Führende Nullen werden grundsätzlich *nicht* mit ausgegeben. Für das Dezimal- und Oktalsystem ist dies nicht weiter störend. Im Binär- und Hexadezimalsystem wäre aufgrund des eher technischen Bezugs ein Auffüllen mit führenden Nullen zum Teil wünschenswert.
2. Es findet eine Vorzeichenerweiterung statt, wodurch es zu einer etwas überraschenden Ausgabe für den Wert `-1` im Hexadezimalsystem kommt. Hier wird ein Implementierungsdetail der Zahlen sichtbar, nämlich die schon besprochene interne Repräsentation der Zahlen als Zweierkomplement:

```
public static void main(final String[] args)
{
    System.out.println( "15=" + Integer.toBinaryString(15) ); // 15=1111
    System.out.println( "15=" + Integer.toOctalString(15) );  // 15=17
    System.out.println( "15=" + Integer.toHexString(15) );    // 15=f
    System.out.println( "255=" + Integer.toHexString(255) );  // 255=ff
    System.out.println( "-1=" + Integer.toHexString(-1) );    // -1=ffffffff
}
```

Listing 5.16 *Ausführbar als* **'NUMBEROUTPUTEXAMPLESPECIAL'**

Konvertierung von Gleitkommazahlen in eine Bitdarstellung

Manchmal kann es nützlich sein, Gleitkommazahlen in eine Bitdarstellung umzuwandeln. Mithilfe der Methode `floatToIntBits(float)` ist es möglich, einen `float`-Wert in einen `int`-Wert mit einer Bitrepräsentation gemäß dem IEEE-754-Format zu transformieren. Aus einer solchen Bitrepräsentation lässt sich mit der Methode `intBitsToFloat(int)` wieder der korrespondierende `float`-Wert ermitteln. Die Klasse `Double` bietet zur Konvertierung analog folgende Methoden:

```
public static long doubleToLongBits(double value)
public static double longBitsToDouble(long bits)
```

Aufgrund des größeren Wertebereichs eines `double` wird hier die Bitrepräsentation jedoch in einem `long`-Wert gespeichert.

Extraktion einzelner Bytes aus einem `int`

Java bietet zur Manipulation auf Bitebene diverse Operationen an: NOT, AND, OR, XOR und Bit-Shifts. Die meisten Operationen arbeiten so, wie man sich das intuitiv vorstellt. Ein bitweiser Links-Shift füllt mit Nullen auf. Im Gegensatz dazu erhält der bitweise Rechts-Shift das Vorzeichen und füllt dazu mit dem jeweiligen Vorzeichenbit auf. Der alternative bitweise Rechts-Shift II füllt mit Nullen auf. Man verliert dadurch die Vorzeicheninformation. Derart veränderte Zahlen sollten nur als Bitmuster, jedoch nicht als Zahl im Zweierkomplement ausgewertet werden. Für viele Bitoperationen ist

diese Einschränkung aber durchaus in Ordnung. In Tabelle 5-2 sind die genannten Bitoperationen (auf 8 Bit gekürzt) dargestellt.

Tabelle 5-2 *Bitoperationen*

Bitmuster	Operation	Operand	Ergebnis	Beschreibung
11001100	~	-/-	00**11**00**11**	NOT
11001100	&	00001111	**00**00**11**00	AND
11001100	|	00001111	11001**11**	OR
11001100	^	00001111	1100**0011**	XOR
11001100	<<	4	1100**0000**	Links-Shift
11001100	>>	4	**1111**1100	Rechts-Shift
11001100	>>>	4	**0000**1100	Rechts-Shift II

Tatsächlich arbeiten die in Tabelle 5-2 gezeigten Operatoren nicht auf dem Typ `byte`, sondern auf dem Typ `int`. Schauen wir uns ein Beispiel an:

```
public static void main(final String[] args)
{
    final int value = 1 | 8 | 256 | 32768;
    System.out.println("Value = " + value);
    System.out.println("Binary Value = " + Integer.toBinaryString(value));

    System.out.println("byte 0 = " + extractByteValue(value));

    System.out.println("value >> 8 = " + (value >> 8));
    System.out.println("byte 1 = " + extractByteValue(value >> 8));
}

public static byte extractByteValue(final int value)
{
    return (byte) (value & 255);
}
```

Listing 5.17 *Ausführbar als* '**EXTRACTBYTEEXAMPLE**'

Man erhält folgende Ausgaben auf der Konsole:

```
Value = 33033
Binary Value = 1000000100001001
byte 0 = 9
value >> 8 = 129
byte 1 = -127
```

Die Verarbeitung startet mit dem Wert 33033, binär '`1000000100001001`'. Schneidet man davon die letzten 8 Bit durch die Operation `& 255` ab, so erhält man den Wert '`00001001`', der der Zahl 9 entspricht. Ein Bit-Shift der binären Darstellung des Werts 33033 um 8 Stellen nach rechts führt zu dem Bitmuster '`10000001`', das als `int` den Wert 129 ergibt. Dieser Wert übersteigt den Wertebereich eines `byte`

und somit wird das Bitmuster als negative Zahl ausgewertet. Dabei gilt die Formel $byteValue = -255 + x - 1$. Hier ergibt sich $-255 + 129 - 1 = -127$ als Wert.

5.3 Stringverarbeitung

Zur Speicherung und Verarbeitung von Zeichenketten existieren in Java die Klassen `String`, `StringBuffer` und `StringBuilder`. Alle drei erfüllen das Read-only-Interface `java.lang.CharSequence`.

Das Interface `CharSequence`

Das Interface `CharSequence` bietet hauptsächlich einen indizierten Zugriff auf einzelne Zeichen vom Typ `char` bzw. auf Zeichenfolgen, die wiederum vom Typ `CharSequence` sind. Dazu sind folgende Methoden deklariert und ergänzend zwei Defaultmethoden sowie eine statische Methode definiert:

```
public interface CharSequence
{
    public char charAt(int index);
    public int length();
    public CharSequence subSequence(int start, int end);
    public String toString();

    public default IntStream chars() { ... }
    public default IntStream codePoints() { ... }
    public static int compare(CharSequence cs1, CharSequence cs2) { ... }
}
```

Das Interface `CharSequence` ermöglicht es also, Zeichenketten indiziert zu verarbeiten, ohne konkretes Wissen über den speziellen Typ zu besitzen. Dadurch kann man Schnittstellen allgemeiner gestalten, sofern nicht spezifische Funktionalitäten der Klassen `String` bzw. `StringBuffer/-Builder` benötigt werden.

Wissenswertes zum Interface `CharSequence` Beachten Sie unbedingt, dass das Interface `CharSequence` keine Aussagen zum Verhalten und den Kontrakten von `equals(Object)` und `hashCode()` macht. Was bedeutet das genauer? Wenn man zwei Instanzen per `equals(Object)` vergleicht, die beide den Typ `CharSequence` besitzen, so ist das Ergebnis des Vergleichs undefiniert. Sind etwa beide Instanzen vom Typ `String`, so ist das Ergebnis bei gleichen textuellen Inhalten `true`. Eine `CharSequence`-Instanz könnte aber auch vom Typ `StringBuffer` sein. Ein Vergleich der Typen `String` und `StringBuffer` liefert immer `false`. Nachfolgendes Programm demonstriert einige Vergleiche:

```
public static void main(final String[] args)
{
    final CharSequence cs1 = new StringBuilder("same");
    final CharSequence cs2 = new StringBuilder("same");
    final CharSequence cs3 = "same";
    final CharSequence cs4 = "same";
    final CharSequence cs5 = new String("same");

    System.out.println(cs1.equals(cs2));  // false
    System.out.println(cs1.equals(cs3));  // false
    System.out.println(cs3.equals(cs4));  // true
    System.out.println(cs3.equals(cs5));  // true
}
```

Listing 5.18 *Ausführbar als* '**CHARSEQUENCEEQUALSEXAMPLE**'

Achtung: `CharSequence` in Collections und Maps

Basierend auf dieser Argumentation sollte man den Typ `CharSequence` niemals als Elemente eines `Set<E>` oder als Schlüssel einer `Map<K,V>` nutzen.

5.3.1 Die Klasse `String`

Die Klasse `java.lang.String` repräsentiert Zeichenfolgen, die bis Java 8 aus Unicode-Zeichen bestehen und ihren Inhalt in Form eines Arrays von `char` speichern. Seit Java 9 wird der Inhalt als `byte[]` modelliert und gemäß einem Encoding ausgewertet (vgl. Abschnitt 5.3.5). Solche Zeichenfolgen kann man entweder durch einen Konstruktoraufruf der Klasse `String` erzeugen oder aber als Zeichenkette in Anführungszeichen, wie dies folgende zwei Zeilen zeigen:

```
final String stringObject = new String("New String Object");
final String stringLiteral = "Stringliteral";
```

Im ersten Fall wird ein ***Stringobjekt*** und im zweiten Fall ein ***Stringliteral*** erzeugt – tatsächlich ist das Argument des Konstruktors im ersten Fall auch ein Literal. Es ist wichtig, zwischen diesen beiden Formen zu unterscheiden. Stringobjekte erhalten durch den Konstruktoraufruf immer eine eigenständige Referenz. Für Stringliterale wird hingegen nur einmalig eine neue `String`-Instanz erzeugt und in einen speziellen Cache, den sogenannten ***Stringliteral-Pool***, eingetragen. Existiert dort für ein Stringliteral bereits ein Eintrag, so wird keine neue `String`-Instanz erzeugt, sondern gleiche Literale »teilen« sich Referenzen auf Stringobjekte aus diesem Stringliteral-Pool. Konkret bedeutet dies: Wird ein Literal mit gleichem Inhalt wie ein bereits vorhandenes erzeugt, so verweist dieses auf dasselbe Objekt.

Gäbe es dieses Caching nicht, so müsste für jedes Stringliteral eine temporäre `String`-Instanz erzeugt und bald danach verworfen werden. Demnach erfolgt das Caching der Stringliterale, um möglichen Performance-Problemen entgegenzuwirken.

Wissenswertes zu Stringliteralen und Stringvergleichen

Die Kenntnis des Unterschieds zwischen einem Referenzvergleich mit dem Operator '==' und einem Inhaltsvergleich per `equals(Object)` bildet die Grundlage zum Verständnis einiger Besonderheiten beim Vergleich von Strings. Durch das Caching von Stringliteralen gibt es Unterschiede zwischen Stringobjekten und Stringliteralen.[8]

Betrachten wir, was passiert, wenn man verschiedene Stringliterale und Stringobjekte miteinander vergleicht. Dazu definieren wir die beiden Konstanten `LITERAL` und `STRINGOBJECT` jeweils mit dem Inhalt `"TEST"`:

```
public static void main(final String[] args)
{
    final String LITERAL = "TEST";
    final String STRINGOBJECT = new String("TEST");

    if (LITERAL == "TEST")
    {
        System.out.println("LITERAL == TEST");
    }
    if (STRINGOBJECT == "TEST")
    {
        System.out.println("STRINGOBJECT == TEST");
    }
    if (STRINGOBJECT == LITERAL)
    {
        System.out.println("STRINGOBJECT == LITERAL");
    }
    if (STRINGOBJECT.equals(LITERAL))
    {
        System.out.println("STRINGOBJECT equals LITERAL");
    }
}
```

Listing 5.19 *Ausführbar als* **'STRINGCOMPARELITERALEXAMPLE'**

Es kommt zu folgender Ausgabe auf der Konsole:

```
LITERAL == TEST
STRINGOBJECT equals LITERAL
```

Die Ausgabe verdeutlicht, dass tatsächlich ein Caching von Stringliteralen erfolgt. Insbesondere auf den positiven Ausgang des ersten Vergleichs darf man sich (eigentlich) nicht verlassen, denn er basiert rein auf dem Caching. Verwenden wir explizit neue `String`-Instanzen, so ist nur noch der `equals(Object)`-Vergleich erfolgreich.

Überraschungen bei Berechnungen in Stringausgaben

Die Klasse `String` bietet zum einfachen Verknüpfen von Texten einen »überladenen« Operator '+' an. Das ist sehr praktisch und erhöht die Lesbarkeit bei der Aufbereitung von textuellen Ausgaben. Allerdings ist bei Berechnungen die Reihenfolge der Operationen entscheidend. Betrachten wir dies an einem Beispiel:

[8] Das erinnert an Probleme des Cachings bei den Wrapper-Klassen (vgl. Abschnitt 5.2.2).

```
System.out.println("Test" + 1 + 2);      // 'Test12'
System.out.println(1 + 2 + "Test");      // '3Test'
```

Die Ausgabe ist sehr unterschiedlich. Wie kommt das? Weil die Auswertung immer von links nach rechts erfolgt, kommt es dabei auf den Typ des linken Operanden an. Beginnen wir mit der einfachen Variante: Bei der zweiten Ausgabe findet zunächst eine Addition von `int`-Werten statt und erst anschließend erfolgt eine Umwandlung in ein Stringobjekt durch Aufruf der statischen Methode `valueOf()`. Wenn der linke Operand bereits vom Typ `String` ist, so findet in jedem Fall eine Stringkonkatenation statt. Dadurch werden scheinbare Additionen von Zahlen nicht als solche ausgeführt, sondern die Zahlen werden zunächst in einen String gewandelt und dann an den ursprünglichen Text angehängt. Folgende Aufschlüsselungen der Abläufe verdeutlichen das Gesagte:

```
System.out.println("Test" + 1 + 2);
                        |   |
                  String + int => String + String.valueOf(int)
                           |
                      String + int => String + String.valueOf(int)
=> "Test12"

System.out.println(1 + 2 + "Test");
                    |   |
                int + int => int
                        |
                    int + String => String.valueOf(int) + String
=> "3Test"
```

Dieses Beispiel scheint konstruiert zu sein. Tatsächlich treten solche Anweisungen in der Praxis aber doch häufiger durch Flüchtigkeitsfehler auf. Das kann überraschende und irreführende Ausgaben zur Folge haben, wie wir es nachfolgend für die berechnete Länge einer Nachricht sehen:

```
log.error("info '" + msg + "' too long! LENGTH: " + tele.size() + msg.length);
=> "... LENGTH: 19240"

log.error("info '" + msg + "' too long! LENGTH: " + (tele.size() + msg.length));
=> "... LENGTH: 259"
```

Ein weiteres Beispiel ist die Berechnung und Ausgabe der Differenz von zwei Zeitpunkten `time0` und `time1`:

```
// System.out.println("Copy took: " + time1-time0 + " ms"); // Compile-Error
System.out.println("Copy took: " + (time1-time0) + " ms");
```

Die obere Anweisung kompiliert nicht, da der Operator '–' auf Strings nicht definiert ist. Durch diese Beispiel wird deutlich, dass fehlerhafte Berechnungen schnell auftreten können.

Besonderheiten beim Einsatz von `toLowerCase()` und `toUpperCase()`

Die Methoden `toLowerCase()` und `toUpperCase()` der Klasse `String` dienen dazu, einen gegebenen String in eine einheitliche Schreibweise in Klein- bzw. Großschreibung zu konvertieren. Das scheint trivial. Ist es aber nicht in jedem Fall. Mathematisch ausgedrückt wird keine bijektive Abbildung realisiert: Es gibt demnach nicht immer eine eineindeutige Hin- und Zurück-Transformation der Strings. Intuitiv ist dies klar, weil etwa "`TeST`" und "`TesT`" jeweils in Kleinschreibung als "`test`" dargestellt werden. Es gibt aber subtilere Fallstricke beim Konvertieren von Groß- in Kleinschreibung und umgekehrt. Betrachten wir anhand des Worts »Fußball« einige Konvertierungsschritte:

```
Original: 'Fußball' toLower: 'fußball' toUpper: 'FUSSBALL' toLower: 'fussball'
```

Durch Hintereinanderausführung von Konvertierungen ist aus dem Text "`Fußball`" der Text "`fussball`" geworden, der nicht eindeutig auf "`Fußball`" rückabbildbar ist.

Unveränderlichkeit von Strings

Strings sind in Java als unveränderliche Objekte realisiert. Dadurch kommt es leicht zu Flüchtigkeitsfehlern in der Anwendung. Betrachten wir dies an einem Beispiel:

```
public static void main(final String[] args)
{
    String test = "Heute";
    test.concat(" ist ein schöner Tag!");

    System.out.println(test);
}
```

Listing 5.20 *Ausführbar als* **'STRINGIMMUTABILITYEXAMPLE'**

Intuitiv erwartet man die Ausgabe "`Heute ist ein schöner Tag!`". Die Ausgabe ist jedoch "`Heute`". Die Erklärung ist einfach: ***Stringverändernde Methoden liefern immer eine neue***[9] ***Stringinstanz zurück, die die Änderungen enthält.*** Ohne eine Zuweisung an eine Variable geht ansonsten die neu erzeugte Stringinstanz und damit die Modifikation verloren. Folgende gebräuchliche Operationen erzeugen neue Instanzen:

- `+, +=`
- `concat(), replace(), subString()`
- `toUpperCase(), toLowerCase()`

Dabei ist der Java-Compiler allerdings einigermaßen schlau. Folgendes Konstrukt wird direkt zu einem Stringliteral zusammengefasst, bevor es mit dem Stringliteral-Pool abgeglichen wird:

```
final String multipleLiterals = "Dies " + "ist " + "ein " + "String";
```

[9] Mit der Ausnahme, dass man bereits genutzte Teile aus Stringliteralen miteinander verknüpft, etwa `"A" + "BC"`, wenn das Stringliteral `"ABC"` schon vorher im Sourcecode vorkam.

Auch für folgendes Konstrukt besitzt der Compiler interne Optimierungen. Zunächst greift die gerade vorgestellte automatische Zusammenfassung von Stringliteralen. Aber was ist mit dem '+' für die String-Objekte?

```
public static void internalProcessingStringObjects()
{
    final String multipleObjects = new String("Dies " + "ist ") +
                                   new String("ein " + "String");

    System.out.println(multipleObjects);
}
```

Tatsächlich resultiert dies nicht in einer Addition der beiden String-Objekte, sondern der Compiler ist schlauer und optimiert eine '+'-Konkatenation von Stringobjekten. Dabei kommt bei komplexeren Ausdrücken die im folgenden Abschnitt vorgestellten Klasse `StringBuilder` zum Einsatz, teilweise nutzen neuere JVMs sogar weitergehende Optimierungen.

5.3.2 Die Klassen `StringBuffer` und `StringBuilder`

Die Veränderlichkeit von textuellen Daten ist wünschenswert, aber für die Klasse `String` aufgrund deren Unveränderlichkeit nicht gegeben. Für Stringmanipulationen existieren die Klassen `StringBuffer` und `StringBuilder`. Beide besitzen ein identisches API mit dem Unterschied, dass die Methoden der Klasse `StringBuffer` synchronisiert sind. Das bedeutet, dass die Methoden jeweils nur von einem Thread zur Zeit aufgerufen werden können (vgl. Kapitel 10.2.1). Das gilt für die Methoden der Klasse `StringBuilder` nicht, was nicht stört, solange darauf nur ein Thread arbeitet.

Stringkonkatenation

Ein häufig anzutreffender Tipp ist, diese Klassen statt des Operators '+' zur Konstruktion umfangreicher Textausgaben einzusetzen. Tatsächlich muss man das Ganze etwas differenzierter betrachten. Einfache Stringkonkatenationen mit dem Operator '+', wie die aus dem obigen Beispiel der Methode `internalProcessingStringObjects()`, werden bei der Kompilierung bereits automatisch stark und noch besser als mit einem `StringBuilder` optimiert. Nur wenn sich in Verknüpfungen variable Anteile finden, wird das Ganze durch den Einsatz eines `StringBuilder`-Objekts ersetzt.

Betrachten wir ein kurzes Beispiel, wobei die Methode `calcMiddle()` den variablen Anteil darstellt:

```
public static void internalProcessingPlus()
{
    final String example = "text start" + calcMiddle() + "text end";
}

private static String calcMiddle()
{
    return "...";
}
```

Der resultierende Bytecode ist identisch mit der folgenden Umsetzung, wie dies im folgenden Praxistipp anhand des erzeugten Bytecodes nachgewiesen wird.

```
public static void internalProcessingSB()
{
    final String example = new StringBuilder("text start").
                           append(calcMiddle()).
                           append("text end").toString();
}
```

Damit ist nachvollziehbar, warum es bei Stringverknüpfungen in der Regel keinen Sinn macht, auf die gut lesbare Schreibweise der Konkatenation durch den Operator '+' zu verzichten. Allerdings gibt es ein Einsatzgebiet, wo die explizite Verwendung eines `StringBuilder`s sinnvoll sein kann. Dies ist immer dann der Fall, wenn sehr häufig mit dem Operator '+=' gearbeitet wird.

IDIOM: PERFORMANTE STRINGKONKATENATION

Für erwiesenermaßen Performance-kritische Abschnitte sollte man Stringkonkatenationen mithilfe eines `StringBuilder`-Objekts explizit nach folgendem Idiom realisieren:

```
String example = "text start";
final StringBuilder sb = new StringBuilder(example);
for (int i = 0; i < n; i++)
{
    sb.append(getValue(n));
}
example = sb.append("text end").toString();
```

Auf diese Weise vermeidet man sämtliche kurzlebigen temporären Objekte, die durch eine automatische Transformation entstanden wären.

Tipp: Class-File-Disassembler-Werkzeug `javap`

Um die Details der Verarbeitung und die der Umsetzung in Bytecode sichtbar zu machen, kann man sich des Tools `javap` bedienen. Ein Aufruf von `javap -v <ClassFileName>` (Achtung: ohne Endung `.class`) zeigt unter anderem die erzeugten Konstanten sowie die Bytecode-Anweisungen der Methoden. Nutzt man `javap` zur Untersuchung der obigen Methoden `internalProcessingPlus()` und `internalProcessingSB()` durch Aufruf von `javap -v bin/main/ch05_javagrundlagen/stringhandling/StringInternalsExample`, so sieht man, dass der generierte Bytecode identisch ist. Somit kann man bevorzugt die lesbare Variante mit '+' nutzen und muss nicht im Sourcecode mit `StringBuilder` arbeiten.

Fallstrick bei `equals()` von `StringBuffer/-Builder`

Bei der Aufbereitung von Stringrepräsentationen mithilfe der Klassen `StringBuffer` und `StringBuilder` benötigt man ab und an eine Prüfung auf inhaltliche Gleichheit. Für Strings ist bekannt, dass man per `==` die Referenzen und mit `equals(Object)` deren Inhalt vergleicht, aber wie verhält es sich für `StringBuffer` und `StringBuilder`? Intuitiv könnte man annehmen, dass deren `equals(Object)`-Methode auf inhaltliche Gleichheit prüft, wie dies für die Klasse `String` der Fall ist. Dann würde man etwa Folgendes schreiben:

```
final StringBuilder sb1 = new StringBuilder("Text");
final StringBuilder sb2 = new StringBuilder("Text");

// ACHTUNG: Falsche Lösung
final boolean equalsOnSB = sb1.equals(sb2);
System.out.println("sb equals: " + equalsOnSB);
```

Wenn man das Programm ausführt, liefert der Vergleich den Wert `false`, da hier (überraschenderweise) ein Referenzvergleich durchgeführt wird. Somit kann man `StringBuffer`- bzw. `StringBuilder`-Objekte nicht sinnvoll per `equals(Object)` vergleichen, sondern muss dafür folgendes Idiom nutzen.

IDIOM: PRÜFUNG AUF INHALTLICHE GLEICHHEIT

Wenn man den Inhalt zweier `StringBuffer`- bzw. `StringBuilder`-Objekte `sb1` und `sb2` miteinander vergleichen möchte, so kann man das wie folgt realisieren:

```
final boolean contentIsEqual = sb1.toString().equals(sb2.toString())
```

Dabei entstehen allerdings zwei Stringobjekte. Falls ein Objekt vom Typ `String` ist, kann man die Methode `contentEquals(CharSequence)` aus der Klasse `String` für die Prüfung auf inhaltliche Gleichheit verwenden:

```
final boolean stringContentIsEqual = string.contentEquals(charSequence);
```

Weitere Funktionalität und Vergleich mit der Klasse `String`

Mitunter praktisch ist, dass die beiden Klassen `StringBuffer` und `StringBuilder` Löschoperationen besitzen: Über die Methoden `deleteCharAt()` und `delete()` können Zeichen aus einer Stringrepräsentation entfernt werden. Analoge Methoden namens `insert()` erlauben es, Zeichen einzufügen. Das stellt einen Vorteil gegenüber Stringobjekten dar, wo dies nicht möglich ist. Allerdings haben sowohl der `StringBuilder` als auch der `StringBuffer` einen Nachteil, denn sie bieten leider diverse Methoden nicht, etwa `toLowerCase()` und `toUpperCase()` – dafür jedoch die (weniger häufig benötigte) Methode `reverse()`, die den Inhalt in umgekehrter Reihenfolge liefert. Tabelle 5-3 zeigt einige wichtige Methoden der Klasse `String` und eine

Abbildung auf diejenigen des `StringBuffer`s bzw. `StringBuilder`s. Es existieren weitere Unterschiede und nicht dargestellte Methoden.

Tabelle 5-3 *Abbildung der Stringmethoden*

String	StringBuffer/-Builder
`+, +=, concat()`	`append()`
`replace(), subString()`	`replace(), subString()`
`indexOf(), startsWith()`	`indexOf()`
`endsWith()`	`lastIndexOf()`
- ***keine Methode vorhanden!*** -	`reverse()`
- ***keine Methode vorhanden!*** -	`insert()`
- ***keine Methode vorhanden!*** -	`delete(), deleteCharAt()`
`toUpperCase()/toLowerCase()`	- ***keine Methode vorhanden!*** -

5.3.3 Ausgaben mit `format()` und `printf()`

Immer wieder müssen Zahlen, Datumsangaben und Texte formatiert werden. Die im Anschluss vorgestellten Methoden `String.format()` und `System.out.printf()` ermöglichen eine Ausgabe, wie sie unter C mit `printf()` erfolgt. Für viele Aufgaben reicht das bereits aus. Eine objektorientierte und flexiblere Realisierung bieten die `Format`-Klassen, die in Abschnitt 13.1.4 genauer vorgestellt werden.

Ausgaben mit `String.format()` aufbereiten

Mit der statischen Methode `format(String, Object[])` der Klasse `String` kann man Zeichenketten nach einem speziellen Muster mit Platzhaltern formatieren.

```
public static void main(final String[] args)
{
    final Object[] sampleArgs = { "pi", 3.1415, 12345 };
    final String str = String.format("%S='%2.5f' Zahl='%,d'", sampleArgs);

    System.out.println(str); // PI='3,14150' Zahl='12.345'

    System.out.println(String.format("%03x %X %b", 7, 161, true));
}
```

Listing 5.21 *Ausführbar als* '**STRINGFORMATEXAMPLE**'

Es kommt zur Ausgabe von

```
PI='3.14150' Zahl='12?345'
007 A1 true
```

Daran sieht man, dass die Platzhalter bei der Ausgabe durch übergebene Werte ersetzt werden. Nachfolgende Aufzählung nennt mögliche Platzhalter (vgl. die API-Dokumentation):

- '`%s`' steht für eine Ausgabe eines Strings,
  '`%S`' wandelt den Wert in Großbuchstaben um.
- '`%d`' dient zur Ausgabe von dezimalen Zahlen,
  '`%,d`' gibt dezimale Zahlen mit Tausenderpunkten aus.
- '`%f`' bzw. '`%m.nf`' dient zur Ausgabe von Gleitkommazahlen, wobei `m` und `n` die Anzahl der Vor- bzw. Nachkommastellen festlegen. Ohne Angabe werden immer die benötigte Anzahl Vorkommastellen und sechs Nachkommastellen ausgegeben.
- '`%x`' bzw. '`%X`' gibt hexadezimale Zahlen aus.
- '`%b`' steht für eine Ausgabe von booleschen Werten.

Nach '`%`' kann man noch ein Füllzeichen und eine Länge angeben, sodass etwa '`%08x`' bedeutet, eine Hexadezimalzahl achtstellig ausgeben und vorne mit Nullen auffüllen.

Ausgaben mit `System.out.printf()`

Die Methode `printf(String, Object[])` gibt es in den Klassen `PrintWriter` und `PrintStream` und steht somit auch im Standardausgabestream `System.out` zur Verfügung. Dadurch wird eine formatierte Ausgabe direkt möglich, ohne zunächst ein Stringobjekt über `format(String, Object[])` aufzubereiten und anschließend über `System.out.println()` auszugeben. Folgendes Listing zeigt beide Varianten:

```
public static void main(final String[] args)
{
    final String str = String.format("Hi %s. Es ist %d Uhr.", "Mike", 12);
    System.out.println(str);   // Hi Mike. Es ist 12 Uhr.

    System.out.printf("Hi %s. Es ist %d Uhr.", "Mike", 12);
}
```

Listing 5.22 *Ausführbar als* '**STRINGFORMATVARARGSEXAMPLE**'

In beiden Fällen kommt es erwartungsgemäß zu der Ausgabe von »`Hi Mike. Es ist 12 Uhr.`«.

5.3.4 Die Methode `split()` und reguläre Ausdrücke

Einfache Zerlegungen von Strings lassen sich mithilfe der Klasse `java.util.StringTokenizer` bewerkstelligen. In diesem Abschnitt schauen wir uns schrittweise komplexere Zerlegungen basierend auf regulären Ausdrücken und der Methode `split(String)` der Klasse `String` an. Diese teilt einen gegebenen String in mehrere Teilstrings. Sie arbeitet dabei mit frei wählbaren Trennzeichen(folgen), die als regulärer Ausdruck angegeben werden. Für das Verständnis regulärer Ausdrücke wollen wir folgende informelle Definition verwenden:

Definition 5.1 *Unter einem **regulären Ausdruck** versteht man eine Zeichenkette, die zur Beschreibung von anderen Zeichenketten dient. Im einfachsten Fall kann dies eine Folge von Zeichen sein. Die Darstellung gewisser Abfolgen oder Alternativen geschieht durch Spezial- oder Metazeichen, etwa [,], -, + usw. Seien a, b, c, d und e Zeichen, so gelten folgende einfache Regeln zur Konstruktion komplexerer regulärer Ausdrücke:*

Ausdruck	Bedeutung
a	**Einzelnes Zeichen**: Das Zeichen 'a' ist erlaubt.
a+	**Wiederholung**: Ein oder mehrere 'a' sind erlaubt.
[ace]	**Zeichenauswahl**: Eines der Zeichen 'a', 'c' oder 'e' ist erlaubt.
[b-d]	**Bereichsdefinition**: Eines der Zeichen aus dem Bereich von 'b' bis 'd' ist erlaubt.
a[c-e]	**Konkatenation**: Eine Zeichenkette, die mit einem 'a' beginnt und danach eines der Zeichen 'c', 'd' oder 'e' enthält, ist erlaubt.

Im Folgenden werde ich mit einem Beispiel einer einfachen Auswertung von Texten beginnen und dort schrittweise weitere Anforderungen integrieren.

Einfache Zeichenfolgen als regulärer Ausdruck

Nehmen wir an, es wären verschiedene Werte durch die Zeichenfolge `"#-#"` voneinander getrennt. Im folgenden Beispiel nutzen wir einen einfachen regulären Ausdruck, der exakt dieser Zeichenfolge entspricht und in der Konstanten `delimiter` definiert ist. Es wird die Eingabe `input` durch Aufruf der Methode `split(String)` aufgespalten:

```
public static void main(final String[] args)
{
    final String input = "#-# Wert1 #-# Wert2 #-# Wert3";
    final String delimiter = "#-#";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens); // Tokens = '[, Wert1 , Wert2 , Wert3]'
}

private static void printTokens(final String[] tokens)
{
    System.out.print("Tokens = '" + Arrays.asList(tokens) + "'");
}
```

Listing 5.23 *Ausführbar als* **'REGEXEXAMPLE'**

Als Ausgabe erhalten wir:

```
Tokens = '[,  Wert1 ,  Wert2 ,  Wert3]'
```

Nur bei genauerer Betrachtung erkennt man, dass die Tokens teilweise Leerzeichen enthalten. Wir verbessern daher die Ausgaberoutine `printTokens(String[])` so, dass einzelne Tokens in Anführungszeichen eingeschlossen werden:

```
static void printTokens(final String[] tokens)
{
    System.out.print("Tokens = '[");
    for (int i = 0; i < tokens.length; i++)
    {
        System.out.print("'" + tokens[i] + "'");
        // Komma anfügen, wenn noch nicht letztes Element
        if (i != tokens.length - 1)
            System.out.print(", ");
    }
    System.out.println("]'");
}
```

Listing 5.24 *Ausführbar als* '**REGEXEXAMPLEIMPROVEDPRINT**'

Die Ausgabe in Hochkommata einzuschließen haben wir bereits in Abschnitt 5.1.1 für Stringausgaben kennengelernt. Auch hier werden dadurch Resultate klarer erkennbar:

```
Tokens = '['', ' Wert1 ', ' Wert2 ', ' Wert3']'
```

Nun fällt auf, dass nicht nur die Tokens Leerzeichen enthalten, sondern dass insbesondere auch die Ausgabe mit einem Leerstring beginnt. Dies ist dadurch begründet, dass das erste Vorkommen eines Trennzeichens an Position 0 liegt. Das Token davor hat demnach die Länge 0. Bei der Angabe der Trennzeichen muss man ebenfalls aufmerksam sein: Leerzeichen tragen dort semantische Bedeutung.

Varianten in regulären Ausdrücken

Ein Erweiterungswunsch könnte sein, Trennzeichen mit einer laufenden Nummer zu versehen. Die Trenner wären dann die Zeichenfolgen "#1#", "#2#" usw. Da wir bisher lediglich auf Zeichenebene vergleichen, müssen wir alle erlaubten Muster in der Art "#1#", "#2#" usw. angeben und dazu eine ***ODER-Beziehung*** nutzen. Diese ***wird in regulären Ausdrücken über das Zeichen '|' dargestellt***. Der reguläre Ausdruck lautet damit:

```
final String delimiter = "#1#|#2#|#3#";
```

Je mehr der invariante Anteil des Musters dominiert, desto umfangreicher wird die Zeichenfolge zur Beschreibung der Trenner. Besser ist es, nur die Varianzen darzustellen. Schnell kommt der Wunsch auf, alle Ziffern von '0' bis '9' als Trennzeichen verwenden zu dürfen. Dies kann explizit durch Angabe aller Ziffern geschehen, etwa wie folgt:

```
public static void main(final String[] args)
{
    final String input = "#1# Wert1 #5# Wert5 #7# Wert7";
    final String delimiter = "#(0|1|2|3|4|5|6|7|8|9)#";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens); // Tokens = '[", ' Wert1 ', ' Wert5 ', ' Wert7']'
}
```

Listing 5.25 *Ausführbar als* '**REGEXEXAMPLEVARIANTS**'

Bereichsangaben und Wiederholungen in regulären Ausdrücken

Eleganter kann man das Ganze durch die Angabe von Wertebereichen schreiben, die die Notation `[Startwert-Endwert]` verwendet:

```
final String delimiter = "#[0-9]#";
```

Wahrscheinlich sollen die erlaubten Zahlen nicht nur einstellig bleiben. Eine Wiederholung von Zeichen wird über die Notation '*' (0 bis n Vorkommen) bzw. '+' (1 bis n Vorkommen) ausgedrückt. Die Angabe der Vorkommen (Multiplizität) lässt sich für beliebige Zeichen und Ausdrücke, hier im Speziellen für die '#'-Zeichen, anwenden. Wir können die Trenner nun flexibel gestalten und fordern lediglich, dass ein Trenner immer mit mindestens einem '#'-Zeichen beginnt und endet sowie mindestens eine Ziffer und optional weitere enthält. Damit ergibt sich der folgende reguläre Ausdruck:

```
public static void main(final String[] args)
{
    final String input = "#27# Wert27 ##228## Wert228 #3# Wert3";
    final String delimiter = "#+[0-9]*#+";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens); // Tokens = '[", ' Wert27 ', ' Wert228 ', ' Wert3']'
}
```

Listing 5.26 *Ausführbar als* **'REGEXEXAMPLERANGES'**

Spezialzeichen in regulären Ausdrücken

Um Schreibarbeit in regulären Ausdrücken zu vermeiden, existieren einige vordefinierte Abkürzungen als Hilfe. Beispielsweise werden beliebige Ziffern über die Angabe von "`\d`" repräsentiert. Der zuvor hergeleitete Ausdruck lässt sich damit weiter verdichten, allerdings muss das Backslash-Zeichen in Strings speziell notiert werden (man spricht auch von Escapen), hier durch "`\\d`":

```
public static void main(final String[] args)
{
    final String input = "#27# Wert27 ##228## Wert228 #3# Wert3";
    final String delimiter2 = "#+\\d+#+";

    final String[] tokens2 = input.split(delimiter2);
    printTokens(tokens2); // Tokens = '[", ' Wert27 ', ' Wert228 ', ' Wert3']'
}
```

Listing 5.27 *Ausführbar als* **'REGEXEXAMPLERANGES2'**

Man ahnt, dass das Muster des regulären Ausdrucks recht schnell unübersichtlich bzw. kryptisch werden kann. Dies gilt vor allem, wenn man nach diesen Spezialzeichen suchen möchte. Zur Angabe als String in Java ist dann ein weiteres Escapen notwendig. Einige gebräuchliche Spezialzeichen sind in Tabelle 5-4 aufgelistet.

Tabelle 5-4 *Spezialzeichen in regulären Ausdrücken*

Spezialzeichen	Bedeutung
`.`	Platzhalter für ein beliebiges Zeichen
`\d`	Platzhalter für eine beliebige Ziffer
`\w`	Platzhalter für einen Buchstaben oder eine Ziffer
`\s`	Platzhalter für Leerzeichen, Tabulatoren und Zeilenumbrüche
`\n`	Platzhalter für einen Zeilenumbruch
`*`	Angabe einer Wiederholung von 0 bis n Vorkommen
`+`	Angabe einer Wiederholung von 1 bis n Vorkommen
`{n}`	Wiederholung von genau n Vorkommen
`?`	Erlaubt 0 bis 1 Vorkommen

Praxisbeispiel Newsticker

Ein aus der Praxis stammendes Beispiel für den Einsatz spezieller Trennzeichen ist ein Newsticker, dessen Nachrichten durch die Zeichenfolgen "+++" voneinander separiert sind. Möchte man die einzelnen News-Texte ermitteln, so könnte man intuitiv auf folgende Realisierung kommen:

```
final String input = "News: First News +++ Second News +++ End";
final String delimiter = "+++";
// Exception in thread »main« java.util.regex.PatternSyntaxException:
// Dangling meta character '+' near index 0
```

Diese Umsetzung führt jedoch zur Laufzeit zu einer `java.util.regex.PatternSyntaxException`, ausgelöst dadurch, dass hier ein Spezialzeichen mit Metainformation als Trennzeichen genutzt wird. Das löst man durch Escapen wie folgt:

```
public static void main(final String[] args)
{
    final String input = "News: First News +++ Second News +++ End";
    final String delimiter = "\\+\\+\\+";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens);
}
```

Listing 5.28 *Ausführbar als* **'REGEXEXAMPLENEWSTICKER'**

Das Programm REGEXEXAMPLENEWSTICKER erzeugt folgende Ausgabe:

```
Tokens = '['News: First News ', ' Second News ', ' End']'
```

5.3.5 Optimierung bei Strings in JDK 9

Die Klasse `String` dient bekanntermaßen der Modellierung von Zeichenketten. Um problemlos verschiedene Zeichensätze bzw. Zeichencodierungen wie ISO-8859-1 oder UTF-8 unterstützen zu können, verwendet die Klasse `String` bis einschließlich JDK 8 als interne Repräsentation der Zeichenkette ein `char[]`. Beobachtungen und Performance-Messungen aus vielen Jahren zeigen aber, dass `String`s und `char[]` in diversen Java-Applikationen oftmals die Topverbraucher bezüglich Speicher sind. Zudem werden häufig nur Codierungen wie ASCII oder ISO-8859-1 genutzt, die sich durch lediglich ein Byte beschreiben lassen, wodurch pro Zeichen das andere Byte eines `char` unnütz Speicher belegt. Unter dem Namen »Compact Strings« wurde in JDK 9 eine Erweiterung realisiert, die dieses Problem adressiert.

Seit JDK 9 speichert die Klasse `String` die Zeichenkette statt in einem `char[]` in einem `byte[]` namens `value`. Ergänzend wird die Zeichencodierung als Attribut `coder` repräsentiert.

Besonders erwähnenswert ist, dass diese Optimierung für den nutzenden Entwickler transparent ist, d. h., das nach außen sichtbare und angebotene Interface der Klasse `String` ändert sich nicht. Trotzdem lassen sich für – bezogen ausschließlich auf Strings codiert in ISO-8859-1 (Latin 1) – fast 50 % Speicherplatz einsparen. Für die gesamte Applikation sind die Einsparungen geringer, aber vielfach immer noch deutlich spürbar. Zudem wirkt sich diese Optimierung selbst dann aus, wenn man seine älteren Programme mit JDK 9 oder neuer startet – es ist nicht einmal eine Neukompilierung nötig.

5.3.6 Neue Methoden in der Klasse `String` in JDK 11

Die Klasse `String` existiert seit JDK 1.0 und hat zwischenzeitlich nur wenige API-Änderungen erfahren. Mit Java 11 wurden folgende Methoden neu eingeführt:

- `isBlank()` – Für Strings war es bisher mühsam oder nur mithilfe von externen Bibliotheken möglich, zu prüfen, ob diese ausschließlich Whitespaces enthalten. Als Abhilfe existiert nun die Methode `isBlank()`, die sich auf `Character.isWhitespace(int)` abstützt.
- `strip()`, `stripLeading()` und `stripTrailing()` – Mit der Methode `strip()` lassen sich aus einem String führende und nachfolgende Leerzeichen (Whitespaces) entfernen. Gibt es dazu nicht schon die Methode `trim()`? Eigentlich ja, aber `trim()` nutzt eine leicht abweichende Definition von Whitespace. Die Methode `strip()` stützt sich – wie `isBlank()` – auf `Character.isWhitespace(int)` ab. Gleiches gilt auch für die beiden Methoden `stripLeading()` und `stripTrailing()`. Erstere entfernt Whitespace nur am Anfang, letztere nur am Ende eines Strings.
- `lines()` – Zerteilt einen String in einzelne Zeilen, sofern dort Zeilenumbrüche vorkommen.
- `repeat(int)` – Wiederholt einen String gemäß der angegebenen Anzahl, also n-mal.

Für die letzten beiden schauen wir uns zum besseren Verständnis im Anschluss einfache Beispiele an.

Die Methode `lines()`

Beim Verarbeiten von Daten aus Dateien müssen des Öfteren Informationen in einzelne Zeilen aufgebrochen werden. Dazu gibt es etwa die Methode `Files.lines(Path)`. Ist die Datenquelle allerdings schon ein String, existierte diese Funktionalität bis JDK 11 noch nicht. Mittlerweile finden wir in der Klasse `String` die Methode `lines()`, die einen `Stream<String>` zurückliefert und wie folgt genutzt werden kann:

```
private static void linesExample()
{
    final String exampleText = "1 This is a\n2 multi line\n" +
                               "3 text with\n4 four lines!";

    final Stream<String> lines = exampleText.lines();
    lines.forEach(System.out::println);
}
```

Diese Methode unterteilt den mehrzeiligen String in vier einzelne Bestandteile und gibt somit Folgendes aus:

```
1 This is a
2 multi line
3 text with
4 four lines!
```

Die Methode `repeat(int)`

Immer mal wieder steht man vor der Aufgabe, einen String mehrmals aneinanderzureihen, also einen bestehenden String n-mal zu wiederholen. Dazu waren bislang eigene Hilfsmethoden oder solche aus externen Bibliotheken nötig. Mit Java 11 kann man stattdessen die Methode `repeat(int)` nutzen, wie es folgendes Beispiel zeigt:

```
private static void repeatExample()
{
    final String star = "*";
    System.out.println(star.repeat(30));

    final String delimeter = " -*- ";
    System.out.println(delimeter.repeat(6));
}
```

Die obige Methode produziert diese Ausgaben:

```
******************************
 -*-  -*-  -*-  -*-  -*-  -*-
```

Tipp: Spezialfälle

Die beiden zuvor gezeigten Beispiele wirken ganz natürlich. Einige Dinge könnte man sich noch fragen unter anderem folgende:

1. Was passiert, wenn man den Wert 0 übergibt? Dann entsteht ein leerer String.
2. Was passiert, wenn man den Wert -1 oder `Integer.MAX_VALUE` übergibt? In beiden Fällen wird eine Exception ausgelöst.

5.4 Varianten innerer Klassen

In Java sind innere Klassen ein verbreitetes Sprachmittel. Sie sind den gewöhnlichen Klassen ähnlich, jedoch innerhalb von Klassen (oder sogar Methoden) definiert. Mithilfe innerer Klassen lassen sich einige Entwurfsprobleme, etwa die Definition von nur innerhalb einer Klasse sichtbaren Hilfsklassen oder Datencontainerklassen, elegant lösen.

```
public class OuterClass
{
    public class InnerClass
    {
        // ...
    }

    public static class StaticInnerClass
    {
        // ...
    }
}
```

Wie im Listing gezeigt, bietet Java zwei verschiedene Varianten innerer Klassen: *normale* und *statische innere* Klassen. Diese stellen häufig Funktionalität bereit, die lediglich für die äußere Klasse von Interesse ist. Bei Bedarf können innere Klassen jedoch auch von anderen Klassen benutzt werden.

Die Sichtbarkeit einer äußeren Klasse wirkt sich auf dort definierte innere Klassen aus. Im Gegensatz zu äußeren Klassen, die nur die Sichtbarkeiten `public` und Package-private erlauben, sind für innere Klassen alle Sichtbarkeiten zulässig. Wenn die äußere Klasse nur Package-private definiert ist, sind innere Klassen für Package-fremde Klassen nicht sichtbar, selbst wenn die inneren Klassen `public` sind. Ist die äußere Klasse allerdings `public`, so gelten für die Sichtbarkeit die bekannten Regeln: `private` und Package-private definierte innere Klassen sind nur im Package selbst sichtbar. Öffentliche innere Klassen sind für alle anderen Klassen sichtbar. Die Sichtbarkeit `protected` ermöglicht den Zugriff für von der äußeren Klasse abgeleitete Klassen und für Klassen aus demselben Package.

»Normale« innere Klassen

Innere Klassen besitzen eine implizite Referenz auf eine Instanz der äußeren Klasse und können dadurch auf deren Elemente zugreifen – sogar auf die privaten. ***Allerdings folgt daraus auch, dass immer ein Objekt der umgebenden Klasse existieren muss, um ein Objekt einer inneren Klasse zu erzeugen***. Dadurch ergibt sich eine etwas merkwürdige Syntax zur Objekterzeugung:

```
// Variante 1
final OuterClass.InnerClass inner = new OuterClass().new InnerClass();

// Variante 2
final OuterClass outer = new OuterClass();
final OuterClass.InnerClass inner2 = outer.new InnerClass();
```

Statische innere Klassen

Wenn äußere Klassen lediglich erzeugt werden, um Zugriff auf innere Klassen zu bieten, so ist dies – wie eingangs dieses Unterkapitels gezeigt – eher unnatürlich. Das gilt insbesondere dann, wenn es sich bei den inneren Klassen nur um Hilfsklassen oder Datencontainer handelt. Oftmals dienen sie als semantische Strukturierung und sind in der Regel unabhängig von der äußeren Klasse. Dafür existiert das Sprachmittel der statischen inneren Klassen. Das illustriert die Klasse `TripleVO`. Diese realisiert einen Datencontainer, der die drei Attribute `value1` bis `value3` anbietet:

```
public final class StaticInnerClassExample
{
    public static final class TripleVO
    {
        private final int value1;
        private final int value2;
        private final int value3;

        private TripleVO(final int value1, final int value2, final int value3)
        {
            this.value1 = value1;
            this.value2 = value2;
            this.value3 = value3;
        }

        public final int getValue1() { return value1; }
        public final int getValue2() { return value2; }
        public final int getValue3() { return value3; }
    }

    // ...
}
```

Statische innere Klassen besitzen keine implizite Referenz auf die äußere Klasse, wodurch auch keine Zugriffe auf nicht statische Attribute der äußeren Klassen möglich sind. Statische innere Klassen können mit folgender Syntax erzeugt werden:

```
final OuterClass.StaticInnerClass inner = new OuterClass.StaticInnerClass();
```

Spezielle Formen innerer Klassen

Neben den bisher vorgestellten Formen von inneren Klassen existieren noch die zwei im Folgenden vorgestellten Spezialformen: *methodenlokale* und *anonyme innere* Klassen.

Methodenlokale innere Klassen Innere Klassen können sogar lokal innerhalb von Methoden definiert werden und sind auch nur dort sichtbar und zugreifbar, weshalb sich keine Sichtbarkeit angeben lässt. Diesen lokalen inneren Klassen ist sowohl ein Zugriff auf Attribute der äußeren Klasse als auch auf die in der Methode definierten Variablen und Methodenparameter möglich, sofern sich diese nicht mehr ändern, also explizit `final` oder »***effectively final***« sind.[10]

```
private void doSomething()
{
   int counter = 100;
   final int constant = 200;

   // Keine Sichtbarkeitsmodifier erlaubt, public usw. => Compile-Error
   /* public */ class MethodLocalInnerClass
   {
      public void printVar()
      {
         // System.out.println("counter = " + counter ); // => Compile-Error
         System.out.println("constant = " + constant);
      }
   }
   counter++; // Ohne diese Zeile wäre der obige Zugriff auf counter erlaubt
   new MethodLocalInnerClass().printVar();
}
```

Diese Einschränkung auf Unveränderlichkeit ist dadurch bedingt, dass eine Stackvariable (etwa `int variable = 100`) beim Verlassen einer Methode »abgeräumt« wird. Eine methodenlokale innere Klasse kann zu diesem Zeitpunkt aber weiterhin aktiv sein (etwa weil sie einen Thread gestartet hat) und könnte später versuchen, auf eine solche Variable zuzugreifen, die es dann aber nicht mehr geben würde. Durch die Definition als `final` wird der Zugriff trotzdem möglich. Wie ist das zu erklären? Man kann sich das Vorgehen in etwa so vorstellen: Das Schlüsselwort `final` garantiert, dass sich der Wert einer Variablen im Verlauf der Methode nicht ändert. Dadurch ist es möglich, implizit eine Kopie des Werts der Variablen zu erstellen und diesen an die methodenlokale innere Klasse zu übergeben. Somit arbeitet diese mit einer Kopie statt mit dem Original aus der umgebenden Methode.

Interessanterweise kann man bis Java 14 im Gegensatz zu Klassen weder `enum`-Aufzählungen noch Interfaces methodenlokal definieren. Das ändert sich mit Java 15 (vgl. Abschnitt 14.1.6). Allerdings dürfte dies nur in seltenen Fällen notwendig sein.

[10] Seit JDK 8 muss eine Variable nicht mehr explizit `final` definiert werden, sondern es reicht aus, wenn sich deren Wert nicht mehr ändert. Diese Unveränderlichkeit wird vom Compiler erkannt.

Anonyme innere Klassen Manchmal sind innere Klassen so speziell und nur für eine einmalige Aufgabe verwendbar, dass man sie weder benennen noch mehrfach erzeugen möchte. Das lässt sich mithilfe anonymer innerer Klassen realisieren. Diese sind innerhalb von Klassen oder Methoden definiert und besitzen keinen Namen, sind also anonym. Ohne Klassennamen können sie allerdings keinen Konstruktor bereitstellen. Meistens bestehen diese Klassen nur aus wenigen Methoden – häufig sogar lediglich aus einer. Man spricht von ***SAM-Typen*** (***Single Abstract Method***). Diese spielen für JDK 8 und ***Lambdas*** eine wichtige Rolle, was wir bereits in Kapitel 4 genauer betrachtet haben. Anonyme innere Klassen sind in ihrer Definition dahingehend eingeschränkt, dass sie entweder auf einem Interface basieren oder eine Klasse erweitern müssen. Der ausschließliche Sinn besteht darin, die Methoden der Basisklasse zu überschreiben bzw. die Methoden eines Interface zu implementieren. Zwar kann man in einer anonymen inneren Klasse zusätzliche Methoden definieren, allerdings können diese niemals von außerhalb der Klasse aufgerufen werden.

Die zur Definition verwendete Schreibweise ist zunächst vielleicht etwas gewöhnungsbedürftig, da weder das Schlüsselwort `extends` noch `implements` angegeben wird. Stattdessen folgt die Klassendefinition syntaktisch direkt der Instanziierung der Basisklasse bzw. des Interface:[11]

```
final Runnable newRunnable = new Runnable()
{
    public void run()
    {
        // ...
    }
}; // ACHTUNG DAS SEMIKOLON IST WICHTIG => SONST COMPILE-ERROR => OCPJP/SCJP
```

Anonyme innere Klassen und der Diamond Operator in JDK 9

Bis einschließlich Java 8 kann bei der Definition anonymer innerer Klassen der Diamond Operator leider nicht genutzt werden, sondern der Typ ist explizit anzugeben. Seit JDK 9 ist es (endlich) erlaubt, die Typangabe auszulassen und somit den Diamond Operator zu verwenden, wie wir dies von anderen Variablendefinitionen bereits gewohnt sind. Die neue Syntax ist hier am Beispiel eines Komparators dargestellt:

```
final Comparator<String> byLength = new Comparator<>()
{
    ...
};
```

[11] In diesem speziellen Fall kann daher genau nur das angegebene Interface erfüllt werden. Sind mehrere Interfaces zu erfüllen, so ist es erforderlich, dass man ein neues Interface einführt, das die benötigten Interfaces erweitert.

5.5 Ein- und Ausgabe (I/O)

Die meisten Programme müssen mit anderen Applikationen interagieren oder Benutzereingaben entgegennehmen, um Berechnungen auszuführen. Java bietet zur Ein- und Ausgabe in den Packages `java.io` und `java.nio` einen objektorientierten Zugang.

Bitte beachten Sie, dass es bei der Kommunikation und Ein- und Ausgabe immer auch zu Fehlern oder zumindest Problemen kommen kann, etwa dass eine Datei nicht vorhanden ist oder nicht in sie geschrieben werden kann. Aus Gründen der Übersichtlichkeit und weil wir das Thema Fehlerbehandlung erst später in Abschnitt 5.6 vertiefen, wird in den nachfolgenden Beispielen zur API-Demonstration der Ein- und Ausgabe auf eine ansonsten notwendige Fehlerbehandlung weitestgehend verzichtet.

5.5.1 Dateibehandlung und die Klasse `File`

Eine Instanz der Klasse `java.io.File` kann entweder eine Datei oder ein Verzeichnis im Dateisystem repräsentieren. Dazu speichert ein `File`-Objekt den Namen und den Pfad lediglich textuell, aber nicht das tatsächliche Dokument oder dessen Inhalt. Es existieren daher in der Klasse `File` keine Methoden, um Daten in eine Datei zu schreiben oder daraus zu lesen. Dazu lernen wir in Abschnitt 5.5.2 die Ein- und Ausgabestreams kennen.

Pfad-Aktionen

Ein `File`-Objekt konstruiert man aus der Angabe eines Pfadnamens. Dieser kann aus dem Datei- oder Verzeichnisnamen und optional der Angabe des übergeordneten Verzeichnisses bestehen:[12]

- `new File(String pathname)`
- `new File(String parentDirName, String filename)`
- `new File(File parentDir, String filename)`

In der Pfadangabe werden sowohl einfache Slashes '/' als auch Backslashes '\' als Pfadtrenner akzeptiert. Auch die Metaverzeichnisse '.' (aktuelles Verzeichnis) und '..' (übergeordnetes Verzeichnis) können dort enthalten sein. Wie auch bei regulären Ausdrücken ist für das Backslash-Zeichen allerdings ein Escapen notwendig.

Nachdem ein `File`-Objekt erzeugt wurde, kann man verschiedene Aktionen ausführen. Die Methode `getName()` liefert den Dateinamen ohne Pfadinformationen. Diese erhält man über `getAbsolutePath()`. Dabei werden auch alle enthaltenen Metainformationen (beispielsweise '.' und '..') ausgegeben. Allerdings werden Slashes '/' und Backslashes '\' in das jeweilige Format des Betriebssystems vereinheitlicht. Eine Auflösung einer beliebigen Pfadangabe in Form eines Strings in einen rea-

[12] Da ein `File`-Objekt lediglich einen Pfad im Dateisystem repräsentiert, wird durch einen Konstruktoraufruf weder eine Datei noch ein Verzeichnis angelegt. Das erfordert explizit einen Methodenaufruf.

len Pfad im Dateisystem erfolgt über die Methode `getCanonicalPath()`. Der Pfad "`C:\toBeChecked\./../toBeChecked/`" wird beispielsweise in folgendes reale Verzeichnis übersetzt: "`C:\toBeChecked`". Da die Methode `getCanonicalPath()` zum Auflösen der Pfadinformationen nicht nur Strings verkettet, sondern zum Teil auf dem Dateisystem operiert, können `java.io.IOException`s ausgelöst werden.

Datei-Aktionen

Nachdem wir zuvor einige wichtige Details zur Verarbeitung von Pfaden kennengelernt haben, betrachten wir nun einige elementare Methoden der Klasse `File`:

- `isFile()` und `isDirectory()` – Diese Methoden prüfen, ob ein `File`-Objekt eine Datei bzw. ein Verzeichnis repräsentiert.
- `exists()` – Prüft, ob ein `File`-Objekt tatsächlich im Dateisystem existiert.
- `createNewFile()` – Erzeugt eine neue Datei, die durch das `File`-Objekt repräsentiert wird.
- `delete()` – Löscht die Datei, die durch das `File`-Objekt repräsentiert wird. Gleiches gilt für Verzeichnisse, allerdings nur, wenn diese leer sind.
- `mkdir()` und `mkdirs()` – Ein Aufruf der Methode `mkdir()` erzeugt das durch das `File`-Objekt angegebene Verzeichnis. Mit `mkdirs()` werden bei Bedarf auch alle dazu benötigten übergeordneten Verzeichnisse angelegt.
- `list()` – Ermittelt den Inhalt eines Verzeichnisses als `String[]`. Die `list()`-Methode liefert in Fehlersituationen den Wert `null` statt eines `String`-Arrays der Länge 0. Details dazu beschreibt der folgende Hinweis »Rückgabewerte der `list()`- bzw. `listFiles()`-Methoden«. Aufrufer müssen daher Spezialfälle unterscheiden, was einerseits zusätzliche Schreibarbeit bedeutet und andererseits potenziell fehlerträchtig ist oder vergessen wird.
- `listFiles()` – Ermittelt, ähnlich zu `list()`, den Inhalt eines Verzeichnisses. Die Rückgabe erfolgt hier als `File[]`.

Hinweis: Rückgabewerte der `list()`- bzw. `listFiles()`-Methoden

Die beiden Methoden `list()` und `listFiles()` provozieren `NullPointerException`s bei Aufrufern, da sie `null` zurückgeben, wenn kein Verzeichnisinhalt ermittelt werden kann. Dies ist immer dann der Fall, wenn der Pfad im `File`-Objekt kein Verzeichnis darstellt oder aber ein I/O-Fehler auftritt. Besser wäre für den ersten Fall die Rückgabe eines Arrays der Länge 0 gewesen. Ein I/O-Fehler sollte meiner Meinung nach durch eine `IOException` und nicht durch einen `null`-Wert kommuniziert werden.[a] Durch das API müssen Anwender jeweils Spezialbehandlungen einbauen.

[a] Genauer diskutiere ich die obige Problematik in Abschnitt 16.3.6 als BAD SMELL: RÜCKGABE VON NULL STATT EXCEPTION IM FEHLERFALL.

Die Interfaces `FileFilter` und `FilenameFilter`

Realisierungen der Interfaces `java.io.FileFilter` und `java.io.FilenameFilter` erlauben, die Ergebnismenge der gerade vorgestellten `list()`- bzw. `listFiles()`-Methoden einzuschränken, etwa um in einem Verzeichnis alle `pdf`-Dateien zu ermitteln. In beiden Interfaces ist eine Methode `accept()` definiert. Diese ist so zu implementieren, dass sie für gewünschte Objekte den Wert `true` zurückgibt.

Im Falle des `FileFilter`s erhält die Methode `accept()` ein `File`-Objekt als Parameter. Dadurch besteht innerhalb der Implementierung der Methode Zugriff auf den Dateinamen und verschiedene weitere Attribute, insbesondere auch den Pfad.

```
public interface FileFilter
{
    boolean accept(File pathname);
}
```

Das Interface `FilenameFilter` abstrahiert weiter. Statt eines `File`-Objekts dient nur noch der Dateiname sowie das übergeordnete Verzeichnis als Eingabe:

```
public interface FilenameFilter
{
    boolean accept(File dir, String name);
}
```

Diese Angabe ist hilfreich, weil man nur mit dem Dateinamen keinen Zugriff auf die Attribute der Datei besitzt. Benötigt man diese und weitere Informationen, so kann man den zusätzlichen Parameter, d. h. das übergeordnete Verzeichnis, auswerten.

Filterung am Beispiel Der folgende Sourcecode-Ausschnitt zeigt, wie man nach `.xml`- bzw. `.pdf`-Dateien filtern kann. Ersteres wird in diesem Beispiel durch einen `FilenameFilter` realisiert. Letzteres nutzt einen `FileFilter` als Basis. Beide Klassen wandeln die jeweiligen Dateinamen mit `toLowerCase()` in Kleinbuchstaben um, wodurch der Vergleich unabhängig von der Schreibweise wird.

```
public static class XmlFilenameFilter implements FilenameFilter
{
    @Override
    public boolean accept(final File dir, final String fileName)
    {
        return fileName.toLowerCase().endsWith(".xml");
    }
}

public static class PdfFileFilter implements FileFilter
{
    @Override
    public boolean accept(final File pathName)
    {
        return pathName.getName().toLowerCase().endsWith(".pdf");
    }
}
```

Parametrisierte Filter Wollte man einen etwas komplexeren Filter realisieren, der exakt auf ein als Parameter übergebenes Präfix und Postfix prüft, so könnte man Folgendes implementieren:

```
public class PreAndPostFixFilenameFilter implements FilenameFilter
{
   private final String prefix;
   private final String postfix;

   public PreAndPostFixFilenameFilter(final String prefix, final String postfix)
   {
      this.prefix = prefix;
      this.postfix = postfix;
   }

   @Override
   public boolean accept(final File dir, final String name)
   {
      return name.startsWith(prefix) && name.endsWith(postfix);
   }
}
```

Vereinfachungen mit JDK 8 Mit dem in JDK 8 neu eingeführten Sprachfeature der Lambdas (vgl. Kapitel 4) lässt sich die Implementierung der beiden Interfaces deutlich kürzer wie folgt schreiben:

```
final FileFilter pdfFilter = (final File file) ->
{
    return file.getName().toLowerCase().endsWith(".pdf");
}

final FilenameFilter xmlFilter = (final File dir, final String fileName) ->
{
    return fileName.toLowerCase().endsWith(".xml");
}
```

Auch der speziellere Prä- und Postfix-Filter lässt sich recht einfach implementieren. Wir müssen hier einen kleinen Trick nutzen: Weil wir Parameter an den Filter übergeben wollen, erzeugen wir den Filter mithilfe einer Methode, die einen Lambda zurückliefert:

```
public static FilenameFilter createPreAndPostFixFilter(final String prefix,
                                                       final String postfix)
{
    return (dir, fileName) -> { return fileName.startsWith(prefix) &&
                                       fileName.endsWith(postfix); };
}
```

Diese Beispiele zeigen, wie hilfreich die mit JDK 8 eingeführten Lambdas sind, um kurze und prägnante Realisierungen zu erstellen.

5.5.2 Ein- und Ausgabestreams im Überblick

Wenden wir uns nach der Vorstellung der Dateiverarbeitung nun der Modifikation von Dateiinhalten zu. Man unterscheidet dabei zwischen byteorientierter und zeichenbasierter Ein- und Ausgabe. Erstere stellen die Klassen `java.io.InputStream` und `java.io.OutputStream` bereit. Beide bilden die Schnittstelle zu Datenquellen und -senken, deren Daten als Bytefolgen repräsentiert sind. Die zeichenbasierte Ein- bzw. Ausgabe erfolgt in Form von `char`s mithilfe der Basisklassen `java.io.Reader` bzw. `java.io.Writer`. Diese Klassen können problemlos Umlaute und Sonderzeichen verarbeiten, sofern die gewählte Zeichencodierung (Encoding) stimmt – Details erläutere ich später in Abschnitt 5.5.3.

Vererbungshierarchie der Streams

Für `Input`- und `OutputStream`s und `Reader` und `Writer` existieren zwei analoge Vererbungshierarchien. Abbildung 5-2 zeigt wichtige Subklassen der abstrakten Basisklasse `InputStream`, beispielsweise um die Eingabe aus einer Datei zu lesen (`java.io.FileInputStream`).

Abbildung 5-2 *Ableitungen der Klasse* `InputStream`

Spezielle Klassen erlauben, dynamisch weitere Funktionalität, etwa eine Pufferung (`java.io.BufferedInputStream`), hinzuzufügen. Zwei besondere Streams zur Interaktion mit der Konsole werden durch die statischen Attribute `System.in` und `System.out` als gepufferte Streams bereitgestellt. ***Aufgrund der Pufferung sind dort manche Ausgaben nicht direkt sichtbar.***

Zur Verarbeitung des Inhalts einer Datei kann man die Klassen `FileInputStream` und `java.io.FileOutputStream` einsetzen. Um die vom Betriebssystem bereitgestellten Ressourcen zu allozieren bzw. wieder freizugeben, bedarf es der Operationen Öffnen und Schließen. Das Öffnen erfolgt meistens implizit (z. B. bei der Konstruktion eines Streams). Für das Schließen ist die Applikation dagegen selbst verantwortlich. Hilfreich ist dabei das mit JDK 7 eingeführte Automatic Resource Management (ARM) (vgl. Abschnitt 5.6).

Die Klassen `InputStream` und `Reader`

Alle `InputStream`s bieten Methoden zum Lesen von Daten und zum Prüfen, ob Daten zum Einlesen vorliegen.

Die Klasse `InputStream` In der abstrakten Basisklasse `InputStream` sind bereits einige überladene `read()`-Methoden implementiert. Viele rufen die abstrakte Methode `read()` auf, die von konkreten Realisierungen der Java-I/O-Bibliotheken abgestimmt auf das jeweilige Eingabemedium (Datei, Socket usw.) implementiert wird. Folgende Aufzählung stellt die wichtigsten Methoden und deren Intention vor:

- `abstract int read()` – Liest das nächste Byte aus dem Eingabestrom. Dieses wird jedoch als `int` codiert. Ein Rückgabewert von -1 signalisiert das Ende des Streams. Ansonsten liegen die eingelesenen Werte im Bereich von 0 – 255 und müssen zur Weiterverarbeitung gegebenenfalls auf ein `byte` gecastet werden. Werte über 127 werden dadurch in negative Zahlen umgewandelt.
- `int read(byte[] buffer)` – Liest die nächsten Bytes aus dem Eingabestrom ein, maximal aber so viele, wie das übergebene `byte[]` groß ist. Sind im Eingabestrom keine Daten vorhanden, so signalisiert ein Rückgabewert von -1 das Ende des Streams. Ansonsten werden die Werte im übergebenen `byte[]` gespeichert, und es wird die Anzahl der tatsächlich gelesenen Bytes zurückgeliefert. Sollen mehr Daten gelesen werden, als derzeit im Eingabestrom vorliegen, so blockiert der Aufruf die weitere Ausführung.
- `int read(byte[] buffer, int pos, int len)` – Sofern verfügbar, werden die nächsten `len` Bytes aus dem Eingabestrom eingelesen und ab Position `pos` im übergebenen `byte[]` gespeichert.
- `int available()` – Liefert die Anzahl der im Eingabestrom zur Verfügung stehenden Bytes. Diese können anschließend über eine der `read()`-Methoden ausgelesen werden, ohne zu einer Blockierung zu führen.
- `void close()` – Der Eingabestrom wird geschlossen, und es werden alle dadurch belegten Betriebssystemressourcen wieder freigegeben.

Erweiterungen in der Klasse `InputStream` in JDK 9

Die Verarbeitung von `Input`- und `OutputStream`s bietet bis einschließlich JDK 8 mitunter nicht den Komfort, den man erwarten würde. Das gilt etwa für das Einlesen und das Kopieren von Daten. Diese Funktionalitäten werden mit JDK 9 durch die folgenden Methoden der Klasse `InputStream` realisiert:

- `readAllBytes()` – Liest alle Bytes aus dem Stream.
- `transferTo(OutputStream)` – Diese Methode erlaubt das Kopieren von Daten von einem `InputStream` in einen `OutputStream`.

Die Klasse `Reader` Die Klasse `Reader` besitzt ähnliche Methoden wie die Klasse `InputStream`. Allerdings arbeitet ein `Reader` nicht auf Bytes, sondern auf Zeichen. Statt wie beim `InputStream` das nächste einzelne Byte zu lesen, stützen sich alle Einlesemethoden auf eine abstrakte Methode `read(char[], int, int)` ab, die einen Teilbereich einlesen kann. Nachfolgend werden einige Methoden der Klasse `Reader` kurz aufgelistet:

- `abstract int read(char[] buffer, int pos, int len)` – Sofern verfügbar, werden die nächsten `len` Zeichen aus dem Eingabestrom eingelesen und ab Position `pos` im übergebenen `char[]` gespeichert.
- `int read()` – Liest das nächste Zeichen aus dem Eingabestrom als `int` und liefert es in der Unicode-Codierung zurück, d. h., es werden gültige Werte im Bereich von 0 – 65535 geliefert. Ein Wert von -1 signalisiert das Ende des Streams.
- `int read(char[] buffer)` – Liest die nächsten Zeichen aus dem Eingabestrom, maximal aber `buffer.length` Zeichen. Die Werte werden im übergebenen `char[]` gespeichert.
- `boolean ready()` – Prüft, ob im Eingabestrom Daten zur Verfügung stehen. Im Unterschied zur Klasse `InputStream` werden keine Angaben über die Anzahl der verfügbaren Daten gemacht.

Die Klassen `OutputStream` und `Writer`

Alle Ausgabestreams bieten Methoden zum Schreiben von Daten in einen Stream.

Die Klasse `OutputStream` Ähnlich zum `InputStream` existieren hier einige überladene `write()`-Methoden, die sich auf eine abstrakte `write()`-Methode stützen. Diese wird von den konkreten Klassen der Java-I/O-Bibliotheken, spezifisch für das jeweilige Ausgabemedium, implementiert. Folgende Aufzählung stellt die wichtigsten Methoden und ihre Aufgaben vor:

- `abstract void write(int byte)` – Schreibt das übergebene Byte `byte` in den Ausgabestrom. Die höherwertigen Bytes des übergebenen `int`-Werts werden dabei ignoriert.
- `void write(byte[] buffer)` – Schreibt alle Bytes aus dem übergebenen `byte[]` in den Ausgabestrom.
- `void flush()` – Entleert den Ausgabestrom und erzwingt ein physikalisches Schreiben möglicherweise zwischengespeicherter Daten.
- `void close()` – Der Ausgabestrom wird geschlossen und es werden alle dadurch belegten Betriebssystemressourcen wieder freigegeben.

Die Klasse `Writer` Die Klasse `Writer` besitzt sehr ähnliche Methoden wie die Klasse `OutputStream`, die allerdings nicht auf Bytes, sondern auf Zeichen arbeiten. Neben den Methoden `flush()` und `close()` finden sich einige überladene `write()`-Methoden, die sowohl `char`-Arrays oder Bereiche als auch Strings und Teilstrings in den Stream schreiben können.

Konvertierung

Die Konvertierung eines `InputStream`s in einen `Reader` wird durch die Klasse `java.io.InputStreamReader` ermöglicht. Das entsprechende Gegenstück für die Ausgabe stellt die Klasse `java.io.OutputStreamWriter` dar, die einen `OutputStream` in einen `Writer` umwandelt.

5.5.3 Zeichencodierungen bei der Ein- und Ausgabe

Für textuelle Daten gibt es insbesondere bei deren Verarbeitung mit byteorientierten Streams einige Dinge zu bedenken: Das Ganze setzt eine Umwandlung in und aus einer Folge von Bytes voraus. Beim Austausch bytecodierter Daten ist allerdings nicht immer sichergestellt, dass die Gegenseite die gleiche Interpretation der Folge von Bytes bei der Rücktransformation in einen String vornimmt. Betrachten wir dies genauer.

Die Klasse `String` speichert bis JDK 8 die Texte als ein Array von `char`, das eine Unicode-Codierung verwendet. Seit JDK 9 kommt ein `byte[]` sowie ein zusätzliches Attribut zur Steuerung der Zeichencodierung zum Einsatz (vgl. Abschnitt 5.3.5). Um eine Abbildung zwischen einem Stringobjekt und einem puren `byte[]` zu ermöglichen, existieren spezielle ***Charsets*** oder Zeichencodierungen, die Zeichen auf Zahlenwerte abbilden. Mithilfe verschiedener Charsets kann man die interne Unicode-Repräsentation in einen beliebigen anderen Zeichensatz umwandeln: Es existieren sogenannte ***Encoder*** und ***Decoder***, die dazu dienen, eine Sequenz von Bytes in einer beliebigen Codierung mit einem `CharsetDecoder` in Unicode-Zeichen umzuwandeln bzw. mit einem `CharsetEncoder` eine Transformation von Unicode in eine Folge von Bytes zu vollziehen. In Java 9 gilt dies für Unicode oder Latin 1 je nachdem, wie der String parametriert ist.

Die Umwandlung eines Strings in ein `byte[]` kann durch Aufruf der Methoden `getBytes()`, `getBytes(Charset)` und `getBytes(String)` erfolgen, die einen String in ein Byte-Array gemäß dem Defaultcharset bzw. dem angegebenen Charset konvertieren. Dabei kann das Charset entweder als `Charset`-Objekt oder per Name übergeben werden.

Ermitteln verfügbarer `Charset`-Objekte

Leider sind nicht alle der erlaubten Namen als Konstanten definiert und zudem manchmal recht kryptisch. So nutzt die DOS-Konsole unter Windows den Zeichensatz »Cp850« (Cp steht für Codepage). Windows selbst verwendet die Codierung »Cp1252«

(Windows-1252 bzw. Latin 1), die eine Anpassung der Codierung »ISO-8859-1« ist. Weitere bekannte Codierungen sind »UTF-8« und »UTF-16«.[13]

Eine `Map<String, Charset>` mit etwa 150 Abbildungen von Namen auf die zugehörigen `Charset`s kann man über die statische Methode `Charset.availableCharsets()` ermitteln. Zugriff auf ein konkretes `Charset` ermöglicht die statische Methode `forName(String)`. Eine Auswahl gebräuchlicher Charsets ist in der Klasse `java.nio.charset.StandardCharsets` definiert.

Einfluss des gewählten `Charset`-Objekts beim Aufruf von `getBytes()`

Die »normalen« Buchstaben besitzen in den gebräuchlichen Charsets den gleichen Zahlenwert (ASCII-kompatibel). Unterschiede finden sich aber bereits bei Umlauten, weshalb ich nachfolgend als Beispiel den String "`ÄÖÜ To`" nutze. Umlaute werden einerseits durch verschiedene Zahlenwerte dargestellt und andererseits werden sie in UTF-8 mit zwei Bytes statt in einem Byte codiert. Das zeigt Tabelle 5-5.

Tabelle 5-5 *Einfluss verschiedener Encodings*

Charset-Encoding	Ä	Ö	Ü	␣	T	o
Default (für Windows)	-60	-42	-36	32	84	111
Cp850	-114	-103	-102	32	84	111
ISO-8859-1	-60	-42	-36	32	84	111
UTF-8	-61, -124	-61, -106	-61, -100	32	84	111

Zum Nachvollziehen starten Sie das folgende Programm STRINGENCODINGS – mitunter gibt es Abweichungen in den Werten, wenn man Mac OS und Linux nutzt.

```
public static void main(final String[] args) throws UnsupportedEncodingException
{
    final byte[] def = "ÄÖÜœTo".getBytes();
    final byte[] dos = "ÄÖÜ To".getBytes("Cp850");
    final byte[] iso = "ÄÖÜ To".getBytes("ISO-8859-1");
    final byte[] utf8 = "ÄÖÜ To".getBytes("UTF-8");

    printBytes(def);  // "Default:" [-60, -42, -36, 32, 84, 111]
    printBytes(dos);  // "Cp850": [-114, -103, -102, 32, 84, 111]
    printBytes(iso);  // "ISO-8859-1": [-60, -42, -36, 32, 84, 111]
    printBytes(utf8); // "UTF-8": [-61, -124, -61, -106, -61, -100, 32, 84, 111]
}
```

Listing 5.29 *Ausführbar als* **'STRINGENCODINGS'**

[13] Eine Liste gültiger Namen findet man unter `http://download.oracle.com/javase/8/docs/technotes/guides/intl/encoding.doc.html`.

Auswahl des passenden `Charset`-Objekts

Wenn man sich beim Datenaustausch darauf verlässt, dass das Defaultcharset passt, führt dies immer wieder zu Problemen. Besser ist es, dass gewünschte Charset explizit zu spezifizieren. Die Wahl hängt davon ab, welche Komponenten die Daten verarbeiten sollen. Greifen auf eine Datei ausschließlich Java-Komponenten, sowohl lesend als auch schreibend, zu, wird man UTF-8 wählen. Sind dagegen Komponenten als Erzeuger oder Konsument beteiligt, die dieses Charset nicht unterstützen, etwa ältere Programme, die z. B. mit ASCII-Darstellungen arbeiten, wird man ein passendes `Charset`-Objekt nutzen, mit dessen Codierung diese Komponenten als Leser umgehen können bzw. die sie als Erzeuger vorgeben. Darüber hinaus gibt es Dateiformate, bei denen das Encoding festgelegt ist und die damit die Auswahlmöglichkeiten einschränken.

5.5.4 Speichern und Laden von Daten und Objekten

Im Folgenden werde ich anhand der Klasse `Person` die Speicherung eines Objekts in einem Stream sowie das Lesen und Rückgewinnen eines Objekts aus einem Stream vorstellen. Dieses einführende Beispiel ist vereinfacht und betrachtet nicht die in der Realität auftretenden Herausforderungen, etwa verschiedene Zeichensätze, umfangreiche Datenmengen, abweichende Versionen der gespeicherten Daten usw.

Wir beginnen mit einer Verarbeitung von einfachen Bytestreams, um danach den Einsatz der komfortableren `DataOutputStream`s bzw. `PrintStream`s zu zeigen. Die folgenden Beispiele verzichten der Übersichtlichkeit halber auf eine Fehlerbehandlung. Dieses wichtige Thema behandle ich dann in Abschnitt 5.6.

Einsatz der Klassen `FileInputStream` und `FileOutputStream`

Es sollen Objekte vom Typ `Person` gespeichert und geladen werden. Folgendes Listing zeigt zur Erinnerung die zu verarbeitenden Attribute:

```
public class Person
{
    private final String name;
    private final String city;
    private final LocalDate birthday;
    // ...
```

Ein erster, naiver Ansatz besteht darin, direkt die Methoden `read()` und `write()` aus den Klassen `InputStream` und `OutputStream` zu verwenden, um die drei Attribute zu verarbeiten. Damit die Applikation nicht mit Details überfrachtet wird, erstellen wir zwei Hilfsmethoden `readPersonFromStream(InputStream)` und `writePersonToStream(Person, OutputStream)`, die diese Aufgaben unter Verwendung von `read()` und `write()` erledigen und kapseln. Folgendes Programm erzeugt ein `Person`-Objekt und ruft anschließend die beiden Hilfsmethoden auf, um das Objekt vom Typ `Person` zu speichern und anschließend wieder zu laden:

```
public static void main(final String[] args) throws Exception
{
    final String tmpDir = System.getProperty("java.io.tmpdir");
    final File file = new File(tmpDir, "Person.ser1");
    final Person max = new Person("Max", "Hamburg", LocalDate.now());

    // Speichere Person 'Max' in einer Datei und lies die Daten wieder ein
    writePersonToStream(max, new FileOutputStream(file));
    final Person newPerson = readPersonFromStream(new FileInputStream(file));

    // Prüfe auf inhaltliche Gleichheit
    System.out.println("Gleich? " + max.equals(newPerson));
}
```

Listing 5.30 *Ausführbar als* '**PERSONSTREAMFILESTREAM**'

Schauen wir uns nun einige Details sowie die Realisierung der beiden Methoden an. Beginnen wir dabei mit dem Speichern von Daten in einem Stream.

Schreiben Das Schreiben von Daten in einen Stream ist komplizierter, als es zunächst wirkt. Um die Daten später wieder korrekt einlesen zu können, muss man sich eine spezielle Form der Speicherung überlegen, die beim Einlesen eine korrekte Rekonstruktion des Objekts erlaubt. Das erfordert für jedes Attribut eine Repräsentation und Konvertierung in eine Folge von Bytes. Alle Daten ohne Trennzeichen oder Längeninformation hintereinander zu schreiben, würde das Einlesen nahezu unmöglich machen: In dem Bytestrom wäre keine Struktur erkennbar und die Daten der Attribute wären nicht voneinander abzugrenzen.[14] Zur Unterscheidung könnte man spezielle Trennzeichen einfügen, die in den Nutzdaten natürlich nicht enthalten sein dürfen (bzw. ein gesondertes Escapen erfordern). Alternativ kann man für jede Byterepräsentation eines Attributs die Länge der Konvertierung mit in den Datenstrom schreiben. Letzteres wollen wir nutzen. Dazu implementieren wir eine Hilfsmethode, die die Daten in eine Codierung aus Längenangabe, Leerzeichen und anschließend der Nutzbytes überführt und schreibt. Für dieses Beispiel wird vereinfachend davon ausgegangen, dass die zu verarbeitenden Daten nur aus Folgen von Bytes bestehen, die nicht länger als 255 Zeichen sind. Diese Vereinfachung erlaubt es, die Methode übersichtlich zu halten und nicht mit Logik zur Längenverarbeitung zu überfrachten. Die Hilfsmethode `writeByteArray(OutputStream, byte[])` wird wie folgt realisiert:

```
public static void writeByteArray(final OutputStream os,
                                  final byte[] bytesToWrite) throws IOException
{
    // Die beiden folgenden Aufrufe nutzen die Methode write(int)
    // Tatsächlich werden die Werte automatisch auf den Typ byte beschränkt
    os.write(bytesToWrite.length);
    os.write(' ');

    // Aufruf der Methode write(byte[])
    os.write(bytesToWrite);
    os.flush();
}
```

[14] Es sei denn, alle Felder haben eine feste und konstante Länge.

Der Aufruf der Methode `flush()` nach dem Schreiben erfolgt, um in den Stream geschriebene Daten tatsächlich auch physikalisch in das durch den Stream gekapselte Medium zu übertragen, etwa bei gepufferter Verarbeitung.

Hinweis: Resource Handling

Die in den Listings gezeigten Realisierungen sind vereinfacht (aber funktionstüchtig). Für die Praxistauglichkeit fehlt eine Freigabe von Ressourcen nach den Aktionen, etwa durch einen Aufruf von `close()`.

Konvertierung in ein `byte[]` Wir können nun ein `byte[]` in einen Stream schreiben. Das ist schon ein guter Schritt, aber die Daten der Attribute liegen selten in Form eines `byte[]` vor. Also stellt sich die Frage: Wie konvertieren wir die Daten in ein `byte[]`? Für Strings ist diese Umwandlung noch relativ einfach, da die Klasse `String` bekanntermaßen eine `getBytes()`-Methode anbietet. Aber bereits beim `LocalDate`-Attribut für das Geburtsdatum müssen wir ein wenig tricksen: Wir können es zunächst in einen String wandeln und dann wie gewohnt weiterarbeiten. Allerdings ist diese Vorgehen nur möglich, weil `LocalDate` ein fixes Format und ein passendes Parsing anbietet. Für diverse andere Klassen führt eine simple textuelle Repräsentation schnell zu Problemen durch verschiedene Interpretationen beim Parsing.

Zum Schreiben des `Person`-Objekts in einen `OutputStream` erstellen wir die Hilfsmethode `writePersonToStream()` wie folgt:

```
public static void writePersonToStream(final Person person,
                                       final OutputStream os) throws IOException
{
    // String => getBytes()
    final byte[] nameBytes = person.getName().getBytes();
    final byte[] cityBytes = person.getCity().getBytes();

    // LocalDate => String => getBytes()
    final String dateString = person.getBirthday().toString();
    final byte[] dateBytes = dateString.getBytes();

    // Schreibe Attribute in den Stream
    writeByteArray(os, nameBytes);
    writeByteArray(os, cityBytes);
    writeByteArray(os, dateBytes);
}
```

Beim Betrachten des Sourcecodes erahnen wir, dass eine solche Art der Konvertierung relativ aufwendig und fehleranfällig werden kann. Doch machen wir erst einmal weiter, bevor wir nach Alternativen schauen.

Lesen Bei der Rückgewinnung von Werten sind einige Dinge zu berücksichtigen, damit wir beim Einlesen aus einer Menge von Bytes wieder die korrekten Informationen ermitteln können. Dabei ist es wichtig, die Details und die Codierung des Schreibens genau zu beachten. Basierend auf der Methode `writeByteArray(OutputStream,`

`byte[])` zum Schreiben erstellen wir die korrespondierende Hilfsmethode `readByteArray()` zum Einlesen wie folgt:

```
public static byte[] readByteArray(final InputStream is) throws IOException
{
    final int bytesLength = is.read();
    is.read();    // Überspringe Leerfeld

    final byte[] bytes = new byte[bytesLength];
    is.read(bytes);
    return bytes;
}
```

Sichere Objektrekonstruktion aus Streams Wir haben nun einige Grundbausteine zum Lesen und Schreiben von Daten erstellt, aber wie wandeln wir die Daten aus einem `InputStream` in ein `Person`-Objekt? Voraussetzung dafür ist, dass beim Einlesen die Implementierungsdetails des Schreibens bekannt sind und das Einlesen der Werte in der gleichen Reihenfolge wie beim Schreiben erfolgt. Auch die Umwandlung in die gewünschten Typen der Attribute setzt eine genaue Kenntnis des zu rekonstruierenden Objekts voraus, weil wir aus Gründen der Vereinfachung des Beispiels keine Typinformationen in den Stream geschrieben haben.[15]

Um inkonsistente Objektzustände und andere Probleme zu vermeiden, ist es empfehlenswert, erst alle für die Objektkonstruktion nötigen Informationen aus dem Stream zu sammeln und das Objekt nur dann zu erzeugen, wenn die Daten vollständig sind:

```
private static Person readPersonFromStream(final InputStream is) throws
      FileNotFoundException, IOException
{
    final byte[] nameBytes = readByteArray(is);
    final String name = new String(nameBytes);

    final byte[] cityBytes = readByteArray(is);
    final String city = new String(cityBytes);

    final byte[] birthdayBytes = readByteArray(is);
    final LocalDate birthday = LocalDate.parse(new String(birthdayBytes));

    // Trick: immer erst alle Daten vor der Konstruktion einlesen
    return new Person(name, city, birthday);
}
```

Mögliche Fallstricke bei der Objektrekonstruktion aus Streams Auf ein spezielles Problem beim Einlesen und der Objektrekonstruktion möchte ich an dieser Stelle hinweisen. Häufig sieht man initial die Konstruktion eines Objekts ohne Zustandsinformationen durch Aufruf eines Defaultkonstruktors. Der Grund ist simpel: Es fehlen einige Informationen, weil diese noch nicht eingelesen wurden. Stück für Stück werden diese Informationen dann nach erfolgreichem Einlesen per `set()`-Methoden

[15] Würde man Typinformationen ebenfalls in den Stream schreiben, so würde man damit beginnen, den in Java integrierten Serialisierungsautomatismus (vgl. Abschnitt 12.3) nachzubauen.

den entsprechenden Attributen zugewiesen. Der Sourcecode sieht dann, vereinfacht und auf die Klasse `Person` übertragen, wie folgt aus:

```
final Person newPerson = new Person();
// read name
newPerson.setName(name);
// read city
newPerson.setCity(city);
// read birthday
newPerson.setBirthday(birthday);
```

Diese Technik der sukzessiven Initialisierung über `set()`-Methoden ist zu vermeiden, da sie fehleranfällig ist. Bei dieser Vorgehensweise drohen inkonsistente Objektzustände, falls nicht alle Daten ermittelt werden können, das Objekt aber schon angelegt wurde. Eine ausführliche Erörterung dieses Problems finden Sie in Abschnitt 3.1.5.

Einsatz der Klassen `DataInputStream` und `DataOutputStream`

Das Speichern und Lesen von `Person`-Objekten in bzw. aus Streams kann durch Einsatz der Klassen `java.io.DataInputStream` und `java.io.DataOutputStream` vereinfacht werden. Beide Klassen bieten spezielle Methoden an, um Daten in einen Stream zu speichern bzw. daraus zu lesen. Dadurch muss man sich für primitive Typen und Strings keine Gedanken um die Konvertierung in Bytefolgen machen. Nützlich sind für dieses Beispiel unter anderem folgende Methoden:

- `writeUTF()` und `readUTF()` – Verarbeitung von Strings
- `writeLong()` und `readLong()` – Verarbeitung von primitiven Typen, hier `long`

Das Schreiben von Personendaten in Streams ist mithilfe dieser Methoden sehr einfach wie folgt möglich:

```
private static void writePersonToStream(final Person person,
                                        final DataOutputStream dataOutStream)
                                              throws IOException
{
    dataOutStream.writeUTF(person.getName());
    dataOutStream.writeUTF(person.getCity());
    dataOutStream.writeUTF(person.getBirthday().toString());
    dataOutStream.flush();
}
```

Auch das Einlesen wird erleichtert und lässt sich folgendermaßen realisieren – wobei hier der obige Tipp bezüglich des sukzessiven Einlesens berücksichtigt ist:

```
private static Person readPersonFromStream(final DataInputStream dataInStream)
      throws IOException
{
    final String name = dataInStream.readUTF();
    final String city = dataInStream.readUTF();
    final LocalDate birthday = LocalDate.parse(dataInStream.readUTF());

    return new Person(name, city, birthday);
}
```

Offensichtlich sind diese Realisierungen kürzer und übersichtlicher. Sie besitzen zudem folgende Vorteile:

1. Es werden gut getestete Standardmethoden aus dem JDK eingesetzt. Die Realisierungen sind dadurch weniger fehleranfällig, besser lesbar und auch für Entwickler verständlich, die die zuvor selbst geschriebenen Methoden nicht kennen.
2. Die Umwandlung verschiedener Typen (`int`, `long` usw.) in eine Folge von Bytes und zurück geschieht durch die spezifischen, überladenen Methoden der Streams, etwa `readLong()`, für jede nutzende Applikation transparent.
3. Der Einsatz der Methoden `readUTF()` und `writeUTF()` erleichtert die Verarbeitung von Strings, z. B. müssen keine Längeninformationen der zu verarbeitenden Strings in den Stream geschrieben bzw. daraus gelesen werden. Dies war bei den gezeigten Eigenimplementierungen erforderlich, bei der eine Codierung der Länge als Byte stattfand. Dadurch kam es zu Längenbeschränkungen für die zu verarbeitenden Strings. Selbst wenn ein primitiver Datentyp mit größerem Wertebereich zum Einsatz kommt, besteht das Problem weiterhin.

Allerdings gibt es nach wie vor das Problem, die Reihenfolge bei der Speicherung und bei einem späteren Einlesen exakt einzuhalten – ansonsten drohen Inkonsistenzen, Zugriffsfehler oder gar Exceptions. Im ungünstigsten Fall merkt man es noch nicht einmal, beispielsweise wenn die Angaben zu Namen und Ort vertauscht werden.

Ausgaben mit der Klasse `PrintStream`

Die Klasse `java.io.PrintStream` stellt die von der Konsolenausgabe bekannten, für verschiedene Typen überladenen `print()`- und `println()`-Methoden zur Verfügung. Verwendet man die Klasse `PrintStream` zur Aufbereitung der Ausgabe, so kann man von den einzelnen spezifischen `write()`-Methoden des `DataOutputStream`s weiter abstrahieren. Man nutzt stattdessen die überladenen `println()`-Methoden:

```
private static void writePersonToPrintStream(final Person person,
                                             final PrintStream printStream)
                                                   throws IOException
{
    printStream.println(person.getName());
    printStream.println(person.getCity());
    printStream.println(person.getBirthday());
    printStream.flush();
}
```

Es existiert allerdings kein passendes Stream-Gegenstück zum `PrintStream`, das dessen Ausgaben wieder einliest. Das Einlesen kann man in diesem Fall durch die Klasse `Scanner` realisieren, die nun besprochen wird.

Die Klasse `Scanner`

Die Klasse `java.util.Scanner` kann Eingabestrings aus verschiedenen Datenquellen in einzelne Bestandteile, sogenannte ***Tokens***, zerlegen und dabei primitive Typen direkt aus der Eingabe parsen. Bei der Konstruktion eines `Scanner`-Objekts ist die gewünschte Eingabequelle zu übergeben, die vom Typ `String`, `File`, `InputStream` oder `java.lang.Readable` ist. Letzteres wird von den beiden abstrakten Klassen `CharBuffer` und `Reader` implementiert und ermöglicht so eine Verarbeitung der I/O-Hierarchie der zeichenbasierten `Reader`. Nach der Konstruktion kann über die Methode `useDelimiter(String)` spezifiziert werden, welcher reguläre Ausdruck die Trennzeichenfolge beschreibt. Sofern dies nicht geschieht, wird als Trennzeichenfolge jede Folge von Whitespace-Charactern (Spaces, Tabs und Zeilenumbruch) verwendet.

Die Klasse `Scanner` implementiert das Interface `Iterator<String>`, wobei die Methode `remove()` eine `java.lang.UnsupportedOperationException` auslöst. Die Methode `next()` gibt das nächste Token als `String` zurück. Dies ist jedoch nur sicher möglich, wenn die Methode `hasNext()` die Existenz eines nachfolgenden Elements bestätigt hat. Ansonsten wird eine `java.util.NoSuchElementException` ausgelöst, wenn keine weiteren Daten mehr folgen. Das Einlesen von Werten primitiver Datentypen erledigen überladene `next<Typ>()`-Methoden, etwa `nextLong()`. Analog zur `hasNext()`-Methode existieren `hasNext<Typ>()`-Methoden, etwa `hasNextLong()`. Durch deren Aufruf lässt sich feststellen, ob ein weiteres Token vom gewünschten Typ in der Eingabe vorhanden ist. Liest man jedoch ungeprüft beispielsweise einen `long`-Wert aus einer beliebigen Eingabequelle und entspricht der gelesene Wert nicht dem erwarteten Typ, so reagiert der `Scanner` mit einer `java.util.InputMismatchException`.

Beispiel: Texte mit Scanner verarbeiten Wir nutzen nun die Trennzeichenfolge aus den Zeichen '.', '_' und '-', um eine Versionsinformation in Einzelbestandteile aufzusplitten.

```
public static void main(final String[] args)
{
    // ARM: Spezielle try-Syntax seit JDK 7
    try (final Scanner scanner = new Scanner("Version-2.17_45"))
    {
        scanner.useDelimiter("\\.|_|-");
        while (scanner.hasNext())
        {
            System.out.print(scanner.next() + " ");
        }
    }
}
```

Listing 5.31 *Ausführbar als* **'SCANNEREXAMPLE'**

Startet man das Programm SCANNEREXAMPLE, so erhält man folgende Ausgabe:

```
Version 2 17 45
```

Abschließend sei darauf hingewiesen, dass man am Ende einer Verarbeitung mit einem `Scanner` diesen durch Aufruf der Methode `close()` wieder schließen sollte, um eventuell belegte Systemressourcen freizugeben. Im obigen Listing sehen wir den Einsatz von Automatic Resource Management (ARM) (vgl. Abschnitt 5.6.3), wodurch die Ressourcen automatisch freigegeben werden, ohne dass ein expliziter Aufruf von `close()` erforderlich ist. Das macht den Sourcecode kürzer und übersichtlicher.

Ausgaben einlesen Kommen wir nun zu unserem Beispiel zurück: Mithilfe der Klasse `PrintStream` wurden die einzelnen Attribute eines `Person`-Objekts mit einem Zeilenendezeichen als Trenner in einen Stream geschrieben. Diese Daten sollen mit einem `Scanner` wieder eingelesen werden. Dieser nutzt Whitespace und insbesondere ein Zeilenende als Begrenzer, weil nicht explizit Trennzeichen über `useDelimiter()` bekanntgegeben worden sind. Das Einlesen kann man wie folgt realisieren:

```
private static Person readPersonWithScanner(final Scanner scanner) throws
      IOException
{
    final String name = scanner.next();
    final String city = scanner.next();
    final String date = scanner.next();
    final LocalDate birthday = LocalDate.parse(date);

    return new Person(name, birthday, city);
}
```

Die hier gezeigte Umsetzung verzichtet bewusst auf die normalerweise sinnvollen `hasNext()`-Prüfungen. Wir erwarten ein ganz spezielles Datenformat, da wir dieses zuvor selbst geschrieben haben. Fehler in den Eingabedaten werden durch diese Verarbeitung sofort sichtbar und führen zu Exceptions.

Tipp: Fehler während des Parsens

Bei der Verarbeitung von Eingaben durch einen `Scanner` kann es zu Fehlern in Form von `IOException`s kommen. Diese werden nicht weiter propagiert, sondern abgefangen und intern gespeichert. Die zuletzt aufgetretene Exception liefert die Methode `ioException()`, die bei Fehlerfreiheit `null` zurückgibt.

Fazit

Anhand der vorangegangenen Beispiele der manuellen Verarbeitung von `Person`-Objekten mit Streams sieht man, dass das Speichern und Einlesen von Objekten kompliziert werden kann. Dies gilt insbesondere, da in der Praxis Objekte selten so einfach aufgebaut sind wie in diesem Beispiel. Vielmehr werden häufig Referenzen auf andere Objekte gehalten. Die Komplexität steigt, wenn man Objekte mit allen referenzierten Objekten, den sogenannten ***Objektgraph***, beim Speichern und Einlesen berücksichtigt muss. Es bietet sich in diesem Fall eher der Einsatz des in Java integrierten Serialisierungsmechanismus an, der in Abschnitt 12.3 genauer beschrieben wird.

5.5.5 Dateiverarbeitung mit dem NIO

In diesem Abschnitt befassen wir uns mit dem NIO (New Input Output) und schauen uns aus dem Package `java.nio.file` die Klassen und Interfaces `DirectoryStream`, `Files`, `FileSystem`, `Path`, `Paths`, `FileVisitor` und `WatchService` an.

Das Interface `Path`

Einen wichtigen Bestandteil zur Interaktion mit dem Dateisystem bildet das Interface `Path`. Zugriff auf ein Objekt vom Typ `Path` erhält man durch Aufruf der überladenen Methoden `get(String)` bzw. `get(java.net.URI)` aus der Hilfsklasse `Paths`. Alternativ lässt sich die Methode `getPath(String, String...)` der Klasse `FileSystem` nutzen. Über das Interface `Path` wird ein Pfad oder im Speziellen eine Datei repräsentiert, die verarbeitet werden soll. Für die eigentlichen Aktionen und Zugriffe ist die Hilfsklasse `Files` zuständig. Dort finden wir Methoden wie `copy(Path, Path, CopyOption...)` und `move(Path, Path, CopyOption...)` zum Kopieren und Verschieben von Dateien. Die Funktionalität des NIO kann auch mit dem älteren API von `File` verwendet werden. Durch Aufruf der Methode `toPath()` lassen sich `File`-Objekte in `Path`-Objekte konvertieren, wie dies hier nachfolgend gezeigt ist:

```
public static void main(final String[] args) throws IOException
{
    // Zugriff auf das systemspezifische tmp-Directory
    final String tmpDir = System.getProperty("java.io.tmpdir");

    // Konstanten für die betroffenen Dateien
    final String SOURCE_FILE_PATH = "src/ch05_javagrundlagen/fileio/" +
                                    "FileIOExample.java";
    final String DESTINATION_FILE_PATH1 = tmpDir + "CopiedFile.java";
    final String DESTINATION_FILE_PATH2 = tmpDir + "CopyOfFileIOExample.java";

    // Erzeugen eines Path-Objekts mit der Klasse FileSystem
    final FileSystem local = FileSystems.getDefault();
    final Path fromPath = local.getPath(SOURCE_FILE_PATH);

    // Erzeugen von Path-Objekten mit der Utility-Klasse Paths
    final Path toPath1 = Paths.get(DESTINATION_FILE_PATH1);
    final Path toPath2 = Paths.get(DESTINATION_FILE_PATH2);

    // Dateien vorsorglich löschen
    Files.deleteIfExists(toPath1);
    Files.deleteIfExists(toPath2);

    // Kopieren und verschieben
    Files.copy(fromPath, toPath1);
    Files.move(toPath1, toPath2);

    final File dir = new File(tmpDir);
    final String[] content =
                   dir.list(new PreAndPostFixFilenameFilter("Copy", ".java"));
    System.out.println("Content of tmpDir '" + tmpDir + "': " +
                       Arrays.toString(content));
}
```

Listing 5.32 *Ausführbar als* '**FILEIOEXAMPLE**'

Das Interface `Path` stellt diverse nützliche Methoden zum Bearbeiten von `Path`-Objekten bereit. Es kann beispielsweise ein Teilpfad mit `subpath(int, int)` extrahiert oder per `relativize(Path)` ein Pfad in einen Pfad relativ zu einem bestehenden Pfad umgewandelt werden. Zur Interaktion mit älterem Sourcecode besteht die Möglichkeit, `File`-Objekte durch Aufruf der Methode `toPath()` in `Path`-Objekte zu konvertieren, wie dies hier für das aktuelle Verzeichnis gezeigt ist:

```
public static void main(final String[] args)
{
    // Konvertierung java.io.File -> java.nio.file.Path
    final Path currentDirOldStyle = new File(".").toPath();

    final FileSystem fileSystem = FileSystems.getDefault();
    final Path currentDir = fileSystem.getPath(".");

    // Als absoluten Pfad ausgeben
    final Path absolutePath = currentDir.toAbsolutePath();
    System.out.println(absolutePath);

    // Pfadbestandteile ermitteln
    System.out.println(absolutePath.getNameCount());
    for (int i = 0; i < absolutePath.getNameCount(); i++ )
    {
        System.out.println(absolutePath.getName(i));
    }
}
```

Listing 5.33 *Ausführbar als* '**PATHOPERATIONSEXAMPLE**'

Das Interface `DirectoryStream`

Das Interface `DirectoryStream` implementiert das Interface `Iterable<Path>` und ermöglicht dadurch, über den Inhalt eines Verzeichnisses zu iterieren. Auf diese Weise kann ein sukzessives Verarbeiten der darin enthaltenen Dateien und Unterverzeichnisse erfolgen. Durch Aufruf der Methode `newDirectoryStream(Path)` der Hilfsklasse `Files` erhält man den beschriebenen Zugriff:

```
public static void main(final String[] args) throws IOException
{
    // Konvertierung java.io.File -> java.nio.file.Path
    final Path currentDir = new File(".").toPath();

    // ARM: Spezielle try-Syntax seit JDK 7
    try (final DirectoryStream<Path> dirStream =
                                     Files.newDirectoryStream(currentDir))
    {
        for (final Path entry : dirStream)
        {
            System.out.println(entry);
        }
    }
}
```

Listing 5.34 *Ausführbar als* '**DIRECTORYSTREAMEXAMPLE**'

Zunächst mag man sich fragen, was der Vorteil an einem `DirectoryStream` sein soll. Tatsächlich ist bei Verzeichnissen auf dem lokalen Rechner mit einer üblichen Anzahl an Dateien (10 – 100) der Vorteil nicht sofort ersichtlich. Es wird aber eine effiziente Verarbeitung auch für solche Verzeichnisse möglich, die sehr viele Einträge enthalten. Bei dieser Form der Verarbeitung werden zudem weniger Ressourcen benötigt, als wenn man den Verzeichnisinhalt vollständig »in einem Rutsch« einlesen würde. Letzteres führt außerdem häufig zu (längeren) Wartezeiten. Durch den Einsatz eines `DirectoryStream`s können selbst bei einer großen Anzahl enthaltener Dateien und Verzeichnisse gute Antwortzeiten sichergestellt werden. Diesen Vorteil bemerkt man vor allem, wenn Verzeichnisse auf anderen Computern über ein Netzwerk mithilfe eines `DirectoryStream`s untersucht werden.

Ähnlich zu den Methoden `list()` und `listFiles()` der Klasse `File` ist es durch die Angabe eines Filters möglich, die Treffermenge einzuschränken.

Die Utility-Klasse `Files`

In der Utility-Klasse `java.nio.file.Files` des NIO (New Input Output[16]) gibt es einiges zu entdecken. Stellvertretend dafür betrachten wir folgende Hilfsmethoden:

- `lines(Path)` – Stellt eine Datei zeilenweise als `Stream<String>` bereit.
- `readAllLines(Path)` – Liest eine Datei zeilenweise ein und gibt die Zeilen als `List<String>` zurück.
- `list(Path)` – Liefert den Inhalt eines Verzeichnisses als `Stream<Path>`. Das Besondere dabei ist, dass der Inhalt sukzessive bei Bedarf ermittelt wird und nicht direkt von vornherein. Es wird, wie im Abschnitt über Streams beschrieben, immer nur ein Teil der Daten angefordert, wenn eine Terminal Operation ausgeführt wird.
- `write(Path, Iterable<? extends CharSequence>, OpenOption...)` – Schreibt die übergebenen Textzeilen in die durch den `Path`-Parameter referenzierte Datei. Der Schreibmodus wird durch die angegebene `OpenOption` bestimmt, etwa `APPEND` oder `WRITE`, und überschreibt gegebenenfalls vorhandene Informationen.

Die oben dargestellten Funktionalitäten sind besonders nützlich, was ich anhand eines Beispiels verdeutlichen möchte. Hierbei erzeugen und befüllen wir eine Textdatei namens `WriteText.txt` im Temp-Verzeichnis des Systems mit ein wenig Inhalt und nutzen dazu die Methode `write()`. Die in die Datei geschriebenen Informationen ermitteln wir in Form eines `Stream<String>`s durch Aufruf von `lines()`. Auf dem Ergebnis wenden wir eine Filterung und eine Gruppierung an. Details zu dem hier genutzten Stream-API finden Sie in Kapitel 7. Abschließend inspizieren wir mithilfe von `list()` das Temp-Verzeichnis und ermitteln alle Dateien, die mit `.txt` enden. Dabei gilt es zu beachten, dass man nicht die Methode `endsWith(String)` des `Path`-Objekts nutzen

[16] Wobei ein »New« in einem Namen potenziell immer schwierig ist. In diesem Fall existiert das NIO bereits seit Java 1.4.

kann, da diese auf Pfadbestandteilen arbeitet und nicht auf Namen. Daher muss zuvor eine Umwandlung in einen String erfolgen:

```
public static void main(final String[] args) throws IOException
{
   final String tempDirPath = System.getProperty("java.io.tmpdir");
   final Path destinationFile = Paths.get(tempDirPath + "/WriteText.txt");
   final List<String> content = Arrays.asList("This", "is", "the", "content");

   // Datei schreiben
   final Path resultFile = Files.write(destinationFile, content,
                                       StandardOpenOption.CREATE,
                                       StandardOpenOption.APPEND);

   // Zeilenweise als Stream<String> einlesen
   try (final Stream<String> contentAsStream = Files.lines(resultFile))
   {
      // Filtern und Gruppieren
      final Map<Integer, List<String>> filterdAndGrouped = contentAsStream.
                                 filter(word -> word.length() > 3).
                                 collect(Collectors.groupingBy(String::length));
      System.out.println(filterdAndGrouped);
   }

   // Verzeichnis als Stream<Path> einlesen
   try (final Stream<Path> tmpDirContent = Files.list(Paths.get(tempDirPath)))
   {
      // Fallstrick: endsWith arbeitet auf Path, nicht auf Dateinamen!
      tmpDirContent.filter(path -> path.toString().endsWith(".txt")).
                                                 forEach(System.out::println);
   }
}
```

Listing 5.35 *Ausführbar als* '**FILESEXAMPLE**'

Nach ein paarmal Starten erhält man dann in etwa folgende Ausgabe:

```
{4=[This, This, This, This], 7=[content, content, content, content]}
C:\Users\micha\AppData\Local\Temp\WriteText.txt
```

5.5.6 Neue Hilfsmethoden in der Klasse `Files` in JDK 11

In Java 11 wurde die Verarbeitung von Strings im Zusammenhang mit Dateien erleichtert. Es ist nun einfach möglich, Strings in eine Datei zu schreiben bzw. daraus zu lesen. Dazu hat die Utility-Klasse `java.nio.file.Files` ein paar Erweiterungen in Form der Methoden `writeString()` und `readString()` erhalten.

Beispiel

Folgendes Beispiel zeigt das Schreiben bzw. Lesen eines Strings, wobei standardmäßig von UTF-8 als Codierung ausgegangen wird:

```
var destinationPath = Path.of("ExampleFile.txt");

Files.writeString(destinationPath, "1: This is a 'string to file' test\n");
Files.writeString(destinationPath, "2: Second line");

final String line1 = Files.readString(destinationPath);
final String line2 = Files.readString(destinationPath);

System.out.println(line1);
System.out.println(line2);
```

Führt man dieses Programmfragment aus, so erwartet man vermutlich die beiden zuvor geschriebenen Zeilen als Ausgabe, doch man erhält abweichend Folgendes:

```
2: Second line
2: Second line
```

Das liegt daran, dass beim Schreiben etwas überraschend der Überschreibemodus und nicht der Modus des Anhängens genutzt wird. Folgende Modifikation behebt das:

```
Files.writeString(destinationPath, "1: This is a 'string to file' test\n");
Files.writeString(destinationPath, "2: Second line", StandardOpenOption.APPEND);

final String line1 = Files.readString(destinationPath);
final String line2 = Files.readString(destinationPath);
```

Führen wir das Ganze nochmals aus, so könnte die Ausgabe wiederum verwundern:

```
1: This is a 'string to file' test
2: Second line
1: This is a 'string to file' test
2: Second line
```

Nun wird die gesamte Datei als eine Zeile eingelesen. Somit spannt sich der Bogen zu `String.lines()` aus Abschnitt 5.3.6. Damit nehmen wir dann folgende Korrektur vor:

```
Files.writeString(destinationPath, "1: This is a 'string to file' test\n");
Files.writeString(destinationPath, "2: Second line", StandardOpenOption.APPEND);

var content = Files.readString(destinationPath);

content.lines().forEach(System.out::println);
```

Dadurch erhalten wir dann die erwartete Ausgabe:

```
1: This is a 'string to file' test
2: Second line
```

5.6 Fehlerbehandlung

Jedes komplexere Programm muss mit gewissen Fehlersituationen umgehen können. Es erfordert immer etwas an Mehrarbeit und zusätzliche Zeilen Sourcecode, auf unerwartete Situationen reagieren oder Fehler abfangen zu können. Dieser Aufwand ist allerdings erforderlich, um für stabile Software zu sorgen.

Zur Behandlung von Fehlern oder außergewöhnlichen Situationen bieten viele moderne Programmiersprachen mit Exceptions die Möglichkeit zur Verarbeitung (Erkennung und Behandlung oder Propagation) von Fehlerzuständen. Insbesondere ist es möglich, Exceptions zu propagieren, falls man an der betreffenden Stelle nicht in der Lage ist, den Fehler angemessen zu behandeln.

Eine fehlende oder inkorrekte Fehlerbehandlung macht sich im User Interface z. B. dadurch bemerkbar, dass der Wartecursor nicht wieder auf den Standardcursor gesetzt wird. In der Business-Logik oder der Datenzugriffsschicht kommt es u. a. zu nicht freigegebenen Ressourcen. Solche Probleme in tieferen Ebenen der Software bleiben dem Anwender mitunter verborgen und machen sich oft erst später bemerkbar.

5.6.1 Einstieg in die Fehlerbehandlung

Java bietet zur Fehlerbehandlung mehrere Alternativen. Zum einen lassen sich mit sogenannten ***Exceptions*** ungewöhnliche Situationen behandeln. Zum anderen können Fehlersituationen durch spezielle Rückgabewerte beschrieben werden. Schließlich ist es mit ***Assertions*** möglich, gewisse Zustände im Programm zu prüfen. Diese Varianten werden im nachfolgenden Listing anhand einer Methode beispielhaft illustriert:

```
public Object findById(final long objectId)
{
    // Ungültiger Parameterwert => Exception
    if (objectId < 0)
        throw new IllegalArgumentException("objectId must not be negative");

    if (DbHelper.isValidId(objectId))
    {
        final Object obj = DbHelper.findById(objectId);

        // Zustandsprüfung mit Assertion
        assert (obj != null);

        return obj;
    }

    // Spezieller Rückgabewert für keinen Treffer
    return null;
}
```

Eine Faustregel ist es, Exceptions für solche Situationen zu verwenden, die außerhalb des erwarteten Verhaltens liegen. Kann eine aufgerufene Methode einen Rückgabewert liefern, der es erlaubt, eine Fehlersituation sinnvoll auszudrücken, so sind Rückgabewerte zum Teil handlicher. Lässt sich eine Fehlersituation auf diese Weise allerdings nicht darstellen oder kann bei einem fehlerhaften Eingabewert nicht sinnvoll weiter-

gearbeitet werden, so sollte man eine Exception auslösen. Dazu dient – wie im Listing bereits gezeigt – das Schlüsselwort `throw`, dem eine Exception-Instanz folgt.

Bereits anhand dieses Beispiels wird Folgendes erkennbar: Die Behandlung von Fehlern besitzt zahlreiche Facetten und ein allgemein richtiges Verhalten gibt es nicht. Vielmehr sollte man eine für die jeweilige Situation adäquate Reaktion auf einen Fehler vornehmen. Insbesondere empfiehlt es sich, immer einen passenden Exception-Typ zu wählen. Vier recht gebräuchliche finden sich im Package `java.lang`:

- `IllegalArgumentException` – Mit einer `IllegalArgumentException` können falsche Belegungen von Parametern ausgedrückt werden.
- `NullPointerException` – Sind Eingabewerte `null`, so kann man darauf mit einer `NullPointerException` reagieren.
- `IllegalStateException` – Sind benötigte Daten nicht korrekt initialisiert, so kann dies über eine `IllegalStateException` kommuniziert werden.
- `UnsupportedOperationException` – Auf eine fehlende Implementierung kann mittels einer `UnsupportedOperationException` hingewiesen werden.

Hinweis: `IllegalArgumentException` vs. `NullPointerException`

Ich persönlich bevorzuge es, bei einem Übergabewert von `null` keine `NullPointerException`, sondern eine `IllegalArgumentException` auszulösen, da sie klarer ausdrückt, dass ein Parameter einen falschen Wert besitzt. Eine `NullPointerException` ist für mich eher eine Exception, die auf unerwartete Initialisierungsprobleme hindeutet. Bei der Nutzung der JDK-Hilfsmethode `Objects.requireNonNull()` weiche ich von dem genannten Ratschlag ab, da diese eine `NullPointerException` auslöst.

Behandlung von Exceptions

Mitunter können Programme sinnvoll fortgeführt werden, obwohl eine Fehlersituation aufgetreten ist. Dazu bedarf es in der Regel einer dem Problem angepassten Fehlerbehandlung. In Java kann man mit dem Exception Handling potenziell Fehler auslösenden Sourcecode von Fehler behandelndem Sourcecode trennen und sorgt so für mehr Übersichtlichkeit. Im `try`-Block stehen diejenigen Anweisungen, die potenziell Fehler verursachen, getreu dem Motto »Give it a try – Versuch mal, ob es funktioniert«. Wenn dann ein Fehler auftritt, kann dieser durch eine Exception modelliert werden. Zur Fehlerbehandlung dient ein `catch`-Block, in dem man den Sourcecode notiert, der dazu ausgeführt werden soll.

Am Beispiel der zuvor gezeigten Methode `findById(long)` verdeutliche ich die Behandlung von Exceptions: Jeder Abschnitt des Sourcecodes, der potenziell eine Exception auslösen kann, muss entweder wie nachfolgend in einem `try`-Block ausgeführt werden oder die Exception in der Signatur aufführen:[17]

[17] Das gilt nur, wenn die Exception vom Typ Checked Exception (vgl. Abschnitt 5.6.2) ist.

```
try
{
    // Potenziell Exception auslösend
    final Object obj = findById(objectId);

    // ...
}
catch (final IllegalArgumentException ex)
{
    // Fehlerbehandlung
    errorCounter++;

    showMessageBox("Ungültige objectId -- darf nicht negativ sein!");
}
```

Tritt bei der Abarbeitung der Anweisungen des `try`-Blocks ein Fehler auf, der eine Exception auslöst, so stoppt die Verarbeitung der Anweisungen im `try`-Block. Es wird direkt ein auf die Exception ausgelegter `catch`-Block angesprungen – sofern ein solcher existiert. Im `catch`-Block erfolgt die Fehlerbehandlung, etwa indem eine Fehlermeldung ausgegeben oder ein Warndialog angezeigt wird. Es können auch mehrere `catch`-Blöcke existieren, die zu verschiedenen Arten von Fehlersituationen korrespondieren und jeweils andere Typen von Exceptions behandeln. Bei Auftreten eines Fehlers wird der Typ der ausgelösten Exception mit den Exception-Typangaben in den `catch`-Blöcken abgeglichen und von dem Block behandelt, der mit dem Typ der Exception inklusive Basistypen (am besten) übereinstimmt. Dabei werden die Blöcke in der Reihenfolge ihrer Deklaration durchgegangen, also von oben nach unten, und der erste, der fängt, gewinnt. Alle anderen Blöcke werden nicht ausgeführt.

Automatische Propagation Existiert kein `catch`-Block, so muss eine möglicherweise auftretende Exception in der Signatur der Methode angegeben werden. Falls bei der Ausführung des Programms dann bei der Fehlerbehandlung kein `catch`-Block passt, wird die Exception automatisch an den Aufrufer weiter propagiert.

Typischer Fallstrick: Unzureichende Behandlung Leider sieht man in einigen Büchern und auch in realem Anwendungscode `catch`-Blöcke, die leer sind oder gerade noch die Exception per `ex.printStackTrace()` auf der Konsole ausgeben, etwa folgendermaßen:

```
// Achtung: Keine geeignete Reaktion auf Fehler
catch (final IllegalArgumentException ex)
{
    ex.printStackTrace();
}
```

Dies ist natürlich keine angemessene Reaktion auf eine Fehlersituation – sondern das Verhalten von unzuverlässiger Schönwettersoftware. Unser Ziel sollte es sein, dass man sich als Anwender des Programms selbst in Fehlersituationen auf dieses verlassen kann. Für unsere Kunden ist etwa wichtig, dass keine Daten zerstört werden oder verloren gehen. Für uns als Entwickler ist wichtig, dass wir möglichst genaue Informationen

zu den Ursachen und auslösenden Programmzeilen sowie aktuellen Wertebelegungen relevanter Variablen erhalten. Daher gilt: ***Behandle auftretende Exceptions, wenn dies sinnvoll möglich ist. Im Zweifelsfall sollten Exceptions an Aufrufer weiter propagiert werden***.

Spezifische Fehlerbehandlung und abgeleitete Exceptions

Oftmals können durch die Anweisungen in einem `try`-Block verschiedene Arten von Exceptions ausgelöst werden. Für diesen Fall kann man zur Behandlung mehrere `catch`-Blöcke wie folgt dafür definieren:

```
try
{
    final File file = new File(...);
    // ...
}
catch (final IOException ioe)
{
    // Fehlerbehandlung
    handleIOException(ioe);
}
catch (final ParseException pex)
{
    // Fehlerbehandlung
    handleParseException(pex);
}
```

Dabei gibt es einen Spezialfall zu bedenken: Stehen die Typen der ***Exceptions in einer Vererbungsbeziehung***, etwa `FileNotFoundException` als Spezialisierung von `IOException`, so muss in den `catch`-Blöcken immer der ***spezialisierteste Typ*** zuerst behandelt werden: Die Reihenfolge der in den `catch`-Blöcken gefangenen Exceptions muss ***entgegengesetzt zur Ableitungshierarchie*** erfolgen:

```
try
{
    final File file = new File(...);
    // ...
}
// Fehlerbehandlung entgegengesetzt zur Ableitungshierarchie der Exceptions
catch (final FileNotFoundException fnfe)
{
    handleFileNotFoundException(fnfe);
}
catch (final IOException ioe)
{
    handleIOException(ioe);
}
```

Um weniger Schreibaufwand zu haben, sieht man teilweise Folgendes:

```
// kein guter Stil
catch (final Exception ex)
{
    handleException(ex);
}
```

Das ist zwar syntaktisch möglich, jedoch nur extrem selten adäquat, da man so nicht mehr spezifisch auf unterschiedliche Probleme reagieren kann. Das behandele ich als BAD SMELL: FANGEN DER ALLGEMEINSTEN EXCEPTION später in Abschnitt 16.3.3.

Abschließende Aktionen und der `finally`-Block

Als Letztes verbleibt die Besprechung des `finally`-Blocks. Dieser ist optional und dessen Anweisungen werden immer dann ausgeführt, wenn der `finally`-Block vorhanden ist – also egal, ob ein Fehler aufgetreten ist oder nicht. Sinnvoll kann man den `finally`-Block nutzen, um auf jeden Fall benötigte Aufräumarbeiten durchzuführen – im Speziellen auch dann, wenn eine Exception ausgelöst wurde, für die kein passender `catch`-Block existiert. Schauen wir uns den Aufbau von `try-catch` und `finally` an:

```
try
{
   // Hier können Exceptions auftreten
}
catch (Exception e)
{
   // Hier werden Exceptions abgearbeitet, sofern der catch-Block vorhanden ist.
   // Ansonsten muss die Exception in der Methodensignatur aufgeführt werden,
   // falls es eine Checked Exception (vgl. Abschnitt 5.6.2) ist.
}
finally
{
   // Wird immer durchlaufen, ist allerdings optional
}
```

Eigene Exception-Typen definieren

In Java ist es leicht, eigene Typen von Exceptions, z. B. eine `CustomerNotFoundException`, zu definieren. Dabei kann man auf die später in Abschnitt 5.6.2 detaillierter behandelten Basistypen `Exception` und `RuntimeException` zurückgreifen.

```
public class CustomerNotFoundException extends Exception
{
    public CustomerNotFoundException(String message)
    {
        super(message);
    }

    public CustomerNotFoundException(String message, Throwable throwable)
    {
        super(message, throwable);
    }
}
```

Rückgabewerte vs. Exceptions

Nach dieser Einführung in die Fehlerbehandlung mithilfe von Exceptions möchte ich auf einen weiteren wichtigen Aspekt dabei eingehen: Die Fehlerbehandlung sollte möglichst sauber vom »Nutzcode« trennbar sein. Meistens lässt sich dies durch die Verwendung von Exceptions einfacher als durch den Einsatz von Rückgabewerten erreichen. Letzteres verursacht recht schnell eine Vermischung von Anwendung und Fehlerbehandlung und führt oftmals zu einer schlechteren Trennung von Zuständigkeiten oder endet gar im Chaos. Exceptions bieten mit ihrer Verarbeitung in `catch`-Blöcken eine bessere Trennung. Allerdings kann es durch mehrere zu behandelnde Exceptions schnell zu vielen `catch`-Blöcken kommen.

Ein Vorteil von Exceptions ist, dass sich diese an Aufrufer propagieren lassen, wenn man meint, an anderer Stelle besser und angemessener auf Fehlersituationen reagieren zu können. Ein Nachteil ist, dass durch Exceptions ein Overhead zur Laufzeit für das Erzeugen (Aufbereitung des Stacktrace), Durchreichen usw. entsteht.[18] Für Rückgabewerte gilt das nicht – allerdings lassen sich diese auch nicht so gut weiter propagieren.

Fallstricke bei der Fehlerbehandlung

Erfahrenen Entwicklern ist bewusst, dass zum Teil mehr Mühe und Gehirnschmalz in die Behandlung möglicher Fehlersituationen einfließen muss als in die normale Programmlogik. Dies gilt vor allem dann, wenn man mit anderen Systemen, Komponenten, Ein- und Ausgabegeräten und im Besonderen mit dem Benutzer interagiert. Für unerfahrene Programmierer ist Fehlerbehandlung oftmals eines der Dinge, um die man sich eher ungern oder halbherzig kümmert, meistens gerade so weit, dass das Programm nicht abstürzt. Zwei Beispiele für misslungene Fehlerbehandlungen sind folgende:

- **»Anstands«-Null-Prüfungen** – Solche Prüfungen dienen lediglich zur Vermeidung von `NullPointerException`s:

  ```
  public static void updateSystemState()
  {
      final SystemState state = calculateSystemState();
      if (state != null)
      {
          systemStateMap.put(KEY_SYSTEM, state);
      }
  }
  ```

  Im Falle eines `null`-Werts für die Variable `state` geschieht einfach nichts. Problematisch daran ist, dass auf diese Weise ein spezieller Zustand (`state == null`), der möglicherweise sogar ein Fehlerfall ist, kaschiert wird und zudem keine Behandlung oder Warnmeldung erfolgt. Abschnitt 16.3.7 diskutiert dies als BAD SMELL: SONDERBEHANDLUNG VON RANDFÄLLEN.

[18] Da meistens eine Ausnahmesituation vorliegt, ist der Mehraufwand oftmals irrelevant.

- **»Delayed Exception«** – Ein Beispiel für diese »Fehlerbehandlungsstrategie« sind öffentliche Methoden, die keine Konsistenzprüfung ihrer Parameter durchführen und so die Eingabe ungültiger Daten ermöglichen. Aufgrund der fehlenden Prüfung führen derartige Eingaben später potenziell zu fehlerhaften Berechnungen oder gar zu (unerwarteten) Exceptions. Im nachfolgenden Beispiel können `null`-Werte in einer `systemStateMap` gespeichert werden, obwohl diese keine gültige Eingabe darstellen. Die Konsistenzprüfung erfolgt erst bei Lesezugriffen (und damit zu spät) in der `getSystemState()`-Methode:

```
public static void setSystemState(final SystemState state)
{
    systemStateMap.put(KEY_SYSTEM, state);
}

public static SystemState getSystemState()
{
    final SystemState state = systemStateMap.get(KEY_SYSTEM);
    if (state == null) // Beispiel für schlechte Fehlerbehandlung
        throw new IllegalStateException("No entry for system state");

    return state;
}
```

  Das Problem fällt zunächst nicht auf. Erst wenn später ein Aufruf von `getSystemState()` erfolgt, wird die ungültige Eingabe sichtbar und löst eine Exception aus: Eine mögliche Fehleingabe bleibt dadurch unter Umständen sehr lange unerkannt und ein solcher Fehler wird viel zu spät erkannt.

5.6.2 Checked Exceptions und Unchecked Exceptions

In Java unterscheidet man zur Behandlung von Fehlersituationen zwischen Checked Exceptions und Unchecked Exceptions. Nachfolgende Abbildung 5-3 zeigt die Ableitungshierarchien der beiden Arten von Exceptions.

Checked Exceptions sind Bestandteil des »Vertrags« zwischen Aufrufer und Bereitsteller einer Methode und zeigen mögliche, durch Aufrufer zu erwartende Fehlersituationen an. Sie müssen daher in der Methodensignatur mit dem Schlüsselwort `throws` angegeben werden. Dies sichert ab, dass Aufrufer entweder selbst aktiv mit einem `catch`-Block darauf reagieren oder ansonsten automatisch eine Propagation an weitere aufrufende Methoden erfolgt.

Eine Behandlung ist für ***Unchecked Exceptions*** mit dem Basistyp `RuntimeException` nicht zwingend erforderlich – aber möglich, und man kann diese normal mit einem `catch`-Block bearbeiten. Weil aber Unchecked Exceptions normalerweise schwerwiegende Programmierprobleme oder unerwartete Situationen ausdrücken, ist das Verarbeiten mit einem `catch`-Block eher ungewöhnlich: Ein Aufrufer kann darauf oft nicht sinnvoll reagieren. Daher sind Unchecked Exceptions in der Regel auch nicht Bestandteil der Methodensignatur (obwohl man sie dort explizit aufführen kann).

Man kann Unchecked Exceptions etwa zur Signalisierung ungültiger Parameterwerte in Form einer `IllegalArgumentException` nutzen.

Abbildung 5-3 *Exception-Hierarchie*

Einige Programmierer geraten in die Versuchung, nur noch Unchecked Exceptions einzusetzen und eigene Exception-Klassen von `java.lang.RuntimeException` abzuleiten. Obwohl dies vom »lästigen« Bearbeiten einer möglicherweise auftretenden Exception befreit und auch keine Fehler beim Kompilieren provoziert, erschwert ein derartiges Vorgehen es manchmal doch, angemessen auf Fehlersituationen reagieren zu können: Einerseits sieht man in der Methodensignatur nicht unbedingt, dass eine Exception ausgelöst wird, weshalb dies in der Methodendokumentation aufgeführt werden sollte. Andererseits kann die Exception einfach ignoriert werden. Sie führt dann zur Laufzeit zu einem unerwarteten Programmfehler. Im Extremfall wird eine solche Exception unbehandelt bis zur `main()`-Methode bzw. `run()`-Methode des gerade aktiven Threads und damit zur JVM propagiert, wodurch dieser Thread terminiert.

Aus dieser Diskussion leiten wir folgenden Hinweis ab: ***Wenn ein Aufrufer eine außergewöhnliche Situation behandeln kann, so kann eine Checked Exception bevorzugt werden. Ist nicht davon auszugehen, dass ein Aufrufer die Fehlersituation korrigieren soll, oder ist ein Aufrufer dazu höchstwahrscheinlich nicht in der Lage, so ist eine Unchecked Exception*** die richtige Wahl.

5.6.3 Besonderheiten beim Exception Handling

Nachfolgend beschäftigen wir uns mit einigen Neuerungen aus JDK 7, die das Exception Handling vereinfachen und insbesondere auch das Resource Handling adressieren.

Multi Catch

Manchmal ist es wünschenswert, mehrere Exception-Typen gleichartig zu behandeln. Idealerweise möchte man dann nur einen gemeinsamen `catch`-Block definieren und nicht für jeden zu verarbeitenden Exception-Typ jeweils einen Block.

Betrachten wir zunächst die Situation vor JDK 7 und hier zur Demonstration eine `main()`-Methode, die eine weitere Methode `exceptionThrowingMethod()` aufruft, die wiederum zwei verschiedene Exceptions in ihrer Signatur definiert. Die klassische Umsetzung einer Behandlung sieht folgendermaßen aus:

```
public static void main(final String[] args)
{
    try
    {
        exceptionThrowingMethod();
    }
    // Gleichartige Behandlung durch zwei catch-Blöcke
    catch (final RemoteException ex)
    {
        reportException(ex);
    }
    catch (final FileNotFoundException ex)
    {
        reportException(ex);
    }
}

private static void exceptionThrowingMethod() throws RemoteException,
                                                     FileNotFoundException
{
    // ...
}
```

Mehrere identische `catch`-Block stellen eine Form der Sourcecode-Duplikation dar und verletzen das DRY-Prinzip (Don't Repeat Yourself). Für dieses Beispiel sollte man die identische Behandlung nur einmalig durchführen. Mit JDK 7 ist das Fangen mehrerer Exceptions mit nur einem `catch`-Block – ***Multi Catch*** genannt – wie folgt möglich:

```
public static void main(final String[] args)
{
    try
    {
        exceptionThrowingMethod();
    }
    // Mehrfachangabe unterschiedlicher Exceptions
    catch (final RemoteException | FileNotFoundException ex)
    {
        reportException(ex);
    }
}
```

Das Verhalten von Multi Catch ist dem mehrerer hintereinander liegender `catch`-Blöcke ähnlich. Es gibt jedoch kleinere Unterschiede – insbesondere gilt dies für die Abarbeitung bzw. die Spezialisierung und Überdeckung von Exceptions. Bei mehreren `catch`-Blöcken kann man eine explizite Behandlung auch für spezielle (Sub-)Typen realisieren, indem der speziellere Typ von Exception im Sourcecode zuerst wie folgt angegeben wird:

```
catch (final FileNotFoundException ex)
{
    reportFileException(ex);
}
catch (final IOException ex)
{
    reportIOException(ex);
}
```

Dieses Verhalten ist mithilfe von Multi Catch so nicht möglich, denn es gilt, dass im Multi Catch nur Exceptions mit unterschiedlichem Basistyp erlaubt sind. Schreiben wir trotzdem einmal Folgendes:

```
// Führt zu einem Compile-Error
catch (final FileNotFoundException | IOException ex)
{
   // ...
}
```

Für die gefangene `IOException` sowie deren Subtyp `FileNotFoundException` kommt es beim Kompilieren zu folgender Fehlermeldung – egal in welcher Reihenfolge Sie die beiden Exceptions angeben: »`The exception FileNotFoundException is already caught by the alternative IOException`«.

Final Rethrow

Das Sprachfeature Final Rethrow ermöglicht es, mehrere Exception-Typen aus einem einzigen `catch`-Block weiter zu propagieren. Somit ist es möglich, dass verschiedene Typen von Exceptions per `catch (Exception ex)` gefangen werden, diese dann aber spezifisch weiter propagiert werden können, ohne dies explizit ausprogrammieren zu müssen. Dadurch kann der `catch`-Block übersichtlich gehalten und um gewünschte Funktionalität ergänzt werden.

Betrachten wir zum Verständnis die Methode `performCalculation(String)`, die laut Signatur zwei verschiedene Typen von Exceptions werfen kann:

```
private void performCalculation(final String fileName) throws IOException,
                                                              RemoteException
```

Nehmen wir an, wir wollten diese Methode derart erweitern, dass zur besseren Wartbarkeit eine gegebenenfalls auftretende Exception in eine Log-Datei geschrieben und anschließend weiter propagiert wird. Folgende Realisierungsidee (ohne den Einsatz von Multi Catch) wäre denkbar: Durch die Notation `catch (Exception)` werden einfach

alle Typen von Exceptions gefangen, dann erfolgt ein Logging der gefangenen Exception und über `throw ex` deren Propagierung:

```
private void finalRethrowV2(final String fileName) throws IOException,
                                                          RemoteException
{
    try
    {
        performCalculation(fileName);
    }
    // Das final verhindert für erste Versionen von JDK 7 einen Compile-Error
    catch (final Exception ex)
    {
        log.error("exception occurred", ex);
        throw ex; // Compile-Error vor JDK 7
    }
}
```

Tatsächlich kompiliert diese Umsetzung vor JDK 7 nicht, da durch die Anweisung `throw ex` lediglich der allgemeine Typ `Exception`, nicht aber die in der Signatur definierten Exceptions propagiert werden. Zudem ist unklar, welche Typen von Exceptions propagiert werden können. Eine (wenn auch schlechte) Idee könnte darin bestehen, den Typ `Exception` in die Signatur aufzunehmen:

```
// ACHTUNG, nicht machen, ganz schlecht
private void finalRethrowBad(final String fileName) throws IOException,
                                                           RemoteException,
                                                           Exception
```

Mit einer solchen Angabe verliert die Signatur allerdings sämtliche Aussagekraft bezüglich der von der Methode ausgelösten Exceptions. Es wird lediglich deutlich, dass überhaupt Exceptions auftreten können.

Praktischerweise ist es seit JDK 7 möglich, mehrere Typen von Exceptions allgemein zu fangen und diese spezialisiert weiterzuleiten. Während es in früheren Versionen von JDK 7 noch notwendig war, die im `catch` angegebene Exception `final` zu definieren – woher auch der Name Final Rethrow herrührt –, kann in den aktuellen Java-Versionen auf die Angabe von `final` verzichtet werden. Die moderneren Compiler können die Unveränderlichkeit einer Variablen selbst erkennen und erfordern keine explizite Kennzeichnung mit `final` mehr. Man spricht in diesem Zusammenhang auch von »effectively final«.

Automatic Resource Management (ARM)

Bis hierher haben wir gesehen, dass Aufräumarbeiten bei I/O-Operationen durch die dazu notwendigen `try-catch`-Blöcke recht umfangreich und unleserlich werden können. Mit JDK 7 wurde genau für diese Aufgaben ein automatisches Ressourcenmanagement in Java integriert. Dieses entlastet den Entwickler von manuellen Aufräumarbeiten und es hilft dabei, weniger Fehler zu machen. Betrachten wir den Aufwand der manuellen Aufräumarbeiten ohne ARM beim Einsatz eines `BufferedReader`s:

```
// I/O ohne ARM
public static String readFirstLine(final String path) // throws IOException
{
    BufferedReader br = null;
    try
    {
        br = new BufferedReader(new FileReader(path));
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
    finally
    {
        // Diese manuellen Aufräumarbeiten werden durch ARM überflüssig
        try
        {
            if (br != null)
            {
                br.close();
            }
        }
        catch(final IOException ioe)
        {
            // ignore
        }
    }
    return "";
}
```

Zur Aktivierung des ARM muss die Verarbeitung in einem speziellen `try`-Block und mit der Angabe der später freizugebenden Ressourcenvariablen erfolgen:

```
public static String readFirstLine(final String path)
{
    // Spezielle Angabe der Ressourcenvariablen
    try (final FileReader fr = new FileReader(path);
         final BufferedReader br = new BufferedReader(fr))
    {
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
    return "";
}
```

Für jede in den runden Klammern des `try`-Blocks angegebene Variable wird beim Verlassen des `try`-Blocks *automatisch* die Methode `close()` aufgerufen. Voraussetzung dafür ist, dass die dort definierten Referenzvariablen das Interface `java.lang.AutoCloseable` erfüllen. Dann werden die Aufrufe vom Compiler an den korrespondierenden Stellen in den Bytecode generiert.

Wie im Listing gezeigt, sollte man Verkettungen von Konstruktoraufrufen vermeiden, da dann möglicherweise nicht alle Ressourcen freigegeben werden. Vielmehr ist eine separate Definition wie oben zu empfehlen.

5.6.4 Exception Handling und Ressourcenfreigabe

Nachdem wir gerade ARM kennengelernt haben, möchte ich das Thema Exceptions und Ressourcenfreigabe nun ausführlicher behandeln. Zur Demonstration wähle ich eine Netzwerkkommunikation über Sockets – zur Einführung finden Sie einige Informationen zu Sockets im nachfolgenden Hinweis »Netzwerkkommunikation mit Sockets im Kurzüberblick«. Das Beispiel zeigt eine unzureichende Fehlerbehandlung. Das Ganze wird schrittweise verbessert. In der letzten Ausbaustufe ist dann sichergestellt, dass belegte Systemressourcen auch wieder freigegeben werden.

Ausgangsbeispiel

Im nachfolgenden Listing wird ein `ServerSocket` erzeugt und dann durch Aufruf einer Methode `handleIncomingConnections()` auf eingehende Daten gewartet.

```
// Beispiel für schlechtes EXCEPTION HANDLING
public static void openAndProcess()
{
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        final ServerSocket serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

        // Weitere Verarbeitung - hier jedoch uninteressant
        handleIncomingConnections(serverSocket);

        serverSocket.close();
    }
    catch (final IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Wie bereits angedeutet ist die Implementierung der Fehlerbehandlung problematisch: Solange alle Aufrufe erfolgreich sind bzw. ohne Exceptions ablaufen, wird nach Abarbeitung der durch einen Kommentar angedeuteten Aktionen das Socket durch einen Aufruf der Methode `close()` wieder geschlossen. Wird jedoch eine Exception ausgelöst, so wird die Ausführung innerhalb des `try`-Blocks unterbrochen und es werden die Anweisungen des `catch`-Blocks ausgeführt – im Beispiel kommt es aber *nicht* zum Aufruf von `close()`. Für diesen Fall bleibt demnach die vom Betriebssystem bereitgestellte Netzwerkressource alloziert (zumindest eine gewisse Zeit).

Hinweis: Netzwerkkommunikation mit Sockets im Kurzüberblick

Sockets bilden logische Endpunkte einer Verbindung zwischen Computern und erlauben das Senden und Empfangen von Nachrichten. Die Kommunikation kann z. B. über TCP erfolgen. ***TCP*** steht für ***Transmission Control Protocol*** und stellt eine sichere und fehlerfreie Verbindung zwischen zwei Kommunikationsendpunkten im Netz dar. Ähnlich wie in einem Telefonnetz einer Firma können Kommunikationspartner über Durchwahlnummern, sogenannte ***Ports***, erreicht werden.

Sockets bieten eine streambasierte Schnittstelle zur Kommunikation. Bei dieser ***Client-Server-Kommunikation*** stellt ein Client den aktiven Part der Kommunikation dar: Er schickt Anfragen an einen Server, der wiederum Antworten an den Client sendet. Dazu wartet ein Server mithilfe eines `ServerSocket`s auf eingehende Verbindungen. Die Kommunikation zwischen Client und Server wird dann mit der Klasse `Socket` realisiert, die Zugriff auf jeweils eine `InputStream`- und eine `OutputStream`-Instanz bietet.

Schritt 1: Ressourcenfreigabe im Fehlerfall

Ein erster Reparaturschritt besteht darin, im `catch`-Block einen Aufruf der Methode `close()` zu ergänzen. Das erfordert aber einige Umbauarbeiten: Weil wir Zugriff auf die Variable `serverSocket` im `catch`-Block benötigen, muss diese vor dem `try-catch`-Block definiert werden. Zudem muss der Aufruf von `close()` im `catch`-Block durch einen weiteren `try-catch`-Block ummantelt werden, da in der Signatur der `close()`-Methode eine `IOException` definiert ist. Auf diese Exception kann man eher selten sinnvoll und oftmals nur durch einen leeren `catch`-Block reagieren.

```
// Beispiel für schlechtes EXCEPTION HANDLING
public static void openAndProcess()
{
    ServerSocket serverSocket = null;
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

        handleIncomingConnections(serverSocket);

        serverSocket.close();
    }
    catch (final IOException ex)
    {
        if (serverSocket != null)
        {
            try
            {
                serverSocket.close();
            }
            catch (final IOException e)
            {
                // Sollte close() eine IOException werfen? Ja und Nein!
                // Häufig ist das eher unpraktisch. Es ist aber hilfreich,
                // wenn sich das Programm merken will, dass eine Ressource
                // nicht korrekt geschlossen werden konnte und der weiter-
                // gehende Zugriff erst einmal unterbunden werden soll.
            }
        }
    }
}
```

Der Konstruktor der Klasse `ServerSocket` kann eine `IOException` auslösen. Daher darf man im `catch`-Block nicht davon ausgehen, dass der Variablen `serverSocket`

ein Wert zugewiesen wurde und diese ungleich `null` ist. Vor dem Aufruf von `close()` muss daher eine Prüfung auf `null` erfolgen. Solche Spezialbehandlungen machen den Sourcecode unübersichtlich. Sowohl die bedingte Anweisung als auch die Behandlung der `IOException` erzeugen weitere Komplexität und reduzieren die Lesbarkeit. Diese »hässliche« Form der Fehlerbehandlung sieht man – bedingt durch das Java-API bei Sockets und beim Stream-I/O – leider häufiger. Darüber hinaus sind die mehrfachen Aufrufe von `close()` eine Form der Duplikation, was gegen das DRY-Prinzip verstößt.

Schritt 2: Duplikation entfernen

Die Sourcecode-Duplikation lässt sich leicht korrigieren, indem man eine Methode `closeServerSocket()` herausfaktoriert, die aus den Anweisungen des `catch`-Blocks besteht: Die Methode ist tolerant gegenüber der Übergabe von `null`-Werten und zudem wird die in der Signatur der Methode `close()` angegebene `IOException` ignoriert, da man diese kaum sinnvoll behandeln kann.

```
private static void closeServerSocket(final ServerSocket serverSocket)
{
    if (serverSocket != null)
    {
        try
        {
            serverSocket.close();
        }
        catch (final IOException e)
        {
            // oftmals keine sinnvolle Behandlung möglich (s.o.)
        }
    }
}
```

Derartige Hilfsmethoden machen den Sourcecode oftmals deutlich übersichtlicher, kürzer und besser lesbar. Dadurch erhalten wir folgende Methode, die den gemeinsamen Aufruf von `closeServerSocket()` hinter den `catch`-Block verlagert:

```
public static void openAndProcess()
{
    ServerSocket serverSocket = null;
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

        handleIncomingConnections(serverSocket);

        // nach unten verlagert: closeServerSocket(serverSocket);
    }
    catch (final IOException ex)
    {
        // nach unten verlagert: closeServerSocket(serverSocket);
    }
    // DRY: keine Duplikation
    closeServerSocket(serverSocket);
}
```

Das war bereits ein guter Schritt in die richtige Richtung: Die Methode ist nun deutlich besser lesbar und behandelt Fehler relativ gut. Wieso nur relativ gut? Wir haben noch ein »Schlupfloch« vergessen. Es existiert ein Problem, wenn `return`-Anweisungen im `try`- oder im `catch`-Block verwendet werden oder dort Exceptions ausgelöst oder weiter propagiert werden. Wird ein `try`- oder `catch`-Block auf diese Weise verlassen, so wird der dem `catch`-Block nachfolgende Sourcecode, in diesem Fall die `closeServerSocket()`-Methode, nicht ausgeführt. Dadurch kann sich das Programmverhalten in Fehlersituationen auf unbestimmte Weise verändern. Möglicherweise werden belegte Systemressourcen nicht wieder freigegeben. Betrachten wir mögliche Abhilfen.

Schritt 3: Variante A: Nutzung von `finally`

Mithilfe des Schlüsselworts `finally` kann das Exception Handling elegant und robust gestaltet werden: Laut JLS (Java Language Specification) wird der `finally`-Block nämlich immer ausgeführt,[19] und zwar unabhängig davon, ob

- der `try`-Block normal beendet,
- eine Exception im `try`-Block geworfen oder
- ein `return` im `try`- und `catch`-Block ausgeführt wird.

Durch dieses Verhalten lassen sich Systemressourcen freigeben – unabhängig davon, wie eine Methode beendet wird, sogar bei einem `return` im `try`- und `catch`-Block:

```
public static boolean openAndProcess()
{
   ServerSocket serverSocket = null;
   try
   {
      final int AUTO_ALLOCATE_PORT = 0;
      serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

      handleIncomingConnections(serverSocket);

      return true; // Ausstieg zur Demonstration
   }
   catch (final IOException e)
   {
      // Die Freigabe der Ressource erfolgt hier in finally. Exception darf
      // nicht propagiert werden, weil sie nicht in der Signatur definiert ist.

      return false;
   }
   finally
   {
      closeServerSocket(serverSocket);
   }
}
```

Trotz der ganzen Korrekturen sind wir immer noch gezwungen, eine `IOException` abzufangen und einen `catch`-Block zu schreiben. Das liegt daran, dass die `IOException`

[19] Außer beim Aufruf von `System.exit()`.

eine sogenannte Checked Exception ist: Wie schon in Abschnitt 5.6.2 beschrieben, muss diese entweder per `throws` in der Methodensignatur definiert sein oder aber durch einen `catch`-Block behandelt werden. Des Weiteren definiert die Methode `openAndProcess()` in ihrer Signatur keine Exception, sodass wir dies aus Kompatibilitätsgründen für diese öffentliche Methode auch nicht ändern dürfen. Natürlich sollten wir aber belegte Systemressourcen freigeben. In unserem Beispiel geschieht dies durch den Methodenaufruf von `closeServerSocket()` im `finally`-Block in jedem Fall.

Schritt 3: Variante B: Nutzung von ARM

Das mit JDK 7 eingeführte und in Abschnitt 5.6.3 beschriebene ***Automatic Resource Management*** (***ARM***) erlaubt es, das Ganze eleganter und kürzer zu implementieren. Praktischerweise muss man sich dann nicht mehr selbst um die Freigabe von Ressourcen kümmern, sondern nur die Syntax mit den runden Klammer wie folgt nutzen:

```
public static void openAndProcess()
{
   final int AUTO_ALLOCATE_PORT = 0;
   try (final ServerSocket serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT))
   {
      handleIncomingConnections(serverSocket);
      return true;
   }
   catch (final IOException ex)
   {
      // Die Freigabe der Ressource erfolgt hier durch ARM. Die Exception darf
      // nicht propagiert werden, weil sie nicht in der Signatur definiert ist.

      return false;
   }
}
```

5.6.5 Assertions

Nachdem wir uns nun recht intensiv mit Exceptions, vor allem im Zusammenhang mit I/O und Ressourcenmanagement, beschäftigt haben, möchte ich nachfolgend auf Assertions (Zusicherungen) eingehen. Diese dienen dazu, erwartete Zustände abzusichern. Zur Formulierung einer solchen Zusicherung wird das Schlüsselwort `assert` sowie eine boolesche Bedingung angegeben. Wird diese zu `false` evaluiert, so wird ein `java.lang.AssertionError` ausgelöst. Das ist ein Subtyp von `java.lang.Throwable`. Demzufolge können `AssertionError`s zwar durch einen `try-catch`-Block bearbeitet werden, allerdings ist dieses Vorgehen selten sinnvoll, da man einen Programmfehler aufdecken und nicht verschweigen möchte. Dabei sollte man zusätzlich wissen, dass die ***Verarbeitung von Assertions für die JVM standardmäßig deaktiviert ist und explizit aktiviert werden muss***, damit die Zusicherungen tatsächlich ausgewertet werden.

Assertions am Beispiel

Aus einem Eingabestring `versions` sollen die zwei Versionsinformationen `majorVersion` und `minorVersion` extrahiert werden. Jeder Bestandteil der Versionsinformation muss mindestens aus einem Zeichen bestehen. Dazu wird die Länge mithilfe von Assertions geprüft. Die aus der Eingabe korrespondierenden Teilstrings ermitteln wir mithilfe eines `StringTokenizer`s und dessen Methode `nextToken()`. Um die spätere Verarbeitung und gegebenenfalls eine Umwandlung in Zahlen zu vereinfachen, werden Leerzeichen durch einen Aufruf von `trim()` abgeschnitten.

```
public static void main(final String[] args)
{
    // ACHTUNG: fehlendes Token Minor-Version
    final String versions = "12. ";
    final StringTokenizer tokenizer = new StringTokenizer(versions, ".");

    final int tokenCount = tokenizer.countTokens();
    if (tokenCount > 1)
    {
        // Versionen auslesen
        final String majorVersion = tokenizer.nextToken().trim();
        final String minorVersion = tokenizer.nextToken().trim();

        // Sicherstellen, dass Tokens einen Wert enthalten
        assert !majorVersion.isEmpty();
        assert !minorVersion.isEmpty();

        System.out.println("Major: '" + majorVersion + "'");
        System.out.println("Minor: '" + minorVersion + "'");
        System.out.println("#Tokens: '" + tokenCount + "'");
    }
    else
    {
       System.err.println("Unexpected version number format => no '.' found");
    }
}
```

Listing 5.36 *Ausführbar als* '**ASSERTEXAMPLE**'

Im Beispiel enthält der zu verarbeitende Eingabewert bewusst einen Leerstring als zweite Versionsinformation, um die Funktionsweise von Assertions zu demonstrieren. Deshalb sollte der String `minorVersion` in diesem speziellen Fall eine Länge von 0 besitzen und damit die Zusicherung `!minorVersion.isEmpty()` verletzen. Demnach erwartet man bei der Eingabe von `"12. "` ein Fehlschlagen der Assertion und als Folge einen Programmabbruch mit einem `AssertionError`. Allerdings läuft das Programm ASSERTEXAMPLE ohne Fehler. Zunächst ist das möglicherweise verwunderlich, doch die folgende Konsolenausgabe bringt uns auf die richtige Spur:

```
Major: '12'
Minor: ''
#Tokens: '2'
```

Mit dem Hinweis der Einleitung im Hinterkopf erinnern wir uns vermutlich daran, dass ***Assertions standardmäßig deaktiviert sind, aber explizit angeschaltet werden können.*** Um die zwei Assertions zu aktivieren, existieren die JVM-Parameter `-ea` bzw. `-enableassertions`. Mit `-da` bzw. `-disableassertions` deaktiviert man Assertions. Wird das Programm mit dem Parameter `-ea` gestartet, so schlägt die Zusicherung fehl und es kommt zu folgender Konsolenausgabe:

```
Exception in thread "main" java.lang.AssertionError
```

Ein solcher `AssertionError` ohne Kontextinformationen ist nicht aussagekräftig, vielmehr sollte ein Hinweis zur Fehlerursache erfolgen. Dazu kann man die im folgenden Listing gezeigten Varianten nutzen:

```
public static void main(final String[] args)
{
    // ACHTUNG: fehlendes Token Minor-Version
    final String versions = "12. ";
    final StringTokenizer tokenizer = new StringTokenizer(versions, ".");

    final int tokenCount = tokenizer.countTokens();
    if (tokenCount > 1)
    {
        final String majorVersion = tokenizer.nextToken().trim();
        final String minorVersion = tokenizer.nextToken().trim();

        // Sicherstellen, dass Tokens einen Wert enthalten
        assert !majorVersion.isEmpty() : "Major-Version must not be empty";
        assert !minorVersion.isEmpty() : buildWarnMessage("Minor-Version");

        System.out.println("Major: '" + majorVersion + "'");
        System.out.println("Minor: '" + minorVersion + "'");
        System.out.println("#Tokens: '" + tokenCount + "'");
    }
    else
    {
        System.err.println("Unexpected version number format => no '.' found");
    }
}

private static String buildWarnMessage(final String versionName)
{
    return versionName + " must not be empty!";
}
```

Listing 5.37 *Ausführbar als* '**ASSERTEXAMPLEWITHMETHODS**'

Beim Ausführen des Programms ASSERTEXAMPLEWITHMETHODS erhalten wir folgende Ausgabe auf der Konsole:

```
Exception in thread "main" java.lang.AssertionError: Minor-Version must not be
      empty!
```

Tipps zum Einsatz von Assertions

Weil es möglich ist, Assertions zur Laufzeit an- und abzuschalten, kann man sie als ein »löchriges« Sicherheitsnetz auffassen. ***Dies kann problematisch sein, wenn Assertions zur Absicherung der falschen Dinge eingesetzt werden, etwa zur Sicherstellung der Programmkonsistenz bei ungültigen Eingabewerten öffentlicher Methoden.*** Aus diesen Vorbemerkungen ergeben sich folgende Hinweise zum Einsatz von Assertions.

Assertions und Eingabeparameter Da Assertions jederzeit ohne unsere Kontrolle ein- bzw. ausgeschaltet werden können, stellen sie kein geeignetes Mittel dar, Eingabeparameter öffentlicher Methoden zu prüfen. Zur Prüfung von Werten innerhalb privater Methoden ist der Einsatz von Assertions vertretbar, da hier der eigene Objektzustand eigentlich immer gesichert sein sollte, indem bereits zuvor durch Parameterprüfungen in öffentlichen Methoden Fehleingaben verhindert wurden. Nichtsdestotrotz bevorzuge ich selbst für private Methoden, mit Exceptions statt Assertions auf Fehleingaben zu reagieren, sofern dort eine Prüfung angebracht ist.

Assertions für Tests Die JLS schlägt vor, Assertions für Situationen einzusetzen, die »niemals« auftreten können bzw. sollen, etwa in einem `switch` für den »unmöglichen« `default`-Fall, wenn alle gültigen Eingabewerte durch ein `case` abgedeckt sind. Auch hier bevorzuge ich das Auslösen einer Exception. Der Grund ist einfach: Für `switch` wird durch ein fehlendes `break` schnell ein Fall-Through und ein Fehler provoziert. Dies fällt bei deaktivierten Assertions nicht sofort als Programmfehler auf.

Assertions als semantischer Kommentar Mit Assertions lassen sich Bedingungen in den Sourcecode einbringen, die wie ein Kommentar zu lesen sind, aber zusätzlich die Validierung gewisser Zusicherungen erlauben. Dadurch können Annahmen klarer als lediglich mit einem reinen Kommentar formuliert werden.

Vorsicht vor Seiteneffekten Die Ausführung von Assertions sollte keine Änderung an Variablen oder Attributen vornehmen, da diese je nach Aktivierung abgearbeitet werden oder eben nicht. Es ist zudem empfehlenswert, dass die in Assertions aufgerufenen Methoden keine ***Seiteneffekte***[20] verursachen. Nehmen wir zur Verdeutlichung an, die Variable `tokenCount` wäre nicht mehr lokal, sondern als Attribut definiert. Nachfolgend wird gezeigt, wie der Aufruf von `checkLength()` zu einem unerwarteten Seiteneffekt führt:

```
private static boolean checkLength(final String version)
{
    tokenCount = 7;  // Seiteneffekt: Attribut wird geändert
    return version.length() > 0;
}
```

[20] Darunter versteht man unerwartete Modifikationen am eigenen Objektzustand (oder noch schlimmer am Objektzustand eines anderen Objekts).

5.7 Weitere Neuerungen in JDK 9, 10 und 11

Nachfolgend möchte ich auf einige relevante Neuerungen aus Java 9, 10 und 11 eingehen. Dabei beginnen wir mit einer Erweiterung bei der `@Deprecated`-Annotation. Danach lernen wir die Syntaxerweiterung `var` kennen und schließlich schauen wir uns die Neuerungen bei der Versionsverarbeitung an.

5.7.1 Erweiterung der `@Deprecated`-Annotation in JDK 9

Die `@Deprecated`-Annotation dient bekanntlich zum Markieren von veraltetem Sourcecode und besaß bis einschließlich JDK 8 keine Parameter. Das ändert sich mit JDK 9: Die `@Deprecated`-Annotation wurde um die zwei Parameter `since` und `forRemoval` erweitert. Das war nötig, weil veraltete Funktionalität nun zum Teil aus dem JDK entfernt wird, statt sie – wie bislang für Java üblich – aus Rückwärtskompatibilitätsgründen ewig beizubehalten. Das folgende Beispiel zeigt eine Anwendung, wie sie aus dem JDK stammen könnte:

```
@Deprecated(since = "1.5", forRemoval = true)
```

Mithilfe der Parameter kann man für veralteten Sourcecode angeben, in welcher Version (`since`) dieser mit der Markierung als `@Deprecated` versehen wurde und ob der Wunsch besteht, die markierten Sourcecode-Teile in zukünftigen Versionen zu entfernen (`forRemoval`). Weil beide Parameter Defaultwerte besitzen (`since` = "" und `forRemoval` = `false`), sind die Angaben optional und können sogar ganz entfallen.

Die Erweiterung der `@Deprecated`-Annotation lässt sich selbstverständlich auch für eigenen Sourcecode nutzen, wodurch sich beschreiben lässt, dass gewisse Funktionalitäten für die Zukunft nicht mehr angeboten werden sollen. Darüber hinaus empfiehlt es sich, in einem Javadoc-Kommentar das `@deprecated`-Tag zu verwenden und dort den Grund der Deprecation und eine empfohlene Alternative aufzuführen. Nachfolgend ist dies exemplarisch für eine veraltete eigene Methode `someOldMethod()` gezeigt:

```
/**
 * @deprecated this method is replaced by someNewMethod()
  * ({@link #someNewMethod()}) which is more stable
 */
@Deprecated(since = "7.2", forRemoval = true)
private static void someOldMethod()
{
    // ...
}
```

5.7.2 Syntaxerweiterung `var` in JDK 10 und 11

Seit Java 10 gibt es die Local Variable Type Inference, besser bekannt als `var`, als Syntaxerweiterung. Diese erlaubt es, auf die explizite Typangabe auf der linken Seite einer Variablendefinition zu verzichten, sofern sich der konkrete Typ für eine lokale Variable anhand der Definition auf der rechten Seite der Zuweisung vom Compiler ermitteln lässt.

Einführende Beispiele

Schauen wir uns einige einführende Beispiele für die Kurzschreibweise mit `var` für Variablendefinitionen an:

```
var name = "Peter";                // var => String
var chars = name.toCharArray();    // var => char[]

var mike = new Person("Mike", 47); // var => Person
var hash = mike.hashCode();        // var => int
```

Insbesondere im Zusammenhang mit generischen Containern spielt die Local Variable Type Inference ihre Vorteile aus:

```
// var => ArrayList<String>
var names = new ArrayList<String>();
names.add("Tim");
names.add("Tom");
names.add("Jerry");

// var => Map<String, Long>
var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L,
                              "Michael", 47L, "Max", 25L);
```

Vor allem wenn die Typangaben mehrere generische Parameter umfassen, kann `var` den Sourcecode deutlich kürzer und mitunter lesbarer machen. Betrachten wir als Beispiel eine Verschachtelung von Typen analog zu den folgenden:

- `Set<Map.Entry<String, Long>>`
- `Map<Character, Set<Map.Entry<String, Long>>>`

In solchen Fällen spart `var` einiges an Schreibarbeit – zusätzlich erfolgt hier noch ein statischer Import verschiedener Kollektoren, um die Lesbarkeit zu steigern:

```
// var => Set<Map.Entry<String, Long»
var entries = personAgeMapping.entrySet();

// var => Map<Character, Set<Map.Entry<String, Long>>>
var filteredPersons = personAgeMapping.entrySet().stream().
                                       collect(groupingBy(firstChar,
                                               filtering(isAdult, toSet())));
```

Im Beispiel werden zur Beschreibung der Gruppierung und zur Filterung folgende zwei Lambdas (vgl. Kapitel 4) genutzt, worauf ich gleich nochmals genauer eingehe:

```
Function<Map.Entry<String, Long>, Character> firstChar =
                                             entry -> entry.getKey().charAt(0);

Predicate<Map.Entry<String, Long>> isAdult = entry -> entry.getValue() >= 18;
```

Hilfestellung in IDEs

So angenehm die Kurzschreibweise in der Regel auch ist, so wünschenswert ist manchmal eine Expansion in oder ein Hinweis auf den konkreten Typ. In beiden Fällen ist der intelligente Tooltip in Eclipse hilfreich, wie es folgende Abbildung 5-4 zeigt.

Abbildung 5-4 *Hilfestellung zu* `var` *in Eclipse*

Vermutlich eher selten besteht das Bedürfnis, doch wieder den konkreten Typ statt `var` zu nutzen. Praktischerweise existieren Quick Fixes in den gebräuchlichen IDEs, um zwischen konkretem Typ und `var` leicht hin- und herzuwechseln.

Lambda-Ausdrücke und `var`

Eine Kleinigkeit sollten wir noch betrachten: Im vorherigen Beispiel kommen beim Aufbereiten der Map folgende zwei Lambdas zum Einsatz, um die Funktionalität zu realisieren:

```
Function<Map.Entry<String, Long>, Character> firstChar =
                                             entry -> entry.getKey().charAt(0);

Predicate<Map.Entry<String, Long>> isAdult = entry -> entry.getValue() >= 18;
```

Wäre es nicht wünschenswert, auch hier die Typangabe mit `var` abzukürzen? Eigentlich ja! Warum dies nicht geht, erkläre ich im Folgenden.

Der Compiler kann allein auf Basis dieser Lambdas den konkreten Typ nicht ermitteln. Somit ist keine Umwandlung in `var` möglich, sondern führt zur Fehlermeldung »`lambda expression needs an explicit target-type`«. Wollte man `var` trotzdem nutzen, so müsste man einen passenden Cast einfügen:

```
var isAdultVar =
    (Predicate<Map.Entry<String, Long>>) entry -> entry.getValue() >= 18;
```

Insgesamt sieht man, dass `var` für Lambda-Ausdrücke eher ungeeignet ist. Das ist insofern schade, weil hier einige Schreibarbeit gespart werden könnte.

Beschränkungen

Rekapitulieren wir kurz: `var` ist für lokale Variablen gedacht, die direkt initialisiert werden. Damit ist es im Speziellen auch für Variablen in `for`-Schleifen und ARM (auch try-with-resources genannt) geeignet. Darüber hinaus scheint mitunter wünschenswert, `var` zur Deklaration von Attributen, Parametern oder Rückgabetypen nutzen zu können. Das ist ebenso wie für Lambdas jedoch nicht möglich, weil sich der Typ vom Compiler nicht eindeutig ermitteln lässt.

Restriktion auf exakten Typ Beim Einsatz von `var` sollte man wissen, dass immer der exakte Typ verwendet wird und nicht ein Basistyp, wie man es für Collections getreu dem Paradigma »program against interfaces« vielfach nutzt. Beachten Sie bitte, dass im Folgenden die Variable `names` deshalb nicht vom Typ `List<String>` ist, sondern vom Typ `ArrayList<String>`:

```
// var => ArrayList<String>
var names = new ArrayList<String>();
names = new LinkedList<String>(); // Compile Error
```

Versucht man, eine Instanz einer `LinkedList<String>` an die Variable `names` zuzuweisen, so erzeugt dies die Fehlermeldung: »`type mismatch: cannot convert from LinkedList<String> to ArrayList<String>`«. Der Grund sollte Ihnen mittlerweile bekannt sein: Es kommt zum Fehler, weil der durch `var` repräsentierte Typ `ArrayList<String>` und eben nicht `List<String>` ist, wie man es vielleicht erwarten würde.

Weitere syntaktische Besonderheiten Es gibt weitere Dinge, die zu Kompilierfehlern führen. Das sind eine fehlende Wertangabe und auch eine fehlende Typangabe bei direkten Array-Initialisierungen:

```
var justDeclaration;                // keine Wertangabe / Definition
var numbers = {0, 1, 2};            // fehlende Typangabe
```

Fallstrick

Auf einen Fallstrick beim Einsatz von `var` möchte ich unbedingt noch eingehen: Manchmal ist man versucht, ohne viel nachzudenken, die Typangabe auf der linken Seite direkt mit `var` zu ersetzen. Betrachten wir dazu ein harmlos wirkendes Beispiel einer auf String typisierten Liste:

```
List<String> names = new ArrayList<>();
names.add("Expected");
// names.add(42); // Compile error
```

Hier kommt auf der rechten Seite der Diamond Operator zum Einsatz, der es erlaubt, auf die explizite Angabe des generischen Typs zu verzichten. Das ist möglich, weil

dieser aus der linken Seite der Variablendeklaration, genauer der Typangabe, hergeleitet werden kann. Was passiert, wenn wir nun die Angabe von `List<String>` durch `var` ersetzen und das Kommentarzeichen vor dem zweiten Aufruf von `add()` löschen?

```
var mixedContent = new ArrayList<>();
mixedContent.add("Strange with var");
mixedContent.add(42);
```

Kompiliert und funktioniert das? Und wenn ja, was ist daran problematisch? Tatsächlich produziert das Ganze keinen Kompilierfehler. Wie kommt das? Aufgrund des Diamond Operators bzw. der nicht vorhandenen Typangabe stehen dem Compiler nicht ausreichend Typinformationen zur Verfügung: Deswegen wird als Fallback `java.lang.Object` als generischer Parameter genutzt und aus der zuvor auf String typisierten Liste wird – wohl eher unerwartet – eine `ArrayList<Object>`!

Die gezeigte Modifikation stellt potenziell einen Flüchtigkeitsfehler dar, der jedoch schwerwiegende Folgen haben kann. Genau deshalb steht der Umwandlungs-Quick-Fix in den IDEs nur dann zur Verfügung, wenn auf der linken Seite der exakte Typ angegeben wurde. In der Praxis ist dies aber durch die oftmals sinnvolle Designregel, gegen Interfaces zu programmieren, vor allem für Collections eher selten der Fall: Regelkonform arbeitet man bei der Variablendeklaration vielfach mit einem Interface statt mit einer konkreten Klasse. Bevor Sie etwas übereifrig dann selbst Hand anlegen und `var` nutzen, fügen Sie bitte die Typangabe auf der rechten Seite ein, um Probleme durch fehlende Typsicherheit direkt zu umgehen.

Local Variable Type Inference für Parameter von Lambdas in JDK 11

In Java 11 wurde eine minimale Anpassung bezüglich der Syntax vorgenommen: Bis einschließlich Java 10 konnte man in einem Lambda-Ausdruck bei den Parametern entweder alle Typen angeben oder alle weglassen:

```
IntFunction<Integer> doubleItTyped = (final int x) -> x * 2;
IntFunction<Integer> doubleItNoType = x -> x * 2;
```

Die Verwendung von `var` war dort nicht möglich. Das wurde mit Java 11 geändert, sodass nun auch folgender Lambda-Ausdruck korrekt ist:

```
(var x, var y) -> x.doSomething(y)
```

Warum sollte aber diese Variante überhaupt nützlich sein, wo man doch vollständig auf die Typangabe verzichten kann? Eine berechtigte Frage! Die Antwort ergibt sich aus dem Wunsch, zwar keinen Typ explizit angeben zu müssen, jedoch etwa den Parameter trotzdem `final` definieren oder Annotations hinzufügen zu können. Das geht natürlich nicht, wenn die Parameterangabe vollständig ohne Typ genutzt wird. Hier erlaubt `var`, auf die explizite Typangabe zu verzichten und trotzdem weiter gehende Informationen bereitstellen zu können. Das wird nachfolgend durch eine Annotation `@NonNull` verdeutlicht:

```
Function<String, String> trimmer = (@NonNull var str) -> str.trim();
```

Mit dieser Anpassung in der Syntax wird mehr Konsistenz erreicht und dies ermöglicht – wie bereits im Beispiel demonstriert – die Angabe zusätzlicher Infos zu einem Parameter.

5.7.3 Versionsverarbeitung mit JDK 9 und 10

In der Vergangenheit war die Versionierung der JDKs mitunter ein wenig verwirrend: Allein anhand der Versionsnummer fiel es extrem schwer, zwischen Minor Releases und Security Updates zu unterscheiden.[21] Auch waren in der Nummerierung die Varianten 8 und 1.8 gebräuchlich.[22] Dies wurde mit JDK 9 durch folgendes Versionsschema `MAJOR.MINOR.SECURITY` vereinheitlicht. Insbesondere wurde der Ballast der führenden 1 sowie des `u` für Updates nun aus dem Versionsschema entfernt. Zur Verarbeitung wurde die Klasse `java.lang.Runtime.Version` ins JDK integriert. Diese hilft auch beim Parsing von Versionsnummern.

Nachdem mit Java 9 endlich ein sinnvolles, nachvollziehbares Versionsschema existierte, wurde diese begrüßenswerte Neuerung leider bereits mit Java 10 und dem damit eingeführten halbjährlichen Release-Zyklus wieder über der Haufen geworfen. Durch die neue Release-Politik spricht man nicht mehr von Major- und Minor-Version sowie Security-Patch mit dem Format MAJOR.MINOR.SECURITY, sondern von Feature, Interim, Update und Patch mit dem Format FEATURE.INTERIM.UPDATE.

Beispiel

Zugriff erhält man über die Klasse `Runtime.Version` und deren Methode `version()`. Die Klasse `Version` bringt eine Methode `parse()` sowie verschiedene Methoden zum Zugriff auf die einzelnen Versionsbestandteile mit. Aufgrund der geänderten Release-Politik sind die Methoden `major()`, `minor()` und `security()` seit Java 10 deprecated und rufen die Methoden `feature()` bzw. `interim()` sowie `update()` auf.

Betrachten wir ein Beispiel zur Demonstration der Neuerungen:

```
var version = Runtime.version();

System.out.println(version);
System.out.println("Major: " + version.major());
System.out.println("Minor: " + version.minor());
System.out.println("Security: " + version.security());
System.out.println("Feature: " + version.feature());
System.out.println("Interim: " + version.interim());
System.out.println("Update: " + version.update());
```

[21] Für eine Unterscheidung, ob Security Fixes dazugekommen sind oder neue Features oder beides, war oftmals ein Blick in die Release Notes oder gar den Sourcecode erforderlich.

[22] Bei Updates findet man dann sogar Varianten wie z. B. `1.7.0_55-b13`, `JDK 7u55` oder `JDK 7 Update 55`.

Als Ausgabe erhält man für ein aktuelles Java-14-Release Folgendes:

```
version ==> 14.0.1+7
14.0.1+7
Major: 14
Minor: 0
Security: 1
Feature: 14
Interim: 0
Update: 1
```

Man sieht sehr schön, dass sowohl `major()` und `feature()` sowie `minor()` und `interim()` als auch `security()` und `update()` jeweils gleiche Werte liefern.

Allerdings wirkt das neue API durch die Vielzahl an Methoden schon ein wenig überladen und unhandlich.

5.8 Weiterführende Literatur

Es gibt viele Bücher zum Einstieg in Java. Manche beschreiben lediglich das API anhand einfacher Beispiele, ohne Fallstricke zu nennen. Für alle, die mehr wollen und Hintergründe kennenlernen möchten, empfehle ich folgende Bücher ausdrücklich:

- »**SCJP Sun Certified Programmer for Java 6 Study Guide**« von Kathy Sierra und Bert Bates [69]
  Ein sehr gelungenes Buch zum Thema SCJP-Zertifizierung zu Java 6. Auf der beiliegenden CD ist Wissenswertes zur Zertifizierung zum SCJP sowie zum SCJD (Sun Certified Java Developer) vorhanden.
- »**A Programmers's Guide to Java SCJP Certification**« von Khalid A. Mughal und Rolf W. Rasmussen [53]
  Dieses Buch bietet einen etwas formaleren und noch präziseren Einstieg in die SCJP-Zertifizierung als das Buch von Sierra und Bates.
- »**The Java Programming Language**« von Ken Arnold, James Gosling und David Holmes [2]
  Ein unglaublich gutes Buch über die Sprache Java, das es schafft, die Informationen detailreich und extrem präzise darzustellen. Es geht in vielen Punkten noch weiter auf die Feinheiten von Java ein als die zuvor genannten SCJP-Bücher.

Weiterführende Informationen zum Thema NIO finden Sie in folgendem Buch:

- »**Java NIO**« von Ron Hitchens [29]
  Dieses Buch beschreibt die Klassen des NIO-Frameworks und geht zudem genauer auf reguläre Ausdrücke sowie Character Sets ein.

II Bausteine stabiler Java-Applikationen

6 Das Collections-Framework

In diesem Kapitel beschreibe ich das Collections-Framework, das wichtige Datenstrukturen wie Listen, Mengen und Schlüssel-Wert-Abbildungen zur Verfügung stellt. Die Lektüre dieses Kapitels soll Ihnen helfen, ein gutes Verständnis für die Arbeitsweise der genannten Datenstrukturen und mögliche Besonderheiten oder Nebenwirkungen ihres Einsatzes zu entwickeln.

Abschnitt 6.1 beschreibt in der Praxis relevante Datenstrukturen und zeigt kurze Nutzungsbeispiele. Auf gebräuchliche Anwendungsfälle wie Suchen und Sortieren gehe ich in Abschnitt 6.2 ein. Diverse weitere Funktionalitäten werden durch die zwei Utility-Klassen `Collections` und `Arrays` bereitgestellt und in Abschnitt 6.3 beschrieben. Abschnitt 6.4 beschäftigt sich mit dem Zusammenspiel von Generics und Collections und zeigt, welche Dinge vor allem in Kombination mit Vererbung beachtet werden sollten. Darüber hinaus widmet sich Abschnitt 6.5 der Klasse `Optional<T>` zur Modellierung optionaler Werte. Abschließend werden in Abschnitt 6.6 einige Besonderheiten und Fallstricke in den Realisierungen des Collections-Frameworks dargestellt, deren Kenntnis dabei hilft, Fehler zu vermeiden.

Das Thema Datenstrukturen und Multithreading wird in diesem Kapitel nicht vertieft. Das gilt ebenso für Hinweise zur Optimierung durch die Wahl geeigneter Datenstrukturen für gewisse Einsatzkontexte. Auf Ersteres geht Abschnitt 11.1 ein. Letzteres wird in Kapitel 22 behandelt.

6.1 Datenstrukturen und Containerklassen

Im Collections-Framework werden Listen, Mengen und Schlüssel-Wert-Abbildungen durch sogenannte ***Containerklassen*** realisiert. Diese heißen so, weil sie Objekte anderer Klassen speichern und verwalten. Als Basis für die Containerklassen dienen die Interfaces `List<E>`, `Set<E>` bzw. `Map<K,V>` aus dem Package `java.util`.

Bevor wir uns konkret mit Datenstrukturen beschäftigen, möchte ich explizit auf eine Besonderheit hinweisen. Nur Arrays können Elemente eines beliebigen Typs speichern – insbesondere können nur sie direkt primitive Typen wie `byte`, `int` oder `double` enthalten. Alle im Folgenden vorgestellten Containerklassen speichern Objektreferenzen. Die Verwaltung primitiver Typen ist dort nur möglich, wenn diese in ein Wrapper-Objekt (wie `Byte`, `Integer` oder `Double`) umgewandelt werden. Durch das Auto-Boxing (vgl. Abschnitt 5.2.1) geschieht dies oft automatisch.

6.1.1 Wahl einer geeigneten Datenstruktur

Um Daten in eigenen Applikationen sinnvoll zu speichern und performant darauf zugreifen zu können, ist der Einsatz geeigneter Datenstrukturen wichtig. Das Collections-Framework stellt bereits eine qualitativ und funktional hochwertige Sammlung von Containerklassen bereit. Diese lassen sich grob in zwei disjunkte Ableitungshierarchien mit den Interfaces `Collection<E>` und `Map<K,V>` als Basis unterteilen. Muss für eine gegebene Aufgabenstellung eine geeignete Datenstruktur gefunden werden, so ist zunächst basierend auf den Anforderungen und dem zu lösenden Problem die Entscheidung zwischen Listen und Mengen mit dem Basisinterface `Collection<E>` sowie Schlüssel-Wert-Abbildungen mit dem Basisinterface `Map<K,V>` zu treffen. Anschließend gilt es, eine geeignete konkrete Realisierung zu finden. Dazu gebe ich nachfolgend einige Hinweise, wo man in der Ableitungshierarchie des Collections-Frameworks »abbiegen« sollte, wenn man auf der Suche nach einer passenden Datenstruktur ist.

Wahl einer Datenstruktur basierend auf dem Interface `Collection`

Für Sammlungen von Elementen mit dem Basistyp `E` kann man eine Implementierung des Interface `Collection<E>` wählen und muss sich dabei zwischen Listen und Mengen entscheiden. Für Daten, die eine Reihenfolge der Speicherung erfordern und auch (mehrfach) gleiche Einträge enthalten dürfen, setzen wir Realisierungen des Interface `List<E>` ein. Möchte man dagegen doppelte Einträge automatisch verhindern, so stellt eine Realisierung des Interface `Set<E>` die geeignete Wahl dar. Abbildung 6-1 zeigt die Typhierarchie von Listen und Mengen, wobei aus Gründen der Übersichtlichkeit die generische Definition nicht dargestellt wird.

Abbildung 6-1 `Collection`*-Hierarchie*

Wahl einer Datenstruktur basierend auf dem Interface `Map`

In der Praxis findet man diverse Anwendungsfälle, in denen man Abbildungen von Objekten auf andere Objekte realisieren muss. Man spricht hier von einem Mapping von Schlüsseln auf Werte. Dazu nutzt man sinnvollerweise das Interface `Map<K,V>`, wobei `K` dem Typ der Schlüssel und `V` demjenigen der Werte entspricht. Verschiedene Ausprägungen von Maps mit ihrer Typhierarchie, bestehend sowohl aus Klassen als auch aus weiteren, von `Map<K,V>` abgeleiteten Interfaces, zeigt Abbildung 6-2 (auch hier wieder vereinfachend ohne generische Typinformation).

Abbildung 6-2 *`Map`-Hierarchie*

Erweiterungen in JDK 5 und 6

Mit JDK 5 und 6 wurde das Collections-Framework erweitert. Unter anderem wurden folgende Interfaces neu eingefügt:

- `Queue<E>` – Durch das Interface `Queue<E>` wird eine sogenannte Warteschlange modelliert. Diese Datenstruktur ermöglicht das Einfügen von Datensätzen am Ende und die Entnahme vom Anfang – nach dem FIFO-Prinzip (First-In-First-Out).
- `Deque<E>` – Dieses Interface definiert die Funktionalität einer doppelseitigen Queue, die Einfüge- und Löschoperationen an beiden Enden erlaubt und zudem das `Queue<E>`-Interface erfüllt.

Mit JDK 6 wurden spezielle Implementierungen der Interfaces `SortedSet<E>` und `SortedMap<K,V>` eingeführt, die eine Sortierung mit einem hohen Grad an Parallelität kombinieren. Dies sind die Klassen `ConcurrentSkipListSet<K,V>` und `ConcurrentSkipListMap<K,V>` aus dem Package `java.util.concurrent`.

6.1.2 Arrays

Arrays sind Datenstrukturen, die in einem zusammenhängenden Speicherbereich entweder Werte eines primitiven Datentyps oder Objektreferenzen verwalten können. Das ist nachfolgend für 100 Zahlen vom Typ `int` und zwei Namen vom Typ `String` gezeigt, wobei Letzteres die Kurzschreibweise und syntaktische Besonderheit der direkten Initialisierung verwendet, bei der die Größe des Arrays automatisch vom Compiler durch die Anzahl der angegebenen Elemente bestimmt wird:

```
final int[] numbers = new int[100];                    // Definition ohne Daten
final String[] names1 = new String[] { "Tim", "Mike" }; // Normalschreibweise
final String[] names2 = { "Tim", "Mike" };              // Kurzschreibweise
```

Ein Array stellt lediglich einen einfachen Datenbehälter bereit, dessen Größe durch die Initialisierung vorgegeben ist und der keinerlei Containerfunktionalität bietet, d. h., es werden weder Zugriffsmethoden angeboten noch findet eine Datenkapselung statt. Diese Funktionalität muss bei Bedarf in einer nutzenden Applikation selbst programmiert werden. Folgendes Beispiel des Einlesens von Personendaten aus einer Datenbank verdeutlicht das Beschriebene, wobei initial Platz für 1.000 Elemente bereitgestellt wird.

```
// Initiale Größenvorgabe
Person[] persons = new Person[1000];

int index = 0;
while (morePersonsAvailableInDb())
{
    if (index == persons.length)
    {
        // Größenanpassung
        persons = Arrays.copyOf(persons, persons.length * 2);
    }
    person[index] = readPersonFromDb();
    index++;
}
```

Eigenschaften von Arrays

Betrachten wir mögliche Auswirkungen beim Einsatz von Arrays: Vorteilhaft ist, dass indizierte Zugriffe typsicher und maximal schnell möglich sind. Auch entsteht kein Overhead wie bei Listen, die gewisse Statusinformationen verwalten, Prüfungen vornehmen und Zugriffsmethoden auf Elemente bieten. Arrays eignen sich damit ganz speziell dann, wenn kaum oder sogar keine Containerfunktionalität, sondern hauptsächlich ein indizierter Zugriff benötigt wird. Auch ist es nur in Arrays möglich, Werte primitiver Typen direkt zu verwalten. Auf der anderen Seite besitzen Arrays gegenüber Listen folgende Einschränkungen:

- Bei der Konstruktion eines Arrays wird eine fixe Größe festgelegt, die das Fassungsvermögen (auch ***Kapazität*** genannt) bestimmt – für eine Größenänderung muss ein neues Array erzeugt werden. Eine sinnvolle Angabe der Kapazität ist jedoch nur dann möglich, wenn die Anzahl zu speichernder Datensätze bei der Konstruktion annähernd bekannt ist. Problematisch wird der Einsatz eines Arrays für den Fall, dass die Anzahl der zu speichernden Daten im Voraus schlecht schätzbar ist oder variiert, etwa bei Suchen.
- Anhand der Größe eines Arrays kann man keine Aussage darüber treffen, wie viele Elemente tatsächlich gespeichert sind. Die Metainformation über den ***Füllgrad*** des Arrays, d. h. die Anzahl der dort gespeicherten Elemente, lässt sich nur aufwendig durch Iterieren über alle Einträge und durch Vergleich des gespeicherten Werts mit einem speziellen Wert, der als Indikator »kein Eintrag« dient, ermitteln. Allerdings muss auch ein solcher spezieller Wert existieren (und darf nicht Bestandteil der erlaubten Werte sein). Häufig eignet sich dazu der Wert `-1`, `0` oder `null`. Eine solche Codierung ist jedoch nicht immer möglich.
- Das Vorhalten ungenutzter Kapazität führt zu einer Verschwendung von Speicherplatz. Dies ist in der Regel für kleinere Arrays (< 1.000 Elemente) vernachlässigbar. Für große Datenstrukturen (einige 100.000 Elemente) kann sich dies aber negativ auf den belegten sowie den restlichen verfügbaren Speicher auswirken.
- Ist die gewählte Größe zu gering, so lassen sich nicht alle gewünschten Daten speichern, da keine automatische Anpassung der Größe stattfindet. Dies muss selbst programmiert werden: Im vorherigen Beispiel haben wir dazu die Methode `Arrays.copyOf()` genutzt, wodurch ein neues, größeres Array angelegt und anschließend alle Elemente des ursprünglichen Arrays in das neue kopiert werden. Dieses Vorgehen ist recht umständlich – insbesondere wenn die Größenanpassung an mehreren Stellen im Sourcecode erforderlich wird. Es bietet sich dann an, diese Funktionalität in einer Methode zu realisieren.
- Ein Nachbau spezifischer Containerfunktionalität ist wenig sinnvoll und erhöht die Gefahr für Probleme, etwa durch veraltete Referenzen: Das gilt, wenn einige Programmteile Referenzen auf ein Array halten und nach einer Größenänderung und einem Kopiervorgang weiterhin mit diesen arbeiten, anstatt die neue Referenz zu verwenden. Eine Lösung ist, sämtliche Zugriffe auf das Array zu kapseln. Dann beginnt man aber mit dem Nachbau einer Datenstruktur ähnlich zu der bereits existierenden `ArrayList<E>`, was wenig sinnvoll ist.

Wir haben nun einige Beschränkungen von Arrays kennengelernt, die besonders dann zum Tragen kommen, wenn die Zusammensetzung der Elemente einer größeren Dynamik unterliegt. Häufig lässt sich für diese Fälle durch den Einsatz von Listen oder Mengen mit dem Basisinterface `Collection<E>`, das ich im folgenden Abschnitt vorstelle, eine vereinfachte Handhabung erreichen.

6.1.3 Das Interface `Collection`

Das Interface `Collection<E>` definiert die Basis für diverse Containerklassen, die das Interface `List<E>` bzw. `Set<E>` erfüllen und somit Listen bzw. Mengen repräsentieren. Wie bereits erwähnt, dienen Containerklassen dazu, mehrere Elemente zu speichern, auf diese zuzugreifen und gewisse Metainformationen (z. B. Anzahl gespeicherter Elemente) ermitteln zu können. Das Interface `Collection<E>` bietet *keinen* indizierten Zugriff, aber folgende Methoden:

- `int size()` – Ermittelt die Anzahl der in der Collection gespeicherten Elemente.
- `boolean isEmpty()` – Prüft, ob Elemente vorhanden sind.
- `boolean add(E element)` – Fügt ein Element zur Collection hinzu.
- `boolean addAll(Collection<? extends E> collection)` – Ist eine auf eine Menge bezogene, sogenannte ***Bulk-Operation*** (Massenoperation), die der Collection alle übergebenen Elemente hinzufügt.

Im Interface `Collection<E>` nutzen einige Methoden den Typparameter `Object` oder den Typplatzhalter '`?`' und sind daher nicht typsicher[1]:

- `boolean remove(Object object)` – Entfernt ein Element aus der Collection.
- `boolean removeAll(Collection<?> collection)` – Entfernt mehrere Elemente aus der Collection.
- `boolean contains(Object object)` – Prüft, ob das Element in der Collection enthalten ist.
- `boolean containsAll(Collection<?> collection)` – Prüft, ob alle angegebenen Elemente in der Collection enthalten sind.
- `boolean retainAll(Collection<?> collection)` – Behält alle Elemente einer Collection bei, die in der übergebenen Collection auch enthalten sind.

Hinweis: Typkürzel beim Einsatz von Generics

In den obigen Methodensignaturen haben wir folgende Typkürzel verwendet:

- `E` – Steht für den Typ der Elemente des Containers.
- `?` – Steht für einen unbekannten Typ.
- `? extends E` – Steht für einen unbekannten Typ, der entweder `E` oder ein Subtyp davon ist.

Die Buchstaben stellen lediglich Vereinbarungen dar und können beliebig gewählt werden. Ein konsistenter Einsatz erleichtert jedoch das Verständnis. Weitere gebräuchliche und in den folgenden Abschnitten genutzte Typkürzel sind:

- `T` – Steht für einen bestimmten Typ.
- `K` bzw. `V` – Steht bei Maps für den Typ des Schlüssels (`K` ⇒ key) bzw. des Werts (`V` ⇒ value).

[1] Ansonsten wäre dies für eine Enthaltensein-Prüfung eine zu starke Einschränkung gewesen.

Mengenoperationen auf Collections

Mit `contains(Object)` bzw. `containsAll(Collection<?>)` kann geprüft werden, ob ein oder mehrere gewünschte Elemente in einer Collection vorhanden sind. Man kann über `containsAll(Collection<?>)` bestimmen, ob eine Collection C eine Teilmenge einer Collection A ist. Mit `removeAll(Collection<?>)` lässt sich die Differenzmenge zweier Collections berechnen, indem z. B. aus einer Collection A alle Elemente einer anderen Collection B gelöscht werden. Mit `retainAll(Collection<?>)` berechnet man die Schnittmenge: Man behält in einer Collection A alle Elemente einer anderen Collection B. Zum leichteren Verständnis ist die Arbeitsweise in Abbildung 6-3 für die Collections A, B und C visualisiert.

Abbildung 6-3 *Mengenoperationen auf Collections*

Erweiterung im Interface `Collection<E>` in JDK 8

Seit JDK 8 kann man mit der Methode `removeIf(Predicate<T>)` aus einer Collection diejenigen Elemente entfernen, die einer übergebenen Bedingung entsprechen. Das wollen wir am Beispiel einer Liste von Namen kennenlernen. Aus dieser sollen diejenigen Einträge herausgelöscht werden, die einen Leereintrag darstellen:

```
public static void main(final String[] args)
{
   final List<String> names = new ArrayList<>();
   names.add("Max");
   names.add("");                  // Leereintrag
   names.add("Andy");
   names.add("   ");                // potenziell auch ein "Leereintrag"
   names.add("Stefan");

   names.removeIf(String::isEmpty) // Löschaktionen ausführen
   System.out.println(names);
}
```

Listing 6.1 *Ausführbar als* **'REMOVEIFEXAMPLE'**

Starten wir das Programm REMOVEIFEXAMPLE, so wird die beschriebene Löschoperation ausgeführt und es kommt zu folgender Ausgabe:

```
[Max, Andy,    , Stefan]
```

Wir sehen, dass der Whitespace-Eintrag in der Liste verblieben ist. Eine minimal komplexere Bedingung hilft, auch diesen Eintrag zu entfernen:

```
names.removeIf(str -> str.trim().isEmpty());
```

Die gezeigte Umsetzung birgt jedoch die Gefahr von `NullPointerException`s, wenn die Eingabewerte auch den Wert `null` enthalten können. Statt den Lambda etwas komplexer zu gestalten, wollen wir später in Abschnitt 6.1.5 als Alternative und zur Korrektur die mit JDK 8 im Interface `List<E>` neu eingeführte Methode `replaceAll(UnaryOperator<T>)` verwenden.

Iterationen mithilfe der Defaultmethode `forEach()` Ein gebräuchlicher und allgemeiner Anwendungsfall sind Iterationen über die Elemente einer Collection. Dazu betrachten wir im Anschluss das Interface `Iterator<T>`. Zuvor schauen wir uns mit der Defaultmethode `forEach(Consumer<? super T>)` eine in JDK 8 hinzugekommene Erweiterung im Interface `java.lang.Iterable<T>` an. Da dieses der Basistyp von `java.util.Collection<E>` ist, steht sie in allen Subtypen davon bereit.

Mit diesem Wissen kann man beispielsweise die Ausgabe einer Collection wie folgt mithilfe einer Methodenreferenz schreiben:

```
persons.forEach(System.out::println);
```

6.1.4 Das Interface `Iterator`

Alle Datenstrukturen, die das Interface `Collection<E>` erfüllen, bieten über die Methode `iterator()` Zugriff auf das Interface `java.util.Iterator<E>`, das einen sogenannten ***Iterator*** modelliert. Damit ist das Durchlaufen der Inhalte möglich, die in den Instanzen der Containerklassen des Collections-Frameworks gespeichert sind.

IDIOM: TRAVERSIERUNG VON COLLECTIONS MIT DEM INTERFACE ITERATOR

Zum Durchlaufen einer Collection mit Iteratoren definieren wir zunächst einen Datenbestand. Dabei nutzen wir den Trick, ein Array in eine Liste durch Aufruf der statischen Hilfsmethode `Arrays.asList(T...)` (vgl. Abschnitt 6.3.1) wandeln zu können. Das Ergebnis ist eine typisierte, aber insbesondere auch unmodifizierbare `List<E>`. Von dieser erhalten wir durch Aufruf von `iterator()` einen `Iterator<E>`. Mit dessen Methode `hasNext()` kann man ermitteln, ob noch Elemente zum Durchlaufen vorhanden sind. Ist dies der Fall, kann über die Methode `next()` auf das nächste Element zugegriffen werden. Damit ergibt sich folgendes Idiom zum Iterieren:

```
public static void main(final String[] args)
{
    final String[] textArray = { "Durchlauf", "mit", "Iterator" };
    final Collection<String> infoTexts = Arrays.asList(textArray);

    final Iterator<String> it = infoTexts.iterator();
    while (it.hasNext())
    {
        System.out.println(it.next());
    }
}
```

Listing 6.2 *Ausführbar als* '**ITERATIONEXAMPLE**'

Das Programm ITERATIONEXAMPLE gibt alle Elemente nacheinander aus:

```
Durchlauf
mit
Iterator
```

Löschfunktionalität im Interface `Iterator<E>`

Im Interface `Iterator<E>` ist auch die parameterlose Methode `remove()` definiert, die es erlaubt, das aktuelle, d. h. das zuvor über die Methode `next()` ermittelte Element zu löschen. Allerdings muss nicht jede Realisierung eines Iterators auch tatsächlich diese Löschfunktionalität unterstützen. In diesem Fall sollte ein Aufruf von `remove()` laut JDK-Kontrakt eine `UnsupportedOperationException` auslösen. Dieses Verhalten wird seit JDK 8 in Form einer Defaultmethode vorgegeben.

Die Definition der Methode `remove()` im Interface `Iterator<E>` wirkt überflüssig, weil doch bereits im Interface `Collection<E>` eine Methode `remove(Object)` existiert. Warum diese scheinbar doppelte Definition notwendig ist, zeige ich an einem Beispiel. Nehmen wir dazu an, aus einer Liste von Stringobjekten sollen diejenigen herausgefiltert werden, die mit einer speziellen Zeichenkette beginnen. Eine intuitive Realisierung mit den zuvor vorgestellten Methoden der Interfaces `Collection<E>` und `Iterator<E>` sieht etwa folgendermaßen aus:

```
private static void removeEntriesWithPrefix(final List<String> entries,
                                            final String prefix)
{
   final Iterator<String> it = entries.iterator();
   while (it.hasNext())
   {
      final String name = it.next();
      if (name.startsWith(prefix))
      {
          // ACHTUNG: remove() der Collection ist intuitiv, aber falsch
          entries.remove(name);
      }
   }
}
```

Schreiben wir ein Testprogramm, um die Funktionalität zu überprüfen. Wir definieren dazu einige Namen in einer Liste von Strings. Aus dieser sollen alle mit 'M' beginnenden Namen mit der zuvor definierten Methode gelöscht werden:

```
public static void main(final String[] args)
{
    final String[] names = { "Andy", "Carsten", "Clemens", "Mike", "Merten" };

    final List<String> namesList = new ArrayList<>();
    namesList.addAll(Arrays.asList(names));

    removeEntriesWithPrefix(namesList, "M");
    System.out.println(namesList);
}
```

Listing 6.3 *Ausführbar als* '**ITERATORCOLLECTIONREMOVEEXAMPLE**'

Man würde erwarten, dass als Ergebnis die Namen "Andy", "Carsten", "Clemens" in der Liste verbleiben und ausgegeben werden. Stattdessen kommt es zu einer `java.util.ConcurrentModificationException`:

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:781)
    at java.util.ArrayList$Itr.next(ArrayList.java:753)
   [...]
```

Eine derartige Exception deutet normalerweise auf Probleme und Veränderungen einer Datenstruktur bei nebenläufigen Zugriffen durch mehrere Threads hin. Etwas merkwürdig ist das schon, weil hier lediglich ein Thread läuft. ***Es ist demnach möglich, eine auf Multithreading-Probleme hindeutende Exception selbst in einfachen Programmen ohne Nebenläufigkeit zu provozieren.*** Gehen wir der Sache auf den Grund. Verursacht wird die Exception dadurch, dass in jeder Collection ein Modifikationszähler zum Schutz vor konkurrierenden Zugriffen genutzt wird. Jeder Iterator ermittelt dessen Wert zu Beginn seiner Iteration und vergleicht diesen bei jedem Aufruf von `next()` mit dem Startwert. Weicht dieser Wert ab, so wird eine `ConcurrentModificationException` ausgelöst. Dieses Verhalten der Iteratoren nennt man ***fail-fast***.

Mit diesem Hintergrundwissen wird die Fehlerursache klar: Der Aufruf der Methode `remove(Object)` auf der Datenstruktur `entries` führt zu einer Änderung des Modifikationszählers! Die einzig sichere Art, während einer Iteration Elemente aus einer Collection zu löschen, ist durch Aufruf der Methode `remove()` aus dem Interface `Iterator<E>`. Wir korrigieren unsere Methode wie folgt:

```
private static void removeEntriesWithPrefix(final List<String> entries,
                                            final String prefix)
{
    final Iterator<String> it = entries.iterator();
    while (it.hasNext())
    {
        if (it.next().startsWith(prefix))
            it.remove();    // KORREKT: Zugriff über remove() des Iterators
    }
}
```

Man erkennt, dass die Methode `remove()` aus dem Interface `Iterator<E>` keinen Parameter benötigt. Der Grund ist einfach: Sie löscht immer das zuvor über die Methode `next()` ermittelte Element.

Anhand der geführten Diskussion ist verständlich, warum die Methode `remove()` nicht nur im Interface `Collection<E>`, sondern auch im Interface `Iterator<E>` definiert ist. Bei dessen Nutzung gibt es noch einen kleinen Fallstrick: Vorsicht ist geboten, wenn man beispielsweise zwei aufeinander folgende Elemente löschen möchte. Intuitiv könnte man dies folgendermaßen umsetzen:

```
it.remove();
// FALSCH: it.next()-Aufruf fehlt
it.remove();
```

Das führt zu einer `IllegalStateException`, da, wie zuvor beschrieben, einem Aufruf von `remove()` immer ein Aufruf von `next()` vorangehen muss.

Achtung: Die Methode `remove()` im Interface `Iterator<E>`

Mein Verständnis von einem Iterator basiert auf dem Entwurfsmuster ITERATOR, das ich in Abschnitt 18.3.1 beschreibe. Laut dessen Definition sollte ein Iterator (lediglich) zum Durchlaufen einer Datenstruktur genutzt werden. Modifikationen der Datenstruktur waren dabei ursprünglich nicht vorgesehen. Man könnte daher die Existenz der Methode `remove()` im Interface `Iterator<E>` kritisieren. Aufgrund der Implementierungsentscheidung für die Fail-fast-Iteratoren wurde die Aufnahme dieser Methode in das Interface `Iterator<E>` allerdings notwendig, um Löschoperationen während einer Iteration zu erlauben.

Keine Methode `add()` Mit derselben Begründung könnte man für eine Methode `add(E)` im Interface `Iterator<E>` plädieren. Diese existiert aber aus gutem Grund nicht. Sie kann nicht angeboten werden, da das Einfügen eines Elements im Gegensatz zu dessen Entfernen nicht allgemeingültig auf Basis der aktuellen Position möglich ist: Für automatisch sortierende Container entspricht beispielsweise die momentane Position des Iterators in der Regel nicht der korrekten Einfügeposition in der Datenstruktur.

Erweiterungen im Interface `Iterator<T>` in JDK 8

Für Iterationen über die Elemente einer Collection bietet das Interface `Iterable<T>` die Methode `forEach()`. Etwas Analoges gibt es mit `forEachRemaining()` für das Interface `Iterator<E>`. Damit kann man beispielsweise die Ausgabe einer Collection wie folgt mithilfe einer Methodenreferenz und den Neuerungen aus Java 9 und 10 in Form der Collection-Factory-Methoden sowie dem Schlüsselwort `var` schreiben:

```
var persons = List.of(new Person("Tim"), new Person("Tom"), new Person("Mike"));

final Iterator<Person> it = persons.iterator();
it.forEachRemaining(System.out::println);
```

6.1.5 Listen und das Interface `List`

Unter einer Liste versteht man eine über ihre Position geordnete Folge von Elementen – dabei können auch identische Elemente mehrfach vorkommen. Das Collections-Framework definiert zur Beschreibung von Listen das Interface `List<E>`. Bekannte Implementierungen sind die Klassen `ArrayList<E>` und `LinkedList<E>` sowie `Vector<E>`. Das Interface `List<E>` ermöglicht einen indizierten Zugriff und erlaubt das Hinzufügen und Entfernen von Elementen – wobei es vereinzelte Ausnahmen gibt.[2]

Das Interface `List<E>`

Das Interface `List<E>` bildet die Basis für alle Listen und bietet *zusätzlich* zu den Methoden des Interface `Collection<E>` folgende indizierte, 0-basierte Zugriffe:

- `E get(int index)` – Ermittelt das Element der Liste an der Position `index`.
- `void add(int index, E element)` – Fügt das Element `element` an der Position `index` der Liste ein.
- `E set(int index, E element)` – Ersetzt das Element an der Position `index` der Liste durch das übergebene Element `element` und liefert das zuvor an dieser Position gespeicherte Element zurück.[3]
- `E remove(int index)` – Entfernt das Element an der Position `index` der Liste und liefert das gelöschte Element zurück.
- `int indexOf(Object object)` und
- `int lastIndexOf(Object object)` – Mit diesen Methoden wird die Position eines gesuchten Elements zurückgeliefert. Die Gleichheit zwischen dem Suchelement und den einzelnen Elementen der Liste wird mit der Methode `equals(Object)` überprüft. Die Suche startet dabei entweder am Anfang (`indexOf(Object)`) oder am Ende der Liste (`lastIndexOf(Object)`).

Folgendes Listing zeigt einige der obigen Methoden im Einsatz. Zunächst werden einer `ArrayList<E>` verschiedene Elemente am Ende und per Positionsangabe hinzugefügt, danach wird indiziert zugegriffen. Schlussendlich werden Löschoperationen per Index ausgeführt, wobei im letzteren Fall zuvor eine Suche mit `indexOf(Object)` erfolgt:

```
public static void main(final String[] args)
{
    // Erzeugen und Hinzufügen von Elementen
    final List<String> list = new ArrayList<>();
    list.add("First");
    list.add("Last");
```

[2] Die von `Collections.unmodifiableList(List<? extends T>)` erzeugte Spezialisierung einer Liste stellt lediglich eine unveränderliche Sicht dar. Ein Aufruf von verändernden Methoden führt zu `UnsupportedOperationExceptions`. Weitere Informationen finden Sie in Abschnitt 6.3.2.

[3] Es kommt zu einer `IndexOutOfBoundsException`, falls kein Element an dieser Position existiert. Für `add()` ist jedoch auch der nicht existierende Index `list.size()` erlaubt.

```
    list.add(1, "Middle");
    System.out.println("List: " + list);

    // Indizierter Zugriff
    System.out.println("3rd:  " + list.get(2));

    // Vorderstes Element löschen, "Last" mit indexOf() suchen und löschen
    list.remove(0);
    list.remove(list.indexOf("Last"));
    System.out.println("List: " + list);
}
```

Listing 6.4 *Ausführbar als* **'FIRSTLISTEXAMPLE'**

Startet man das Programm FIRSTLISTEXAMPLE, so kommt es zu folgender Ausgabe:

```
List: [First, Middle, Last]
3rd:  Last
List: [Middle]
```

Sublisten Die Methode `List<E> subList(int, int)` liefert einen Ausschnitt aus der Liste von Position `fromIndex` (einschließlich) bis `toIndex` (ausschließlich) und ermöglicht verschiedene Operationen auf Teillisten: Da die Rückgabe vom Typ `List<E>` ist, können tatsächlich alle Methoden des Interface `List<E>` aufgerufen werden. Dadurch kann man beispielsweise innerhalb eines gewissen Bereichs eine Suche durchführen oder diesen Bereich löschen. ***Dabei sollte man allerdings beachten, dass lediglich eine Sicht auf die ursprüngliche Liste geliefert wird***. Somit kommt es bei Veränderungen an der Teilliste auch zu Änderungen in der ursprünglichen Liste. Ändert man jedoch in der Originalliste, so kommt es zu einer `ConcurrentModificationException`. Folgendes Programm demonstriert dieses Verhalten:

```
public static void main(final String[] args)
{
    final List<String> original = new ArrayList<>(Arrays.asList("ABC", "DEF",
                                                                 "GHI", "JKL",
                                                                 "MNO", "PQR"));

    final List<String> first3 = original.subList(0, 3);

    first3.remove(1);
    printLists(original, first3);

    original.add("XXX");  // Führt später zur Exception
    printLists(original, first3);
}

private static void printLists(final List<String> original,
                               final List<String> subList)
{
    System.out.println("Original: " + original);
    System.out.println("Sublist:  " + subList);
}
```

Listing 6.5 *Ausführbar als* **'SUBLISTEXAMPLE'**

Startet man das Programm SUBLISTEXAMPLE, so kommt es zu folgender Ausgabe:

```
Original: [ABC, GHI, JKL, MNO, PQR]
Sublist:  [ABC, GHI]
Original: [ABC, GHI, JKL, MNO, PQR, XXX]
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$SubList.checkForComodification(ArrayList.java:1231)
    at java.util.ArrayList$SubList.listIterator(ArrayList.java:1091)
```

Das Interface `ListIterator<E>`

Alle Datenstrukturen, die das Interface `List<E>` erfüllen, bieten über die Methode `listIterator()` Zugriff auf einen speziellen Iterator vom Typ `ListIterator<E>`, der auf die Besonderheiten des indizierten Zugriffs angepasst wurde. Mit einem solchen Iterator ist das Durchlaufen einer Liste zusätzlich zu einem normalen Iterator auch in Rückwärtsrichtung möglich. Dazu dienen die beiden Methoden `hasPrevious()` sowie `previous()`. Außerdem kann man den Index des nächsten bzw. des vorherigen Elements über die Methoden `nextIndex()` bzw. `previousIndex()` abfragen.

Achtung: Die Methoden `set()` und `add()` im Interface `ListIterator`

Zusätzlich zur Methode `remove()` wurden in das Interface `ListIterator<E>` mit den Methoden `set(E)` und `add(E)` zwei weitere Daten verändernde Methoden eingeführt. Gemäß der Argumentation aus dem vorherigen Praxistipp kann man deren Existenz kritisieren. Wenn man Listen allerdings während einer Iteration manipulieren will, muss man diese modifizierenden Methoden im `ListIterator<E>` anbieten. Ohne sie würde aufgrund der Fail-fast-Eigenschaft ein Aufruf etwa der Methode `add(E)` aus dem Interface `Collection<E>` eine Exception auslösen.

Arbeitsweise der Klassen `ArrayList<E>` und `Vector<E>`

Die Klassen `ArrayList<E>` und `Vector<E>` verwenden zur Datenspeicherung intern Arrays und erweitern diese um Containerfunktionalität: Wächst die Anzahl zu verarbeitender Elemente, so wird automatisch dafür gesorgt, dass ausreichend Speicher zur Verfügung steht, indem bei Überschreiten der Größe des verwendeten Arrays ein neues, größeres Array angelegt wird. Zudem werden die Elemente des alten Arrays in das neue kopiert. Fortan wird das neue Array zur Speicherung der Elemente verwendet. Das alte Array ist daraufhin obsolet. Die Tatsache, dass intern mit Arrays gearbeitet wird, ist für den Benutzer transparent. Neben dieser Kapselung und der impliziten Größenanpassung, die oftmals ein entscheidender Vorteil gegenüber dem Einsatz von Arrays darstellt, bieten die Klassen `ArrayList<E>` und `Vector<E>` einige weitere Annehmlichkeiten einer Containerklasse: Ein Beispiel dafür ist die Unterscheidung zwischen der Füllstandsabfrage (`size()`), also der Anzahl der momentan gespeicherten Elemente, und den zur Verfügung stehenden Speicherplätzen (Kapazität), die die tatsächliche Größe des Arrays, also das momentane Fassungsvermögen, bestimmen. Der Aufbau

einer Array-basierten Liste wird in Abbildung 6-4 skizziert. Die Texte `Obj 1`, `Obj 2` usw. repräsentieren hier nicht das Objekt selbst, sondern die Referenz darauf.

Abbildung 6-4 *Array der Klasse* `ArrayList`

Zugriff auf ein Element an Position $index$ **– `get(index)`** Der Zugriff auf beliebige Elemente wird durch einen indexbasierten Array-Zugriff realisiert und ist dadurch sehr performant.

Element an letzter Position hinzufügen – `add(element)` Nehmen wir an, es ist (gerade) noch ausreichend Kapazität im Array vorhanden und es soll ein Element als letztes Element in die Datenstruktur eingefügt werden. In diesem Fall muss nur die Referenz in dem Array gespeichert werden:

Das zugrunde liegende Array ist jetzt komplett belegt und es ist kein Platz für ein weiteres Element vorhanden. Soll erneut ein Element hinzugefügt werden, muss als Folge das Array in seiner Größe angepasst werden:

Element an Position $index$ einfügen – `add(index, element)` Wird an einer Position $index$ ein Element eingefügt, so müssen als Folge alle Elemente mit einer Position $>= index$ nach hinten verschoben werden, um Platz für das neue Element zu schaffen. Mit $index = 0$ muss der Inhalt des gesamten Arrays verschoben werden. Reicht die Kapazität nicht mehr aus, so wird Speicher für ein neues Array alloziert und mit dem Inhalt aus dem alten Array gefüllt. Im Folgenden ist dies für das Einfügen an Position 8 und das Element NEW gezeigt:

Element an Position $index$ entfernen – `remove(index)` Wird an einer Position $index$ ein Element gelöscht, so müssen als Folge alle Elemente des Arrays mit einer Position $> index$ nach vorne verschoben werden. Im Extremfall mit $index = 0$ geschieht dies für das gesamte Array.

Größenanpassungen und Speicherverbrauch Beim Einfügen sorgen die Klassen `ArrayList<E>` und `Vector<E>` automatisch dafür, dass bei Bedarf die Größe schrittweise angepasst wird. Die Kopiervorgänge kosten mit zunehmender Größe immer mehr Zeit – das ist aber oftmals kaum messbar. Allerdings muss der Garbage Collector (vgl. Abschnitt 12.4) die obsoleten Arrays wieder wegräumen: Das ist Arbeit, die sich häufig verringern oder vermeiden lässt, indem man ***die erwartete Maximalgröße als Konstruktorparameter übergibt***. Jedoch ist die Wahl einer geeigneten Größe, wie bereits in Abschnitt 6.1.2 für Arrays erwähnt, nicht immer einfach oder gar möglich.

Bei zunehmendem Datenvolumen erhöht sich zudem die Wahrscheinlichkeit für Probleme durch die Speicherung als Array, weil immer ein zusammenhängender Speicherbereich benötigt wird. Je größer die Anzahl der Elemente wird, desto stärker wirkt sich dies aus. Zwei Probleme treten auf: Erstens kann im Extremfall eine Out-of-Memory-Situation eintreten, obwohl im Grunde noch genug Speicher vorhanden ist, aber kein zusammenhängender Speicherbereich der erforderlichen Größe bereitgestellt werden kann. Zweitens werden bei einem Vergrößerungsschritt beim Kopieren in ein neues, größeres Array temporär zwei Arrays gebraucht: einmal das alte und dann noch das neue Array. Wenn das alte Array z. B. 500 MB groß ist, dann benötigt man beim Kopieren vorübergehend ungefähr 1,25 GB.[4] Das kann ebenfalls zu einer Out-of-Memory-Situation führen, obwohl der Speicher eigentlich noch ausreicht – allerdings nur für eine Version des Arrays. Besonders verwirrend ist dies, wenn man als Programmierer nicht weiß, dass im Hintergrund eine Kopie angelegt wird.

[4] Das neu entstehende Array ist um die Hälfte größer als die ursprüngliche Größe, weil diese Vergrößerung derart in der Implementierung der `ArrayList<E>` programmiert ist. Im Beispiel wäre die neue Größe also 750 MB.

Achtung: Versteckte Memory Leaks und Abhilfemaßnahmen

Die Größe des Daten speichernden Arrays wird bei Einfügeoperationen bei Bedarf automatisch angepasst. Für das Entfernen von Elementen gilt dies allerdings nicht: Eine einmal bereitgestellte Kapazität wird dabei nicht wieder reduziert. Wurde einmalig viel Speicher alloziert, so erzeugt man ein verstecktes ***Memory Leak***, dadurch, dass die `ArrayList<E>` bzw. der `Vector<E>` immer noch den gesamten Speicher belegt, obwohl durch Löschoperationen mittlerweile viel weniger Elemente zu speichern sind.

Mithilfe der Methode `trimToSize()` kann man in einem solchen Fall dafür sorgen, dass das Array auf die benötigte Größe verkleinert wird. Der zuvor belegte Speicher ist anschließend unreferenziert und kann vom Garbage Collector freigeräumt werden. Der Applikation steht dieser Speicher daraufhin wieder zur Verfügung.

Unterschied zwischen `ArrayList<E>` und `Vector<E>` Die Klassen `ArrayList<E>` und `Vector<E>` unterscheiden sich in ihrer Arbeitsweise lediglich in einem Detail: In der Klasse `Vector<E>` sind die Methoden `synchronized` definiert, um bei konkurrierenden Zugriffen für Konsistenz der gespeicherten Daten zu sorgen. Häufig möchte man in einer Anwendung eine Kombination mehrerer Aufrufe schützen, sodass diese feingranulare Art der Synchronisierung nicht ausreichend für die benötigte Art von Thread-Sicherheit ist (vgl. Kapitel 10). Somit gibt es eher selten Anwendungsfälle für einen `Vector<E>` und in der Regel sollte man die ***`ArrayList<E>` bevorzugen***.

Arbeitsweise der Klasse `LinkedList<E>`

Die Klasse `LinkedList<E>` verwendet zur Speicherung von Elementen miteinander verbundene kleine Einheiten, sogenannte ***Knoten*** oder ***Nodes***. Jeder Knoten speichert sowohl eine Referenz auf die Daten als auch jeweils eine Referenz auf Vorgänger und Nachfolger. Dadurch wird eine Navigation vorwärts und rückwärts möglich. Diese Art der Speicherung führt dazu, dass die `LinkedList<E>` nur eine Größe (Anzahl der Knoten), aber keine Kapazität besitzt. Den schematischen Aufbau zeigt Abbildung 6-5, wobei `Obj 1`, `Obj 2` usw. Referenzen auf Objekte repräsentieren.

Abbildung 6-5 *Schematischer Aufbau eines Objekts der Klasse* `LinkedList`

Zugriff auf ein Element an Position $index$ – `get(index)` Durch die Organisation als verkettete Liste erfordert ein indizierter Zugriff auf ein Element an einer Position $index$ immer einen Durchlauf bis zu der gewünschten Stelle. Als Optimierung werden Zugriffe entweder vom Anfang oder Ende gestartet, abhängig davon, was davon näher bei dem Zielindex liegt. Im Gegensatz zur `ArrayList<E>` mit konstanter Zugriffszeit wächst daher die Zugriffszeit bei der `LinkedList<E>` linear mit der Anzahl der gespeicherten Elemente. Dies kann bei relativ großen Datenmengen (genaue Angaben sind schwierig, in der Regel aber mehr als 500.000 Einträge) negative Auswirkungen auf die Laufzeit haben. Das zeigt sich deutlich bei der Optimierung der Darstellung umfangreicher Datenmengen mit einer `JTable` in Abschnitt 22.4.

Element an letzter Position hinzufügen – `add(element)` Wenn ein Element hinten an letzter Position angefügt werden soll, so wird zunächst ein neuer Knoten erzeugt und danach mit dem bisher letzten Knoten verbunden.

Element an Position $index$ einfügen – `add(index, element)` Um ein Element an einer beliebigen Position einzufügen, wird ein neuer Knoten erzeugt. Zudem muss die Einfügeposition bestimmt werden, was linearen Aufwand durch eine Iteration durch die Liste bis zu der gewünschten Stelle erfordert. Schließlich sind noch einige Referenzen umzusetzen, um den neuen Knoten in die Liste einzufügen.

Element an Position $index$ entfernen – `remove(index)` Für eine Löschoperation sind lediglich Referenzanpassungen nötig, um den zu löschenden Knoten aus der Liste zu entfernen. Auch hier muss zunächst mit linearem Aufwand zur Position $index$ iteriert werden.

Größenanpassungen und Speicherverbrauch Die `LinkedList<E>` besitzt bezüglich des Speicherverbrauchs einige Vorteile gegenüber der `ArrayList<E>`. Erstens wird, abgesehen von einem gewissen Overhead, immer nur genau so viel Speicher für Elemente belegt, wie tatsächlich benötigt wird. Zweitens kann durch den Garbage Collector eine automatische Freigabe des Speichers an das System erfolgen, wenn Elemente gelöscht werden. Insbesondere bei großer Dynamik und temporär hohen Datenvolumina ist dies von Vorteil, da Speicherbereiche nicht unbenutzt belegt bleiben, wie dies beim Einsatz der `ArrayList<E>` der Fall sein kann. Allerdings wird das durch einen Nachteil erkauft: Während eine `ArrayList<E>` für jedes gespeicherte Element lediglich eine Objektreferenz hält, speichert jeder Knoten einer `LinkedList<E>` zusätzlich je eine Referenz auf den Vorgänger und auf den Nachfolger. Somit benötigt eine `LinkedList<E>` zur Verwaltung insgesamt mehr Speicher (in etwa Faktor drei) als eine `ArrayList<E>` mit gleicher Anzahl gespeicherter Elemente. Beachten Sie unbedingt, dass sich dieser Speicherbedarf nur auf den Verbrauch durch die Datenstruktur selbst bezieht und nicht auf den Speicherplatzbedarf der dort referenzierten Objekte, der in der Regel um Größenordnungen höher sein wird.

Erweiterungen im Interface `List<E>` in JDK 8

Auch das Interface `List<E>` wurde mit JDK 8 durch folgende zwei relevante Erweiterungen ergänzt:

- Sortierungen mit `sort()` – Seit Java 8 ist der Aufruf von `sort(Comparator<? super E>)` direkt auf der Instanz einer `List<E>` möglich.
- Modifikationen aller Elemente mit `replaceAll()` – Diese Methode ermöglicht es, für alle Elemente einer `Collection<E>` eine Aktion auszuführen.

Modifikationen aller Elemente mit `replaceAll()` Mithilfe dieser Methode wird jedes Element durch den Rückgabewert der Implementierung der Methode `apply(T)` des Interface `UnaryOperator<T>` ersetzt. Im Speziellen müssen nicht immer alle Elemente tatsächlich auch verändert werden. Das »`All`« im Namen bezieht sich also lediglich darauf, dass `apply(T)` für alle Elemente der Liste aufgerufen wird.

Schauen wir uns ein konkretes Anwendungsbeispiel an: Nehmen wir an, wir würden eine Liste von Eingabewerten erhalten. Oftmals entsprechen solche Eingaben nicht den Erwartungen und verstoßen gegen Regeln, so sind Einträge beispielsweise leer, bestehen nur aus Leerzeichen oder enthalten diese am Anfang oder Ende. All dies erschwert die weitere Bearbeitung. Die Grundlagen für eine Korrekturfunktionalität, die derartige Werte umwandelt oder herausfiltert, haben wir bereits kennengelernt. Wir müssen das Ganze nur noch geeignet kombinieren:

```
public static void main(final String[] args)
{
    final List<String> names = createDemoNames();

    // Spezialbehandlung von null-Werten
    final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" : str;
    names.replaceAll(mapNullToEmpty);

    // Leerzeichen abschneiden
    names.replaceAll(String::trim);

    // Leereinträge herausfiltern
    names.removeIf(String::isEmpty);

    System.out.println(names);
}

private static List<String> createDemoNames()
{
   final List<String> names = new ArrayList<>();
   names.add("  Max");
   names.add("");              // Leereintrag
   names.add("  Andy ");
   names.add("   ");           // potenziell auch ein "Leereintrag"
   names.add("Stefan  ");
   return names;
}
```

Listing 6.6 *Ausführbar als* '**REPLACEALLEXAMPLE**'

Mit dieser Erweiterung werden nicht nur potenzielle `null`-Einträge entfernt, sondern auch diejenigen Einträge, die Leerstrings oder lediglich Leerzeichen enthalten. Außerdem werden Leerzeichen am Anfang und Ende von Einträgen gelöscht. Die Ausgabe des Programms REPLACEALLEXAMPLE verdeutlicht dies:

```
[Max, Andy, Stefan]
```

6.1.6 Mengen und das Interface `Set`

Zum Einstieg in das Collections-Framework haben wir uns zunächst ausführlich mit Listen beschäftigt, kommen wir nun zu Mengen und dem Interface `Set<E>`. Das mathematische Konzept der Mengen besagt, dass diese keine Duplikate enthalten. Das Interface `Set<E>` basiert auf dem Interface `Collection<E>`. Im Gegensatz zum Interface `List<E>` sind im Interface `Set<E>` keine Methoden zusätzlich zu denen des Interface `Collection<E>` vorhanden – allerdings wird ein anderes Verhalten für die Methoden `add(E)` und `addAll(Collection<? extends E>)` vorgeschrieben. Dieser Unterschied zwischen `Set<E>` und dem zugrunde liegenden Interface `Collection<E>` ist nötig, um Duplikatfreiheit zu garantieren, selbst dann, wenn der Menge das gleiche Objekt mehrfach hinzugefügt wird.

Beispiel: Realisierungen von Mengen und ihre Besonderheiten

Für einen ersten Zugang zum Thema Mengen nutzen wir mit den Klassen `HashSet<E>` und `TreeSet<E>` zwei gebräuchliche Implementierungen des Interface `Set<E>` und füllen diese mit Werten vom Typ `String` und auch `StringBuilder`:

```
public static void main(final String[] args)
{
    fillAndExploreHashSet();
    fillAndExploreTreeSet();
}

private static void fillAndExploreHashSet()
{
    // String definiert hashCode() und equals()
    final Set<String> hashSet = new HashSet<>();
    addStringDemoData(hashSet);
    System.out.println(hashSet);

    // StringBuilder definiert selbst weder hashCode() noch equals()
    final Set<StringBuilder> hashSetSurprise = new HashSet<>();
    addStringBuilderDemoData(hashSetSurprise);
    System.out.println(hashSetSurprise);
}

private static void fillAndExploreTreeSet()
{
    // String implementiert Comparable
    final Set<String> treeSet = new TreeSet<>();
    addStringDemoData(treeSet);
    System.out.println(treeSet);
```

```
    // StringBuilder implementiert Comparable nicht
    final Set<StringBuilder> treeSetSurprise = new TreeSet<>();
    addStringBuilderDemoData(treeSetSurprise);
    System.out.println(treeSetSurprise);
}

private static void addStringDemoData(final Set<String> set)
{
    set.add("Hallo");
    set.add("Welt");
    set.add("Welt");
}

private static void addStringBuilderDemoData(final Set<StringBuilder> set)
{
    set.add(new StringBuilder("Hallo"));
    set.add(new StringBuilder("Welt"));
    set.add(new StringBuilder("Welt"));
}
```

Listing 6.7 *Ausführbar als* '**FIRSTSETEXAMPLE**'

Starten wir das Programm FIRSTSETEXAMPLE, so kommt es zu folgenden Ausgaben:

```
[Hallo, Welt]
[Welt, Welt, Hallo]
[Hallo, Welt]
Exception in thread "main" java.lang.ClassCastException: java.lang.StringBuilder
       cannot be cast to java.lang.Comparable
```

Das Beispiel zeigt, dass man zum sicheren Umgang mit den Mengen verstehen sollte, welche Mechanismen dort die Eindeutigkeit bewirken: Bei Strings funktioniert alles wie erwartet, für den Typ `StringBuilder` werden Duplikate nicht erkannt und für das `TreeSet<StringBuilder>` wird sogar eine Exception ausgelöst. Für zu speichernde Klassen ist eine korrekte und den jeweiligen Kontrakten folgende Implementierung einiger Methoden erforderlich. Für die Klasse `HashSet<E>` dient die Methode `hashCode()` zum Klassifizieren von Elementen in Form eines `int`-Zahlenwerts und die Methode `equals(Object)` zum Auffinden. Die Klasse `TreeSet<E>` nutzt dazu die Methoden `compareTo(T)` bzw. `compare(T, T)` aus den Interfaces `Comparable<T>` bzw. `Comparator<T>`. Abschnitt 6.1.9 geht auf das Zusammenspiel der relevanten Methoden im Detail ein. In den Abschnitten 6.1.7 und 6.1.8 werden zuvor sowohl die Grundlagen von hashbasierten Containern (zum Verständnis der Klasse `HashSet<E>`) als auch die Grundlagen automatisch sortierender Container (als Basis für die Klasse `TreeSet<E>`) vorgestellt.

Bevor wir tiefer in die Details abtauchen, wollen wir einfache Beispiele für `HashSet<E>` und `TreeSet<E>` betrachten, um ein besseres Gespür für die Arbeit mit Mengen zu erhalten.

Fallstrick: Fehlende Angabe eines Sortierkriteriums

Zur Kompilierzeit wird für ein `TreeSet<E>` nicht geprüft, ob dort nur Objekte gespeichert werden, die das Interface `Comparable<T>` erfüllen. Das ist durchaus berechtigt, da auch ein `Comparator<T>` zur Beschreibung des Sortierkriteriums dienen kann. Eine fehlende Angabe eines Sortierkriteriums macht sich daher erst zur Laufzeit beim Einfügen von Elementen durch eine `java.lang.ClassCastException` bemerkbar.

Die Klasse `HashSet<E>`

Die Klasse `HashSet<E>` ist eine Spezialisierung der abstrakten Klasse `AbstractSet<E>` und speichert Elemente ungeordnet in einem Hashcontainer (genauer: in einer später in Abschnitt 6.1.10 vorgestellten `HashMap<K,V>`). Dadurch sind die Operationen `add(E)`, `remove(Object)`, `contains(Object)` usw. performant.

Betrachten wir ein kurzes Beispiel, in dem die Werte `1` bis `3` in absteigender Reihenfolge in ein `HashSet<Integer>` eingefügt werden:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new HashSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);   // 1, 2, 3
}
```

Listing 6.8 *Ausführbar als* **'HASHSETSTORAGEEXAMPLE'**

Bei einem Blick auf die Ausgabe des Programms HASHSETSTORAGEEXAMPLE scheint ein `HashSet<Integer>` die eingefügten Werte zu sortieren:

```
Initial: [1, 2, 3]
```

Dies ist jedoch nur ein zufälliger Effekt. Dieser wird durch kleine Datenmengen und die gewählte Abbildung der zu speichernden Daten ausgelöst. Bei der Speicherung von Werten darf man sich bei einer *ungeordneten Menge*, wie sie von der Klasse `HashSet<E>` realisiert wird, *niemals* auf eine *definierte* Reihenfolge der Elemente verlassen. Dies wird deutlich, wenn man weitere Elemente einfügt, etwa die Werte `33`, `11` und `22`:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new HashSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);   // 1, 2, 3

    final Integer[] moreInts = new Integer[] { 33, 11, 22 };
    numberSet.addAll(Arrays.asList(moreInts));
    System.out.println("Add: " + numberSet);       // 1, 33, 2, 3, 22, 11
}
```

Listing 6.9 *Ausführbar als* **'HASHSETSTORAGEEXAMPLE2'**

Die Ausgabe des Programms HASHSETSTORAGEEXAMPLE2 wirkt zufällig, ist aber durch die Verteilung im Container verursacht:

```
Initial: [1, 2, 3]
Add: [1, 33, 2, 3, 22, 11]
```

Die Klasse TreeSet<E>

Mitunter benötigt man eine Ordnung der Elemente. Dann bietet sich der Einsatz der Klasse TreeSet<E> an, die das Interface SortedSet<E> implementiert. Die Sortierung der Elemente wird entweder durch das Interface Comparable<T> oder einen explizit im Konstruktor von TreeSet<E> übergebenen Comparator<T> festgelegt. Wir schauen uns ein ähnliches Beispiel an wie für die Klasse HashSet<E>:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet); // 1, 2, 3

    final Integer[] moreInts = new Integer[] { 33, 11, 22 };
    numberSet.addAll(Arrays.asList(moreInts));
    System.out.println("Add: " + numberSet); // 1, 2, 3, 11, 22, 33
}
```

Listing 6.10 *Ausführbar als* **'TREESETSTORAGEEXAMPLE'**

Weil im Konstruktor kein Comparator<T> übergeben wurde, basiert die Sortierung auf Comparable<T>, das von der zu speichernden Klasse Integer erfüllt wird und eine sogenannte natürliche Ordnung (realisiert über eine Vergleichsfunktion durch die Methode compareTo(T)) bereitstellt. Das lässt sich durch einen Start des Programms TREESETSTORAGEEXAMPLE nachvollziehen:

```
Initial: [1, 2, 3]
Add: [1, 2, 3, 11, 22, 33]
```

Das Interface SortedSet<E> Die Klasse TreeSet<E> bietet neben der Sortierung folgende nützliche Funktionalität aus dem Interface SortedSet<E>:

- E first() und E last() – Mit diesen beiden Methoden kann das erste bzw. letzte Element der Menge ermittelt werden.
- SortedSet<E> headSet(E toElement) – Liefert die Teilmenge der Elemente, die kleiner als das übergebene Element toElement sind.
- SortedSet<E> tailSet(E fromElement) – Liefert die Teilmenge der Elemente, die größer oder gleich dem übergebenen Element fromElement sind: Ein übergebenes Element ist im Gegensatz zu headSet(E) in der zurückgelieferten Menge enthalten, wenn es Bestandteil der Originalmenge war.

- `SortedSet<E> subSet(E fromElement, E toElement)` – Liefert die Teilmenge der Elemente, startend von inklusive `fromElement` bis exklusive `toElement`.

Wir bauen unser Beispiel ein wenig aus und integrieren zwei Änderungsaktionen, um zu zeigen, dass es sich bei den durch die obigen Methoden gelieferten Sets jeweils um Sichten handelt, die Änderungen propagieren:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1, 33, 11, 22 };
    final SortedSet<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);   // 1, 2, 3, 11, 22, 33

    System.out.println("first: " + numberSet.first());           // 1
    System.out.println("last: " + numberSet.last());             // 33

    final SortedSet<Integer> headSet  = numberSet.headSet(7);
    System.out.println("headSet:   " + headSet);                 // 1, 2, 3
    System.out.println("tailSet:   " + numberSet.tailSet(7));    // 11, 22, 33
    System.out.println("subSet:    " + numberSet.subSet(7, 23)); // 11, 22

    // Modifikationen an einem einzelnen Set
    headSet.remove(3);
    headSet.add(6);
    System.out.println("headSet:   " + headSet);   // 1, 2, 6
    System.out.println("numberSet: " + numberSet); // 1, 2, 6, 11, 22, 33
}
```

Listing 6.11 *Ausführbar als* **'TREESETSTORAGEEXAMPLE2'**

Die Ausgabe des Programms TREESETSTORAGEEXAMPLE2 demonstriert die Arbeitsweise der obigen Methoden:

```
Initial: [1, 2, 3, 11, 22, 33]
first: 1
last: 33
headSet:   [1, 2, 3]
tailSet:   [11, 22, 33]
subSet:    [11, 22]
headSet:   [1, 2, 6]
numberSet: [1, 2, 6, 11, 22, 33]
```

6.1.7 Grundlagen von hashbasierten Containern

Arrays und Listen haben einen in manchen Situationen unangenehmen Nachteil: Die Suche nach gespeicherten Daten und der Zugriff auf diese kann sehr aufwendig sein. Im Extremfall müssen alle enthaltenen Elemente betrachtet werden. Hashbasierte Container zeichnen sich dagegen dadurch aus, dass Suchen und diverse andere Operationen extrem performant ausgeführt werden können. Die Laufzeiten der Operationen Einfügen, Löschen und Zugriff sind von der Anzahl gespeicherter Elemente in der Regel (weitgehend) unabhängig. Allerdings erfordern hashbasierte Container einen zusätzlichen Aufwand, weil spezielle Hashwerte berechnet werden müssen, um diese Effizi-

enz zu erreichen. Darum sind die hashbasierten Container etwas schwieriger zu verstehen als Arrays und Listen. Die im Folgenden beschriebenen Grundlagen helfen dabei, die hashbasierten Container gewinnbringend einzusetzen. Zum leichteren Einstieg beginne ich mit einer Analogie aus dem realen Leben und einer vereinfachten Darstellung der Arbeitsweise, die im Verlauf der Beschreibung immer weiter präzisiert wird.

Analogie aus dem realen Leben

Hashbasierte Container kann man sich wie riesige Schrankwände mit nummerierten Schubladen vorstellen. In diesen Schubladen ist wiederum Platz für beliebig viele Sachen. Diese speziellen Schubladen werden in der Informatik auch als ***Bucket*** (zu deutsch: Eimer) bezeichnet. Soll ein Objekt in der Schrankwand abgelegt werden, so wird diesem eine Schubladennummer zugeteilt – wobei diese Nummer von den Eigenschaften (Attributen) des Objekts abhängt, das abgelegt werden soll. Wenn man später wieder auf Objekte zugreifen möchte, kann man dies mit der zuvor zugewiesenen Nummer tun. Zum leichteren Verwalten von Dingen in einer Schrankwand können wir uns intuitiv folgende Auswirkungen klarmachen:

1. Benutzt man immer nur ein und dieselbe Schublade, so quillt diese bald über und man findet seine Sachen nur mühselig wieder: Erschwerend kommt hinzu, dass der Inhalt einer Schublade des Öfteren komplett zu durchsuchen ist.
2. Verteilt man die Sachen relativ gleichmäßig über möglichst viele Schubladen, so kann man Sachen (nahezu) ohne Suchaufwand finden – die Kenntnis der richtigen Schublade vorausgesetzt.
3. Wenn kein gezielter Zugriff auf die korrekte Schublade erfolgt, etwa weil man sich in der Schublade irrt, so muss man im Extremfall alle Schubladen durchsuchen, um die gewünschten Sachen zu finden.

Die Analogie erleichtert das Verständnis der Anforderungen an hashbasierte Container und vor allem an die Methode `hashCode()`, die einen `int` zurückliefert:

1. Mithilfe der Methode `hashCode()` eines Objekts wird, vereinfacht gesagt, die Nummer für die Schublade berechnet, in der sich das Objekt befinden soll. Auch wenn es möglich und zulässig ist, dass `hashCode()` für unterschiedliche Objekte den gleichen `int`-Wert berechnet, sollte man das möglichst vermeiden. Wenn nämlich für zwei unterschiedliche Objekte derselbe Hashwert berechnet wird, so kommt es zu einer sogenannten ***Kollision***. Verschiedene Objekte werden dann im gleichen Bucket gespeichert und erfordern eine möglicherweise aufwendigere Suche innerhalb des Buckets.
2. Eine gleichmäßige Verteilung von Objekten auf Buckets lässt sich erreichen, indem die `hashCode()`-Methode für verschiedene Objekte möglichst verschiedene Werte zurückgibt. Idealerweise bildet man die Attribute selbst wieder auf Zahlen ab und multipliziert diese mit Primzahlen, wie wir es später noch sehen werden.

3. Aus dem letzten Punkt der Analogie kann man schließen, dass man die Nummern der Schubladen nicht verlieren oder verwechseln sollte. ***Um Schwierigkeiten zu vermeiden, empfiehlt es sich, dass sich der über die Methode `hashCode()` für ein Objekt berechnete Hashwert zur Laufzeit möglichst nicht ändert.*** Wenn sich allerdings die Grundlagen zur Berechnung ändern, kann man natürlich Änderungen am Hashwert nicht vermeiden. Man sollte sich jedoch der möglicherweise entstehenden Probleme bewusst sein, wie es im folgenden Praxishinweis beschrieben ist.

Hinweis: Auswirkungen bei Änderungen im berechneten Hashwert

Wie gerade angedeutet, wird für ein Objekt ein anderer Hashwert berechnet, weil sich die Werte dabei verwendeter Attribute ändern. Das hat Konsequenzen, die man kennen sollte: Liefern zu unterschiedlichen Zeiten die Berechnungen des Hashwerts für ein Objekt unterschiedliche Ergebnisse, so kann das Element nicht mehr über seinen zuvor berechneten Wert im Hashcontainer gefunden werden: Durch die Wertänderung wird an der falschen Stelle gesucht. Darüber hinaus kann eine Änderung im berechneten Hashwert zu der Inkonsistenz führen, dass mehrere gleiche Elemente in unterschiedlichen Buckets eingetragen werden, was ebenfalls verschiedenste andere Probleme mit sich bringt. ***Demnach ist es – wenn möglich – zu vermeiden, dass sich der berechnete Hashwert ändert***.

Realisierung in Java

Weil die Anzahl von Buckets beschränkt ist, muss der durch `hashCode()` berechnete `int`-Wert auf die Anzahl der tatsächlich verfügbaren Buckets – ***Kapazität*** genannt – abgebildet werden. Die Speicherung der Buckets erfolgt als eindimensionales Array in einer sogenannten ***Hashtabelle***. Jedes Bucket kann wiederum mehrere Elemente speichern. Um für ein zu speicherndes Objekt die Bucket-Nummer, also den Index innerhalb der Hashtabelle, zu bestimmen, wird das in Abbildung 6-6 angedeutete Verfahren genutzt, das folgender Berechnungsabfolge entspricht:

$$Object \xrightarrow{\texttt{hashCode}()} Hashwert \xrightarrow{f(Hashwert)} Bucket\text{-}Nummer$$

Als Abbildungsfunktion $f(Hashwert)$ zur Bestimmung der Bucket-Nummer wird von den Hashcontainern des JDKs eine Modulo-Operation angewendet: $f(Hashwert) = Hashwert \ \% \ Kapazität$. Die vorgestellte Arbeitsweise hat gewisse Konsequenzen:

- Selbst wenn sich die für Objekte berechneten Hashwerte unterscheiden, ist es aufgrund der Abbildungsfunktion f möglich, dass dieselbe Bucket-Nummer ermittelt wird und es zu einer ***Kollision*** kommt: Verschiedene Objekte werden in dasselbe Bucket eingeordnet.

Abbildung 6-6 *Aufbau von hashbasierten Containern*

- Würden alle Objekte lediglich wenige unterschiedliche Bucket-Nummern (oder gar dieselbe) zurückliefern, so würde keine einigermaßen gleichmäßige Verteilung mehr erfolgen, sondern es käme zu einem Effekt, den man ***Clustering*** nennt. Das meint, dass in einigen Buckets sehr viele Elemente gespeichert werden und in anderen nahezu keine. Im Extremfall wird für alle Elemente die gleiche Bucket-Nummer berechnet. Der Hashcontainer würde dadurch auf eine Liste bzw. auf einen Baum[5] reduziert werden – zusätzlich aber mit Verwaltungsoverhead. Trotzdem würde das Programm noch funktionieren, nur recht inperformant.

Ein Zugriff auf ein Element in einem Hashcontainer oder eine Suche danach erfordert durch den Hashcontainer ein zweistufiges Vorgehen:

1. Zunächst wird das Bucket bestimmt. Dazu wird die Methode `hashCode()` und die interne Abbildungsfunktion f des Hashcontainers benutzt.
2. Anschließend wird mithilfe von `equals(Object)` in der Collection des Buckets nach dem gewünschten Element gesucht.

Nach diesen grundsätzlichen Betrachtungen zur Arbeitsweise wollen wir uns konkret die Implikationen für Java-Klassen ansehen. Die Klasse `Object` stellt bekanntlich Defaultimplementierungen der Methoden `hashCode()` und `equals(Object)` bereit. Diese sind aber lediglich für sehr wenige Anwendungsfälle ausreichend. ***Darum sollten bei der Speicherung von Objekten eigener Klassen in hashbasierten Containern unbedingt immer deren Methoden `hashCode()` und `equals(Object)` konsistent zueinander überschrieben werden***.

[5]Die Verwaltung der Elemente erfolgt gewöhnlich in einer Liste. Seit JDK 8 wird als Optimierung ein Baum genutzt, sofern die zu speichernden Typen das Interface `Comparable<T>` erfüllen.

Hinweis: Hashwerte in Mengen bzw. Schlüssel-Wert-Abbildungen

Der Hashwert wird mit der `hashCode()`-Methode entweder des Objekts selbst (bei Mengen) oder für Schlüssel-Wert-Abbildungen desjenigen Objekts, das als Schlüssel dient, berechnet.[a] Damit ich beides im Anschluss nicht immer auseinanderhalten muss, beschreiben die folgenden Ausführungen zur einfacheren Darstellung den Ablauf für Mengen. Für Schlüssel-Wert-Abbildungen muss man sich abweichend davon nur gewahr sein, dass die `hashCode()`-Methode für Objekte vom Typ des Schlüssels und zur späteren Suche im Bucket die `equals(Object)`-Methode derjenigen Klasse aufgerufen wird, die den Typ des Werts beschreibt.

[a] Für die Elemente von Listen kann man zwar auch `hashCode()` implementieren, hier hat es jedoch keinen Einfluss auf die Speicherung innerhalb der Liste.

Die Rolle von `hashCode()` beim Suchen

Konkretisieren wir die gerade gemachten Aussagen. Dazu wird das in Abschnitt 5.1.2 zur Demonstration der Methode `equals(Object)` verwendete Beispiel mit Objekten des Typs `Spielkarte` etwas abgewandelt: Statt einer Speicherung in einer `ArrayList<Spielkarte>` erfolgt diese nun in einem bereits in Abschnitt 6.1.6 vorgestellten `HashSet<E>`, nachfolgend also `HashSet<Spielkarte>`:

```
public static void main(final String[] args)
{
    final Collection<Spielkarte> spielkarten = new HashSet<>();
    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));
    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK, 8));
    System.out.println("gefunden = " + gefunden);
}
```

Listing 6.12 *Ausführbar als* **'SPIELKARTEINHASHSET'**

Wir erwarten, dass das Ergebnis einer Suche unabhängig davon ist, ob man Objekte in einem `HashSet<Spielkarte>` oder einer `ArrayList<Spielkarte>` speichert. Es stellt sich die Frage: Gefunden oder nicht gefunden? Prüfen wir den Wert der Variablen `gefunden`. Möglicherweise erleben wir dabei eine Überraschung. Die gesuchte »Pik 8« wird im Container nicht gefunden. Das ist merkwürdig, da die Methode `equals(Object)` der Klasse `Spielkarte` in Abschnitt 5.1.2 korrekt implementiert wurde.

Ein kurzes Nachdenken bringt die Lösung: Beim Zugriff auf Hashcontainer wird zunächst durch Aufruf von `hashCode()` die Schublade berechnet, in der anschließend mit `equals(Object)` nach Objekten gesucht wird. Für die Klasse `Spielkarte` wurde die Methode `hashCode()` jedoch nicht überschrieben. Dadurch wird die Defaultimple-

mentierung aus der Klasse `Object` ausgeführt, die typischerweise als `hashCode()` die Speicheradresse der Objektreferenz zurückliefert. Zur Suche wird aber ein neu erzeugtes Objekt verwendet, das zwar die gleiche Spielkarte darstellt, aber eine unterschiedliche Referenz besitzt. Dadurch sind die für die beiden Spielkartenobjekte berechneten Hashwerte unterschiedlich und es wird in zwei unterschiedlichen Buckets gesucht.

Wir müssen also die `hashCode()`-Methode der Klasse `Spielkarte` korrigieren. Betrachten wir dazu zunächst den `hashCode()`-Kontrakt.

Der `hashCode()`-Kontrakt

Die Methode `hashCode()` bildet den Objektzustand (besser: den möglichst unveränderlichen Teil davon) auf eine Zahl ab und wird in der Regel dazu benötigt, Objekte in hashbasierten Containern verarbeiten zu können. Die Methode `hashCode()` ist durch die JLS (Java Language Specification) mit folgender Signatur definiert:

```
public int hashCode()
```

Eine Implementierung von `hashCode()` sollte folgende Eigenschaften erfüllen:[6]

- **Eindeutigkeit** – Während der Ausführung eines Programms sollte der Aufruf der Methode `hashCode()` für ein Objekt, sofern möglich (d. h. falls sich keine relevanten Attribute ändern), denselben Wert zurückliefern.
- **Verträglichkeit mit `equals()`** – Wenn die Methode `equals(Object)` für zwei Objekte `true` zurückgibt, dann muss die Methode `hashCode()` für beide Objekte denselben Wert liefern. Umgekehrt gilt dies nicht: Bei gleichem Hashwert können zwei Objekte per `equals(Object)` verschieden sein.

Daraus können wir folgende Hinweise zur Realisierung der Methode `hashCode()` herleiten: Zur Berechnung sollten diejenigen (möglichst ***unveränderlichen***) Attribute verwendet werden, die auch in `equals(Object)` zur Bestimmung der Gleichheit genutzt werden. Dadurch werden Änderungen des Hashwerts vermieden bzw. auf nur tatsächlich benötigte Fälle eingeschränkt. Die Verträglichkeit mit `equals(Object)` ist automatisch dadurch gegeben, dass nur diejenigen Attribute zur Berechnung genutzt werden (oder auch nur ein Teil davon), die in `equals(Object)` verglichen werden.

Fallstricke bei der Implementierung von `hashCode()` Ein typischer Fehler ist, dass die Methode `equals(Object)` überschrieben wird, die Methode `hashCode()` jedoch nicht. Dadurch wird in der Regel die Zusicherung verletzt, die besagt, dass für zwei laut `equals(Object)` gleiche Objekte auch der gleiche Wert durch `hashCode()` berechnet wird.

Leider sieht man immer mal wieder Realisierungen von `hashCode()`, die veränderliche Attribute zur Berechnung verwenden. Das kann in einigen Fällen korrekt sein,

[6] Werden diese nicht eingehalten, sollte dies unbedingt in der Javadoc vermerkt werden.

wird aber vermutlich häufiger Probleme bereiten.[7] In dem Zusammenhang hatte ich bereits angesprochen, dass man Objekte in Hashcontainern nicht mehr wiederfindet, wenn durch eine Änderung des Hashwerts im falschen Bucket gesucht wird. Aufgrund dessen sollte man einen kritischen Blick auf die Zusammensetzung der zur `hashCode()`-Berechnung verwendeten Attribute werfen und versuchen, bevorzugt unveränderliche Attribute zu nutzen. So kann man vermeiden, dass sich der berechnete Hashwert bei jeder Modifikation von Attributen des Objekts ändert.

Realisierung von `hashCode()` für die Klasse `Spielkarte` Zur Korrektur der Klasse `Spielkarte` implementieren wir dort die Methode `hashCode()`. Dazu werfen wir einen Blick auf die Methode `equals(Object)`:

```
@Override
public boolean equals(Object other)
{
    if (other == null) // Null-Akzeptanz
        return false;
    if (this == other) // Reflexivität
        return true;
    if (this.getClass() != other.getClass()) // Typgleichheit
        return false;

    // int mit Wertevergleich, Enum mit equals()
    final Spielkarte karte = (Spielkarte) other;
    return this.wert == karte.wert && this.farbe.equals(karte.farbe);
}
```

In der Realisierung der Methode `hashCode()` können wir in diesem Fall die Attribute `wert` und `farbe` zur Berechnung verwenden. Um die Verteilung auf die Buckets möglichst gut zu streuen, kann man mit Primzahlen als Multiplikatoren arbeiten: Jeder Spielkartenwert wird mit einem Primzahlfaktor multipliziert und dann der Hashwert der Farbe hinzu addiert, etwa wie folgt:

```
@Override
public int hashCode()
{
    final int PRIME = 37;
    return this.wert * PRIME + this.farbe.hashCode();
}
```

Man erreicht zwar so eine gleichmäßige Verteilung – für komplexere Klassen wird die Implementierung der Methode `hashCode()` aber doch schnell unübersichtlich und kompliziert. Als weitere Anforderung neben der gleichmäßigen Verteilung sollten die Hashfunktionen recht einfach und effizient zu berechnen sein, da sie unter Umständen sehr oft aufgerufen werden. Um sich darüber nicht allzu viele Gedanken machen zu müssen, ist der Einsatz einer passenden Utility-Klasse wünschenswert.

[7] Die in Eclipse eingebaute Automatik aus dem Menü SOURCE –> GENERATE HASHCODE() AND EQUALS()… erzeugt eine `hashCode()`-Methode, die potenziell zu viele Attribute nutzt.

Einsatz einer Utility-Klasse zur Berechnung von `hashCode()`

In den vorherigen Auflagen dieses Buchs habe ich noch eine Utility-Klasse `HashUtils` selbst entwickelt. Weil aber mittlerweile im JDK eine adäquate und einfach zu nutzende Realisierung existiert und man zudem Standards eigenen Entwicklungen vorziehen sollte, zeige ich hier den Einsatz der Methode `hash()` aus der Utility-Klasse `java.util.Objects`.

Einsatz der Utility-Klasse Für die Klasse `Spielkarte` nutzen wir die Klasse `Objects`, um die Berechnung in `hashCode()` einfach und übersichtlich zu schreiben:

```
@Override
public int hashCode()
{
   return Objects.hash(this.wert, this.farbe);
}
```

Realisierung von `hashCode()` für die Klasse `Person` Mit dem bis hierher erlangten Wissen können wir nun auch die Klasse `Person` um eine passende, gut verständliche Realisierung von `hashCode()` erweitern:

```
@Override
public int hashCode()
{
    return Objects.hash(this.name, this.birthday, this.city);
}
```

Füllgrad (Load Factor)

Wir besitzen nun das Wissen, um `hashCode()` so zu implementieren, dass Kollisionen weitestgehend vermieden werden, indem eine möglichst gleichmäßige Verteilung der zu speichernden Elemente erfolgt. Das hängt einerseits von der gewählten Hashfunktion sowie andererseits von der Anzahl verfügbarer Buckets und gespeicherter Elemente ab: Werden fortlaufend immer mehr Elemente in einem hashbasierten Container gespeichert, so wächst die Wahrscheinlichkeit für und die Anzahl von Kollisionen. Ähnlich wie die Klasse `ArrayList<E>` führen auch Hashcontainer eine automatische Größenanpassung der Hashtabelle, d. h. eine Erweiterung um Buckets, durch, wenn die Anzahl gespeicherter Elemente einen Grenzwert übersteigt. Um zu bestimmen, ob eine Größenanpassung nötig ist, betrachtet man nicht den Inhalt aller Buckets im Einzelnen, sondern nutzt eine einfachere, aber effektive Variante: Dabei hilft als Kenngröße der sogenannte ***Füllgrad***, auch ***Load Factor*** genannt, der sich aus dem Quotienten der Anzahl gespeicherter Elemente und der Anzahl der Buckets ergibt. Dieser Wert beschreibt, wie voll der Hashcontainer werden darf, bis es zu einer Größenanpassung kommt. Bis zu einem Füllgrad von etwa 75 % sind Kollisionen erfahrungsgemäß eher unwahrscheinlich (vgl. Javadoc-Dokumentation der Klassen `Hashtable<K,V>` und `HashMap<K,V>`).

Die Hashtabelle wird erweitert, wenn folgende Bedingung zwischen Füllgrad, Anzahl gespeicherter Elemente und der Kapazität der Hashtabelle erfüllt ist:

$$\textit{maximaler Füllgrad} * \textit{Kapazität} \geq \textit{Anzahl Elemente}$$

Wenn man zu dem Zeitpunkt, an dem die Hashtabelle angelegt wird, die ungefähre Anzahl der später zu speichernden Elemente kennt, kann man nachträgliche Größenanpassungen oftmals vermeiden, indem man die initiale Kapazität als Quotient aus der Anzahl der Elemente und dem maximal akzeptierten Füllgrad passend wählt:

$$\textit{initiale Kapazität} = \textit{Anzahl Elemente}/\textit{maximaler Füllgrad}$$

Für 1.000 Elemente ergibt sich bei einem maximalen Füllgrad von 75 % die initiale Kapazität wie folgt: *initiale Kapazität* $= 1.000/0,75 \approx 1.333$. Bei der Konstruktion eines Hashcontainers kann man den berechneten Wert der initialen Kapazität angeben, wobei dieser rund 1,3-mal so groß wie die geplante Anzahl der zu verwaltenden Elemente sein sollte. Bei einem angenommenen idealen Füllfaktor von 75 % bedeutet dies, dass immer etwa 30 % Kapazität unbelegt bleiben.

Der maximal erlaubte Füllgrad stellt damit eine Stellschraube von Hashcontainern dar, die sich auf Speicherplatz und Zugriffszeit auswirkt. Je kleiner der Wert des maximal erlaubten Füllgrads, desto geringer ist die Wahrscheinlichkeit für Kollisionen. Damit ist der Zugriff schneller, als wenn es Kollisionen gibt. Allerdings geht dies zulasten des benötigten Speichers und erhöht den Anteil unbenutzter Buckets. Umgekehrt »verschwendet« ein maximal erlaubter Füllgrad größer als 75 % zwar weniger Speicher, jedoch steigt die Wahrscheinlichkeit für Kollisionen. Dadurch verschlechtern sich die Zugriffszeiten auf die Elemente.

Auswirkungen von Größenanpassungen

Werden mehr Elemente in einem Hashcontainer gespeichert als zunächst erwartet, und übersteigt der momentane Füllgrad die durch den maximal erlaubten Füllgrad angegebene Schwelle, so wird die Hashtabelle automatisch vergrößert. Es stehen daraufhin mehr Buckets zur Verfügung. Im Gegensatz zu Array-basierten Listen, die neue Daten nach einem solchen Vergrößerungsschritt einfach am Ende anfügen können, ist dies für hashbasierte Container komplizierter. ***Nach einer Größenanpassung muss die Hashtabelle vollständig neu organisiert werden***, da die Abbildungsfunktion für zuvor gespeicherte Werte nicht mehr korrekt arbeitet: *Die Modulo-Operation liefert nun in der Regel andere Werte als zuvor*. Für jedes Element in der Hashtabelle muss das entsprechende aufnehmende Bucket neu ermittelt werden. Diesen Umsortierungsvorgang nennt man ***Rehashing***. Da jedes gespeicherte Element betrachtet werden muss, ist dieser Vorgang relativ aufwendig. Als Optimierung wird vom Hashcontainer zu jedem Element dessen zuvor über die Methode `hashCode()` berechneter Wert zwischengespeichert. Dieser ändert sich bei einem Rehashing nicht. Dadurch werden zusätzliche Performance-Einbußen durch die Neuberechnung der Hashwerte durch Aufrufe von `hashCode()`

vermieden. Das neue, aufnehmende Bucket kann auf Basis des zwischengespeicherten Hashwerts bestimmt werden. Das Rehashing kostet Rechenzeit, macht spätere Zugriffe aber wieder performanter, weil dadurch weniger Kollisionen auftreten.

Neben dem Rehashing gibt es folgendes Detail zu bedenken: Falls in einem Hashcontainer irgendwann einmal sehr viele Elemente gespeichert wurden, kam es als Folge höchstwahrscheinlich zu einigen Vergrößerungsschritten und damit auch Rehashing-Vorgängen. Werden später Elemente gelöscht, so wird die Größe der Hashtabelle nicht automatisch verkleinert und der Speicher bleibt (unnütz) belegt. Schlimmer noch: ***Im Gegensatz zu Listen gibt es für die Hashcontainer kein Pendant zu `trimToSize()`, das es nach einer mittlerweile hinfälligen Expansion erlaubt, den Speicherverbrauch zu beschneiden***. Als Abhilfe kann man einen neuen Hashcontainer mit passend gewählter Größe anlegen, der mit dem Inhalt des bisherigen gefüllt wird.

Um wieder das Beispiel einer Schrankwand zu bemühen: Analog zu den Vergrößerungen wird diese um einen Anbausatz und damit weiteren Stauraum ergänzt, wenn der Platz eng wird. Ein Umräumvorgang sorgt für eine bessere Verteilung der Sachen auch auf die neuen Schubladen und erleichtert eine spätere Suche, da wieder mehr Ordnung herrscht und in jeder Schublade weniger Dinge gelagert sind. Bezogen auf die Speicherverschwendung gilt in etwa folgende Analogie: Nach einem Frühjahrsputz sind beispielsweise mehr als die Hälfte aller Schubladen des Schranks leer. Der Anbausatz wird aber nicht abmontiert, sondern nimmt dann einfach nur noch Platz weg.

6.1.8 Grundlagen automatisch sortierender Container

Mitunter ist es praktisch, wenn die in einer Containerklasse verwalteten Daten sortiert vorliegen. Bekanntermaßen stellen die Containerklassen `TreeSet<E>` bzw. `TreeMap<K,V>` automatisch eine Sortierung von Elementen ohne weiteren Implementierungsaufwand im Applikationscode her. Für Arrays und Listen gibt es keine solche Automatik im JDK. Um diese sortiert zu halten, wird ein manueller Schritt notwendig. Hierbei unterstützen die Methoden `sort()` aus den Utility-Klassen `Arrays` und `Collections` aus dem Package `java.util` (vgl. Abschnitt 6.2.2). Seit Java 8 bietet das Interface `List<E>` praktischerweise eine `sort()`-Methode (vgl. Abschnitt 6.1.5).

Aber unabhängig von automatischer oder manueller Sortierung muss immer eine Ordnung festgelegt werden, um bei Vergleichen von Objekten »kleiner« bzw. »größer« oder »gleich« ausdrücken zu können. Das kann man mithilfe von Implementierungen der Interfaces `Comparable<T>` und `Comparator<T>` beschreiben:

- **Natürliche Ordnung und `Comparable<T>`** – Sofern zu speichernde Objekte das Interface `Comparable<T>` erfüllen, können sie darüber ihre Ordnung, d. h. ihre Reihenfolge untereinander, beschreiben. Diese Reihenfolge wird auch als ***natürliche Ordnung*** bezeichnet, da sie durch die Objekte selbst bestimmt wird.[8]

[8] Über das »natürlich« kann man sich trefflich streiten, weil es nur um die Vorgabe einer Reihenfolge durch die Implementierung geht und die Ordnung auch nicht intuitiv oder unerwartet sein kann und sich somit möglicherweise sogar eher unnatürlich anfühlt.

- **Weitere Ordnungen und `Comparator<T>`** – Teilweise benötigt man zusätzlich zur natürlichen Ordnung weitere oder alternative Sortierungen, etwa wenn man Personen nicht nach Nachname, sondern alternativ nach Vorname und Geburtsdatum ordnen möchte. Diese ergänzenden Sortierungen können mithilfe von Implementierungen des Interface `Comparator<T>` festgelegt werden. Dadurch lassen sich von der natürlichen Ordnung abweichende Sortierungen für Objekte einer Klasse realisieren und auch Objekte von Klassen sortieren, für die keine natürliche Ordnung definiert ist, weil das Interface `Comparable<T>` nicht implementiert wird.

Sortierungen und das Interface `Comparable<T>`

Oftmals besitzen Werte oder Objekte eine natürliche Ordnung: Das gilt etwa für Zahlen und Strings. Für komplexe Typen ist die Aussage »kleiner« bzw. »größer« nicht immer sofort ersichtlich, lässt sich aber bei Bedarf selbst definieren.

Dazu erlaubt das Interface `Comparable<T>` typsichere Vergleiche und deklariert die Methode `compareTo(T)` folgendermaßen:

```
public interface Comparable<T>
{
    public int compareTo(T o);
}
```

Das Vorzeichen des Rückgabewerts bestimmt die Reihenfolge der Elemente:

- $= 0$: Der Wert 0 bedeutet Gleichheit des aktuellen und des übergebenen Objekts.
- < 0: Das aktuelle Objekt ist kleiner als das übergebene Objekt.
- > 0: Das aktuelle Objekt ist größer als das übergebene Objekt.

Diverse Klassen im JDK (alle Wrapper-Klassen, `String`, `Date` usw.) implementieren das Interface `Comparable<T>` und sind damit automatisch sortierbar.

Implementieren von `compareTo()` in eigenen Klassen Wie man das Interface `Comparable<T>` für eigene Klassen implementiert, zeige ich für die folgende Klasse `Person`. Dort wird anstatt des Geburtsdatums als Objekt das Alter bewusst als primitiver Typ gespeichert, um einige Varianten bei der Realisierung des Interface `Comparable<Person>` zu verdeutlichen:

```
public final class Person implements Comparable<Person>
{
    private final String name;
    private final String city;
    private final int    age;

    public Person(final String name, final String city, final int age)
    {
        this.name = Objects.requireNonNull(name, "name must not be null");
        this.city = Objects.requireNonNull(city, "city must not be null");
        this.age = age;
    }
```

Eine erste Implementierung von `compareTo(Person)` könnte die natürliche Ordnung für `Person`-Objekten (leicht unintuitiv) ausschließlich über deren Namen realisieren:

```
@Override
public int compareTo(final Person otherPerson)
{
    return getName().compareTo(otherPerson.getName());
}
```

Hier ist hilfreich, dass die für den Namen genutzte Klasse `String` das Interface `Comparable<String>` erfüllt. Zudem benötigen wir keine `null`-Prüfung, weil laut Kontrakt eine `NullPointerException` ausgelöst werden soll, sofern ein Aufruf von `compareTo(null)` erfolgt.

Betrachten wir den Einsatz unserer Methode `compareTo(Person)` und nehmen dazu an, eine Kundenliste `customers` enthielte etwa folgende Einträge:

```
customers.add(new Person("Müller", "Bremen", 27));
customers.add(new Person("Müller", "Kiel", 37));
```

Nutzen wir die obige Umsetzung, so werden laut `compareTo(Person)` alle Objekte vom Typ `Person` mit gleichem Namen als gleich angesehen. Dass dies keine wirklich gelungene Realisierung einer natürlichen Ordnung für Personen darstellt, wird nach ein wenig Überlegen klar: Herr Müller aus Kiel ist nicht Herr Müller aus Bremen. Wie geht es also besser? Einen guten Anhaltspunkt stellt oft die Methode `equals(Object)` und die dort zum Vergleich verwendeten Attribute dar. Nachfolgend werden hier neben dem Namen zusätzlich die Attribute `city` und `age` zur Gleichheitsprüfung herangezogen:

```
public boolean equals(final Object obj)
{
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;

  final Person other = (Person) obj;
  return name.equals(other.name) && city.equals(other.city) && age == other.age;
}
```

Vorgehen zur Implementierung von `compareTo()` Im Allgemeinen kann man sich beim Implementieren von `compareTo(T)` an einer bestehenden Realisierung der Methode `equals(Object)` der jeweiligen Klasse orientieren. Alle dort verglichenen Attribute sind in der Regel auch für die Sortierung gemäß der natürlichen Ordnung relevant.[9] Für die Attribute kann man wie folgt vorgehen:

[9] Allerdings ist die Reihenfolge für den Vergleich unbedingt zu beachten. Während diese für `equals(Object)` keinen Einfluss auf das Ergebnis besitzt, macht es für `compareTo(T)` möglicherweise einen großen Unterschied: Es ist entscheidend, ob erst die Namen und dann das Alter oder erst das Alter und dann die Namen verglichen werden.

1. Referenztypen, die das Interface `Comparable<T>` implementieren, verwenden deren `compareTo(T)`-Methoden.
2. Für primitive Datentypen lassen sich die Vergleichsoperatoren '<', '=' und '>' einsetzen, etwa für einen Vergleich des Attributs `age`.[10] Statt jedoch die drei Fälle größer, kleiner und gleich selbst abzufragen und den passenden Rückgabewert bereitzustellen, ist es sinnvoller, die Methoden `compare()` der jeweiligen Wrapper-Klasse zu nutzen, weil diese einem Arbeit abnehmen und der Vergleich wie folgt kürzer und klarer notiert werden kann:

```
int result = Integer.compare(this.getAge(), otherPerson.getAge());
```

3. Für alle Attribute anderen Typs muss der Vergleich selbst implementiert werden. Wenn man den Sourcecode der Klasse des Attributs im Zugriff hat, kann man diese derart erweitern, dass sie das Interface `Comparable<T>` erfüllt und dort die gewünschten Attribute vergleicht. Hat man eine Klasse jedoch nicht im Zugriff oder soll/darf diese nicht verändert werden, so muss der Vergleich der relevanten Attribute dieser Klasse gemäß den Schritten 1 und 2 selbst programmiert werden.

Konsistenz von `compareTo()` und `equals()` Die Methoden `compareTo(T)` und `equals(Object)` sollten so implementiert werden, dass `x.compareTo(y)` für beliebige `x` und `y` gleichen Typs genau dann den Wert 0 zurückgibt, wenn der Vergleich `x.equals(y)` den Wert `true` liefert. Wird gegen diese Regel verstoßen, so empfiehlt es sich, dies im Javadoc zu vermerken.

Für unser Beispiel ist die Forderung nicht eingehalten, weil `compareTo(Person)` schwächer prüft als `equals(Object)`. Um nicht für Verwirrung beim Einsatz zu sorgen, korrigieren wir die Implementierung dahingehend, dass `compareTo(Person)` auch die Attribute `city` und `age` beim Vergleich heranzieht:

```
@Override
public int compareTo(final Person otherPerson)
{
    Objects.requireNonNull(otherPerson, "otherPerson must not be null");

    int ret = getName().compareTo(otherPerson.getName());
    if (ret == 0)
    {
        ret = getCity().compareTo(otherPerson.getCity());
    }
    if (ret == 0)
    {
        ret = Integer.compare(getAge(), otherPerson.getAge()));
    }
    return ret;
}
```

[10] Für die Typen `float` und `double` sind Rundungsfehler zu bedenken (vgl. Abschnitt 5.1.2).

Falls man in den Methoden `compareTo(T)` und `equals(Object)` dieselben Attribute nutzt, lässt sich Sourcecode-Duplikation vermeiden, indem man in `equals(Object)` die Methode `compareTo(T)` aufruft:

```
@Override
public boolean equals(final Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    final Person other = (Person) obj;
    return compareTo(other) == 0;        // Vergleich mittels compareTo(Person)
}
```

Diese Art der Realisierung vermeidet zum einen Konsistenzprobleme zwischen beiden Methoden und führt zum anderen dazu, dass die Vergleichslogik nur einmal in der Methode `compareTo(T)` realisiert wird. Dies ist wiederum hilfreich, wenn Änderungen an der Klasse erfolgen, etwa Attribute hinzugefügt werden. Schnell wird dabei übersehen, beide Implementierungen anzupassen. Auch hier erkennt man den Vorteil, eine Funktionalität möglichst nur einmal zu realisieren. Andrew Hunt und David Thomas beschreiben dies in ihrem Buch »Der Pragmatische Programmierer« [34] als das sogenannte DRY-Prinzip (**D**on't **R**epeat **Y**ourself).

Entwicklung: `compareTo(T)` basierend auf `equals(Object)`

Häufig entwickelt man bei einer Neuimplementierung einer Klasse zunächst eine `equals(Object)`-Methode. Wird im Verlauf der Entwicklung eine natürliche Ordnung durch Erfüllen des Interface `Comparable<T>` erforderlich, so bietet es sich oftmals an, `equals(Object)` durch Aufruf von `compareTo(T)` zu realisieren und die Vergleichslogik in `compareTo(T)` zu verlagern.

Sortierungen und das Interface `Comparator<T>`

Wir haben zur Beschreibung der natürlichen Ordnung das Interface `Comparable<T>` kennengelernt. Darüber lässt sich lediglich *eine* spezielle Sortierung beschreiben. In vielen Anwendungsfällen sind weitere Sortierungen wünschenswert, z. B. möchte man in Tabellen häufig nach jeder beliebigen Spalte sortieren können. Dies wird durch den Einsatz der im Folgenden beschriebenen ***Komparatoren*** möglich. Der Vorteil dieses Vorgehens ist, dass man Anwendungsklassen nicht mit Sortierfunktionalität überfrachtet, sondern diese in ***eigenständigen Vergleichsklassen*** definiert wird. Dazu müssen diese das Interface `Comparator<T>` erfüllen und die gewünschte Sortierung realisieren. Als Hinweis sei angemerkt, dass die dafür benötigten Attribute bzw. deren Zugriffsmethoden in ihrer Sichtbarkeit möglicherweise eingeschränkt und im `Comparator<T>` nicht zugreifbar sind. Mit `Comparable<T>` hat man immer Zugriff auf alle Attribute.

Das Interface `Comparator<T>` Das Interface `Comparator<T>` beschreibt einen Baustein zum Vergleich von Objekten des Typs `T`. Hierfür wird die Methode `int compare(T, T)` angeboten, die dazu dient, zwei beliebige Objekte des Typs `T` miteinander zu vergleichen:

```
public interface Comparator<T>
{
    int compare(T o1, T o2);
}
```

Über den Rückgabewert wird die Reihenfolge der Sortierung bestimmt. Es gilt:

- $= 0$: Der Wert 0 bedeutet Gleichheit der beiden Objekte `o1` und `o2`.
- < 0: Das erste Objekt `o1` ist als kleiner als das zweite Objekt `o2`.
- > 0: Bei positiven Rückgabewerten ist das erste Objekt `o1` größer als das zweite Objekt `o2`.

Grundgerüst eines einfachen Komparators Stellen wir uns vor, unsere Aufgabe bestünde darin, eine Liste mit `Person`-Objekten nach verschiedenen Kriterien zu sortieren, etwa nach Name, Wohnort oder Alter. Der grundsätzliche Aufbau einer Realisierung für Komparatoren für einen Typ `T` folgt oftmals einem sehr ähnlichen Schema: In der `compare(T,T)`-Methode werden die benötigten Vergleiche der gewünschten Attribute durchgeführt. Einen Vergleich basierend auf Namen realisiert man beispielsweise wie folgt:

```
public final class PersonNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        Objects.requireNonNull(person1, "person1 must not be null");
        Objects.requireNonNull(person2, "person2 must not be null");

        return person1.getName().compareTo(person2.getName());
    }
}
```

Vereinfachungen mit JDK 8 Zunächst einmal kann man mit JDK 8 einen Lambda wie folgt nutzen:

```
final Comparator<Person> nameComparator = (person1, person2) ->
{
    return person1.getName().compareTo(person2.getName());
};
```

Noch prägnanter geht es mit den Erweiterungen aus dem Interface `Comparator<T>` selbst (vgl. Abschnitt 6.2.4), etwa `comparing()`, wie folgt:

```
final Comparator<Person> nameComparator = Comparator.comparing(Person::getName);
```

Diskussion: Konsistenz von `compare()` und `equals()`

Obwohl im `Comparator<T>` im Javadoc von `compare(T,T)` empfohlen wird, dass `(compare(x,y) == 0)` == `(x.equals(y))` gelten sollte, ist dies in der Praxis häufig nicht der Fall. Eine entsprechende Forderung wurde bereits bezüglich des Interface `Comparable<T>` aufgestellt. Dabei gibt es zwischen beiden Forderungen die folgenden, entscheidenden Unterschiede.

Unterschiede der Forderungen von `compareTo()` und `compare()`

Realisierungen des Interface `Comparable<T>` sind meistens bijektiv, d. h., es existiert eine »Genau-dann-wenn«-Beziehung: Aus einer Gleichheit bezüglich `compareTo(T)` folgt eine Gleichheit bezüglich `equals(Object)` und auch umgekehrt: Ergibt `equals(Object)` Gleichheit, so gilt dies auch für `compareTo(T)`.

Realisierungen des Interface `Comparator<T>` sind dagegen oftmals injektiv, d. h., es wird eine »Daraus-folgt«-Beziehung beschrieben: Aus einer Gleichheit gemäß `equals(Object)` kann man (in der Regel) auf eine Gleichheit bezüglich `compare(T,T)` schließen. Aus einer Gleichheit gemäß `compare(T,T)` folgt jedoch meist *keine* Gleichheit bezüglich `equals(Object)`. Anhand von Komparatoren für `Person`-Objekte, die die Attribute `name` bzw. `city` vergleichen, kann man sich dies verdeutlichen: Zwei gleichnamige oder in der gleichen Stadt wohnende Personen werden über die jeweilige Realisierung von `compare(Person, Person)` als gleich angesehen, für `equals(Object)` gilt das logischerweise nicht, da hier noch weitere Attribute wie z. B. Geburtstag oder Größe verglichen werden.

Hintergrundwissen: Arbeitsweise sortierender Datenstrukturen

Zum besseren Verständnis der Arbeitsweise der Containerklassen `TreeSet<E>` bzw. `TreeMap<K,V>` betrachten wir ein `TreeSet<Long>`, das initial die Werte 1, 2 und 3 speichert und in das anschließend die Werte 4, 5 und 6 eingefügt werden. Bevor ich auf Details beim Einfügen eingehe, erläutere ich kurz die zugrunde liegende Datenstruktur.

Es wird ein sogenannter ***binärer Baum*** genutzt, der sich dadurch auszeichnet, dass es einen speziellen Startknoten (***Wurzel*** genannt) gibt, der maximal einen direkten linken und einen direkten rechten Kindknoten besitzt. Diese Kindknoten können wiederum jeweils maximal zwei direkte Kindknoten haben, dies aber beliebig fortgesetzt, sodass ein Knoten beliebig viele Nachfahren besitzen kann. Die ***Tiefe des Baums*** ist als die maximale Anzahl der Knoten auf dem Weg von der Wurzel bis zu einem Knoten ohne Nachfahren (auch ***Blatt*** genannt) definiert. Per Definition werden jeweils in den linken Kindknoten diejenigen Elemente eingefügt, die in der Wertebelegung ihrer Attribute als kleiner als der momentane Knoten anzusehen sind. Analog gilt dies für »größere« Elemente, die im rechten Teilbaum gespeichert werden.[11]

[11] Schnell stellt sich die Frage: Was ist mit gleichen Werten? In den baumbasierten Datenstrukturen `TreeSet<E>` und `TreeMap<K,V>` werden nicht mehrere gleiche Werte gespeichert, sondern es existiert jeweils nur ein derartiger Eintrag. Für Mengen ist dies per Definition so, für

Für unser Beispiel des `TreeSet<Long>` ergibt sich mit diesem Wissen und den initialen Werten ein Baum, dessen Wurzelknoten den Wert 2 hat und der einen linken sowie rechten Nachfolger mit den Werten 1 bzw. 3 besitzt. Um die Arbeitsweise beim Einfügen von Elementen zu verdeutlichen, werden dann sukzessive die Elemente 4, 5 und 6 eingefügt. Bei Einfügeoperationen (und selbstverständlich auch bei den hier nicht gezeigten Löschoperationen) wird einerseits immer die gewünschte Sortierung hergestellt und andererseits durch ***Balancierung*** (Höhenausgleich der Teilbäume) für eine ausgeglichene Verteilung der innerhalb der Datenstruktur gespeicherten Elemente gesorgt. Die Auswirkungen verschiedener Aktionen auf den Baum zeigt Abbildung 6-7.

Abbildung 6-7 *Arbeitsweise eines balancierten Baums*

Da die Elemente 4, 5 und 6 größer als die Wurzel sind, werden sie zunächst im rechten Teilbaum einsortiert. Durch Einfügen des Werts 4 entsteht lediglich eine Höhendifferenz von 1 zwischen dem linken und dem rechten Teilbaum. Eine solche Differenz führt nicht zu einem Ausgleichsvorgang, weil sich eine derartige Dysbalance nicht in jedem Fall vermeiden lässt – beispielsweise ist bei zwei gespeicherten Elementen immer ein Teilbaum leer. Durch das Einfügen der Werte 5 und 6 im obigen Beispiel ergibt sich allerdings eine Unausgewogenheit in der Tiefe der Teilbäume, deren Differenz größer als eins ist. Wie in der Abbildung ersichtlich, erhält man nach einem Einfügen möglicherweise fast so etwas wie eine lineare Liste. Um performante Suchen zu gewährleisten,

Maps gilt dies, da hier die Eindeutigkeit von Schlüsseln gefordert wird. Versucht man trotzdem einen gleichen Wert zu speichern, so wird der alte Eintrag ersetzt, also für `TreeMap<K,V>` ein neuer Wert für den Schlüssel eingetragen.

ist es das Ziel, die Tiefe minimal zu halten, den Baum also möglichst auszugleichen. Dies wird durch ein Rotieren der Knoten erreicht, wodurch auch eine Degeneration vermieden wird. Hier wird der Knoten mit dem Wert 4 zur neuen Wurzel.

Durch die beschriebenen Ausgleichsvorgänge wird die Tiefe des Baums nahezu identisch für den linken und den rechten Teilbaum gehalten – genauer: Die Höhendifferenz überschreitet nie den Wert eins. Damit bleibt die maximale Tiefe immer logarithmisch zur Anzahl der im Baum gespeicherten Elemente. Die Ausgeglichenheit sorgt dafür, die maximale Suchgeschwindigkeit auf logarithmische Komplexität zu begrenzen. Das ermöglicht sehr performante Suchvorgänge: Bei 1.000 Elementen beträgt die Tiefe 10 und definiert damit auch die maximale Anzahl an Suchschritten bis zum Auffinden des gesuchten Elements bzw. zum Erkennen, dass kein solches existiert. Selbst bei 1 Million gespeicherter Elemente sind dadurch maximal 20 Schritte notwendig.

6.1.9 Die Methoden `equals()`, `hashCode()` und `compareTo()` im Zusammenspiel

Nachdem wir nun ein gutes Verständnis zu `HashSet<E>` und `TreeSet<E>` aufgebaut haben, wollen wir noch mal auf Besonderheiten der drei Methoden `equals(Object)`, `hashCode()` und `compareTo(T)` eingehen. Dabei ist es vor allem wichtig, diese Methoden konsistent zueinander zu implementieren, wobei `compareTo(T)` nicht in jedem Fall angeboten werden muss. Wenn es aber existiert, dann sollte es konsistent zu `equals(Object)` sein.[12] Beachtet man die Forderung nach Konsistenz nicht, kann es zu Fehlern kommen, die sich nur schwierig reproduzieren lassen und sich in merkwürdigem Programmverhalten äußern.

Betrachten wir dies anhand der Verwaltung einiger Objekte der folgenden Klasse `SimplePerson` mithilfe der Datenstrukturen `HashSet<SimplePerson>` und `TreeSet<SimplePerson>`. Die Klasse `SimplePerson` implementiert das Interface `Comparable<SimplePerson>` und ist wie folgt definiert:

```
private static class SimplePerson implements Comparable<SimplePerson>
{
    private final String name;

    SimplePerson(final String name)
    {
        this.name = name;
    }

    @Override
    public int compareTo(final SimplePerson other)
    {
        return name.compareTo(other.name);
    }
}
```

Das folgende Listing zeigt, wie zwei inhaltlich gleiche `SimplePerson`-Objekte erzeugt und per `add(SimplePerson)` in einem `HashSet<SimplePerson>` und einem `Tree-`

[12] Ausnahmen davon sind entsprechend zu dokumentieren.

Set<SimplePerson> gespeichert werden. Anschließend ermitteln wir durch Aufruf der Methode size() die Anzahl der gespeicherten Elemente im jeweiligen Container:

```
public static void main(final String[] args)
{
    final Set<SimplePerson> hashSet = new HashSet<>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2

    final Set<SimplePerson> treeSet = new TreeSet<>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
```

Listing 6.13 *Ausführbar als* **'LAWOFBIG3EXAMPLE'**

Zunächst ist überraschend, dass im HashSet<SimplePerson> zwei Elemente vorhanden sind und nicht wie im TreeSet<SimplePerson> nur eins. Wie ist das zu erklären? Das liegt daran, dass die Methode equals(Object), die zur Bestimmung der Gleichheit von Einträgen innerhalb von Buckets verwendet wird, in der Klasse SimplePerson nicht implementiert ist. Somit findet ein Referenzvergleich statt, wenn zwei SimplePerson-Objekte verglichen werden. Für die Klasse TreeSet<SimplePerson> wird beim Hinzufügen zum Ausschluss doppelter Einträge und damit zum Erhalt der Integrität der Menge die Methode compareTo(SimplePerson) anstelle von equals(Object) verwendet. In diesem Beispiel besteht demnach das Problem, dass die Methode compareTo(SimplePerson)[13] nicht mit equals(Object) kompatibel ist, wie dies in Abschnitt 6.1.8 gefordert wurde. Es fehlt eine entsprechende Implementierung von equals(Object) in der Klasse SimplePerson:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit
        return false;

    final SimplePerson otherPerson = (SimplePerson) other;
    return compareTo(otherPerson) == 0;         // Vergleich mit compareTo()
}
```

Listing 6.14 *Ausführbar als* **'LAWOFBIG3EXAMPLE2'**

Ein erneuter Testlauf liefert immer noch zwei Elemente im HashSet<SimplePerson> und eins im TreeSet<SimplePerson>. Wie kann das sein, nachdem wir auch die equals(Object)-Methode korrigiert haben? Überlegen wir kurz.

Die Erklärung ist einfach: Zwar werden nun zwei SimplePerson-Objekte als gleich angesehen, wenn sie denselben Inhalt besitzen, aber zu diesem Vergleich kommt

[13] Das gilt ebenso, wenn ein Comparator<SimplePerson> genutzt wird.

es erst gar nicht. Wie bereits in Abschnitt 6.1.7 angedeutet, berechnet die Methode `hashCode()` zunächst das Bucket zur Speicherung der Objekte. Da keine eigene Implementierung der Methode `hashCode()` existiert, werden die von `equals(Object)` als gleich angesehenen `SimplePerson`-Objekte in unterschiedlichen Buckets gespeichert. Dies widerspricht dem `hashCode()`-Kontrakt und führt dazu, dass das gleiche `SimplePerson`-Objekt zweimal in das `HashSet<SimplePerson>` eingefügt wird. Als Korrektur realisieren wir die `hashCode()`-Methode wie folgt:

```
@Override
public int hashCode()
{
    return name.hashCode();
}
```

Listing 6.15 *Ausführbar als* '**LAWOFBIG3EXAMPLE3**'

Sowohl das `HashSet<SimplePerson>` als auch das `TreeSet<SimplePerson>` enthalten nach dieser Korrektur nur noch einen Eintrag.

Fazit

Dieses einfache Beispiel verdeutlicht das Zusammenspiel der drei Methoden und die Notwendigkeit, die Forderungen der jeweiligen Methodenkontrakte einzuhalten, um Überraschungen oder Merkwürdigkeiten zu vermeiden.

6.1.10 Schlüssel-Wert-Abbildungen und das Interface `Map`

Nachdem wir bisher die konkreten Realisierungen des Interface `Collection<E>` besprochen haben, wenden wir uns nun den davon unabhängigen Implementierungen des Interface `Map<K,V>` zu. Sie realisieren, wie bereits erwähnt, Abbildungen von Schlüsseln auf Werte. Häufig werden Maps deshalb auch als ***Dictionary*** oder ***Lookup-Tabelle*** bezeichnet.

Die zugrunde liegende Idee ist, jedem gespeicherten Wert einen eindeutigen Schlüssel zuzuordnen. Ein intuitiv verständliches Beispiel sind Telefonbücher: Hier werden Namen auf Telefonnummern abgebildet. Eine Suche über einen Namen (Schlüssel) liefert meistens recht schnell eine Telefonnummer (Wert). Da jedoch keine Rückabbildung von Telefonnummer auf Name existiert, wird das Ermitteln eines Namens zu einer Telefonnummer recht aufwendig.

Das Interface `Map`

Maps speichern Schlüssel-Wert-Paare. Jeder Eintrag wird durch das innere Interface `Map.Entry<K,V>` repräsentiert, das die Abbildung zwischen Schlüsseln (Typparameter `K`) und Werten (Typparameter `V`) realisiert. Die Methoden im Interface `Map<K,V>` sind daher auf diese spezielle Form der Speicherung von Schlüssel-Wert-Abbildungen ausgelegt, ähneln aber denen des Interface `Collection<E>`. Allerdings

bildet `Collection<E>` nicht die Basis von `Map<K,V>`, das vielmehr einen davon unabhängigen Basistyp darstellt.

Das Interface `Map<K,V>` bietet unter anderem folgende Methoden:

- `V put(K key, V value)` – Fügt dieser Map eine Abbildung (Schlüssel auf Wert) als Eintrag hinzu. Falls zu dem übergebenen Schlüssel bereits ein Wert gespeichert ist, so wird dieser mit dem neuen Wert überschrieben. Die Methode gibt den zuvor mit diesem Schlüssel verbundenen Wert zurück, sofern es einen derartigen Eintrag gab, ansonsten wird `null` zurückgegeben.
- `void putAll(Map<? extends K, ? extends V> map)` – Fügt alle Einträge aus der übergebenen Map in diese Map ein. Werte bereits existierender Einträge werden, analog zur Arbeitsweise der Methode `put(K,V)`, überschrieben.
- `V remove(Object key)` – Löscht einen Eintrag (Schlüssel und zugehörigen Wert) aus der Map. Als Rückgabe erhält man den zum Schlüssel `key` gehörenden Wert oder `null`, wenn zu diesem Schlüssel kein Eintrag gespeichert war.
- `V get(Object key)` – Ermittelt zu einem Schlüssel `key` den assoziierten Wert. Existiert kein Eintrag zu dem Schlüssel, so wird `null` zurückgegeben.
- `boolean containsKey(Object key)` – Prüft, ob der Schlüssel `key` in der Map gespeichert ist, und liefert genau dann `true`, wenn dies der Fall ist.
- `boolean containsValue(Object value)` – Prüft, ob der Wert `value` in der Map gespeichert ist, und liefert genau dann `true`, wenn dies der Fall ist.
- `void clear()` – Löscht alle Einträge der Map.
- `int size()` – Ermittelt die Anzahl der in der Map gespeicherten Einträge.
- `boolean isEmpty()` – Prüft, ob die Map leer ist.

Zum Umgang mit dem Wert `null` als Schlüssel oder Wert beachten Sie bitte den nachfolgenden Praxistipp.

Ergänzend zu den gerade vorgestellten Methoden gibt es folgende Methoden, die Zugriff auf gespeicherte Schlüssel, Werte und Einträge bieten:

- `Set<K> keySet()` – Liefert eine Menge mit allen Schlüsseln.
- `Collection<V> values()` – Liefert die Werte in Form einer Collection.
- `Set<Map.Entry<K,V>> entrySet()` – Liefert die Menge aller Einträge. Dadurch hat man sowohl Zugriff auf die Schlüssel als auch auf die Werte.

Diese drei Methoden liefern jeweils Sichten auf die Daten. Erfolgen Veränderungen in der zugrunde liegenden Map, so werden diese in den Sichten widergespiegelt. ***Beachten Sie bitte, dass Änderungen in der jeweiligen Sicht ebenfalls in die Map übertragen werden***. Ähnliches haben wir für Listen und Sets kennengelernt.

Tipp: Der Wert `null` als Schlüssel und als Wert

Liefert die Methode `get(Object)` den Wert `null`, so wird dies vielfach als Nichtvorhandensein eines Eintrags in der Map gedeutet. Diese Schlussfolgerung ist allerdings nicht immer korrekt: In einigen Realisierungen des Interface `Map<K,V>` ist `null` als Wert und sogar als Schlüssel erlaubt. Für `null`-Werte kann man dadurch die Fälle »kein Wert« und »Speicherung des Werts `null`« anhand der Rückgabe von `get()` nicht voneinander unterscheiden. Für diesen Zweck gibt es die Methode `containsKey(Object)`.

Beispiel: Maps im Einsatz

Bevor wir uns die konkreten Realisierungen des Interface `Map<K,V>` anschauen, wollen wir durch ein kleines Beispiel ein wenig vertrauter mit Maps werden. Wir bauen eine Art Telefonbuch nach bzw. realisieren eine Abbildung von `String` auf `Integer`:

```
public static void main(final String[] args)
{
   final Map<String, Integer> nameToNumber = new TreeMap<>();
   nameToNumber.put("Micha", 4711);
   nameToNumber.put("Tim", 0714);
   nameToNumber.put("Jens", 1234);
   nameToNumber.put("Tim", 1508);    // Zweites put() für "Tim"
   nameToNumber.put("Ralph", 2208);

   // Verschiedene Aktionen ausführen
   System.out.println(nameToNumber);
   System.out.println(nameToNumber.containsKey("Tim")); // Prüfe Schlüssel
   System.out.println(nameToNumber.get("Jens"));        // Zugriff per Schlüssel
   System.out.println(nameToNumber.size());             // Anzahl der Einträge
   System.out.println(nameToNumber.keySet());           // Alle Schlüssel
   System.out.println(nameToNumber.values());           // Alle Werte
}
```

Listing 6.16 *Ausführbar als* **'FIRSTMAPEXAMPLE'**

Starten wir das Programm FIRSTMAPEXAMPLE, so kommt es zu folgenden Ausgaben, die uns schon ein paar Dinge über Maps verraten, nämlich etwa, dass Werte überschrieben werden, wenn mehrmals Daten zum gleichen Schlüssel eingefügt werden:

```
{Jens=1234, Micha=4711, Ralph=2208, Tim=1508}
true
1234
4
[Jens, Micha, Ralph, Tim]
[1234, 4711, 2208, 1508]
```

Die Klasse HashMap<K,V>

Die Klasse HashMap<K,V> ist eine Realisierung der abstrakten Klasse AbstractMap<K,V>, die das Interface Map<K,V> implementiert. Die Datenhaltung geschieht in einer Hashtabelle und ermöglicht dadurch eine effiziente Ausführung gebräuchlicher Operationen wie get(Object), put(K,V), containsKey(Object) und size(). Die Reihenfolge der Elemente bei einer Iteration wirkt zufällig. Tatsächlich wird sie durch den jeweiligen Hashwert sowie die Verteilung auf die Buckets bestimmt, wie dies bereits für das HashSet<E> besprochen wurde (vgl. Abschnitt 6.1.7).

Beispiel: Lookup-Map Zur Demonstration der Klasse HashMap<K,V> wollen wir einen in der Praxis häufig anzutreffenden Anwendungsfall betrachten, bei dem eine Menge von Eingabewerten auf eine Menge von Ausgabewerten abgebildet werden soll. Dazu sieht man häufig if- oder switch-Anweisungen wie die folgende:

```
private static Color mapToColor(final String colorName)
{
    switch (colorName)
    {
    case "BLACK":
        return Color.BLACK;
    case "RED":
        return Color.RED;
    case "GREEN":
        return Color.GREEN;
    // ... viele mehr ...
    default:
        throw new IllegalArgumentException("No color for: '" + colorName + "'");
    }
}
```

Sind nur ein paar wenige Fälle abzudecken, kann diese Realisierung durchaus akzeptabel sein, je mehr Fälle jedoch aufeinander abgebildet werden sollen, desto umfangreicher und schwieriger wartbar werden solche Konstrukte. Als Abhilfe kann man sich eine ***Abbildungstabelle*** in Form einer HashMap<K,V> definieren und die Abbildung wird durch einen Zugriff mit dem entsprechenden Schlüssel realisiert:

```
private static final Map<String, Color> nameToColor = new HashMap<>();

public static void main(final String[] args)
{
    initMapping(nameToColor);

    System.out.println(mapToColor("RED")); // java.awt.Color[r=255,g=0,b=0]
    System.out.println(mapToColor("GREEN"));   // java.awt.Color[r=0,g=255,b=0]
    System.out.println(mapToColor("UNKNOWN")); // => Exception
}

private static void initMapping(final Map<String, Color> nameToColor)
{
    nameToColor.put("BLACK", Color.BLACK)};
    nameToColor.put("RED", Color.RED);
    nameToColor.put("GREEN", Color.GREEN);
    // ... viele mehr ...
}
```

```
private static Color mapToColor(final String colorName)
{
    if (!nameToColor.containsKey(colorName))
        throw new IllegalArgumentException("No color for: '" + colorName + "'");

    return nameToColor.get(colorName);
}
```

Listing 6.17 *Ausführbar als* '**HASHMAPLOOKUPEXAMPLE**'

Diese Art der Realisierung hält die Funktionalität der Abbildung in der Applikation selbst kurz, lesbar und übersichtlich. Hier im Beispiel wird aus Gründen der Einfachheit eine statische Definition und Initialisierung genutzt. Sofern benötigt kann die Initialisierung auch ausgelagert werden und mithilfe einer externen Datenquelle, etwa einer Datei, erfolgen. Dadurch erzielt man eine größere Flexibilität.

Die Klasse `LinkedHashMap<K,V>`

Die Klasse `LinkedHashMap<K,V>` bietet die Funktionalität einer `HashMap<K,V>` und erweitert diese um die Möglichkeit, Elemente in einer definierten Reihenfolge (wahlweise Einfüge- bzw. Zugriffsreihenfolge) zu speichern und abrufen zu können.

Zum einen kann dies nützlich sein, wenn man eine feste Reihenfolge bei der Iteration benötigt – für `HashMap<K,V>` ist die Ausgabe recht willkürlich. Zum anderen – und für die Praxis relevanter – lässt sich mithilfe der Klasse `LinkedHashMap<K,V>` auf einfache Weise ein Zwischenspeicher, auch ***Cache*** genannt, realisieren. Ein solcher ist immer dann nützlich, wenn man beispielsweise wiederholt auf Daten aus dem Dateisystem oder einer Datenbank zugreift. Diese Zugriffe sind teuer, d. h., sie sind aufwendig und führen durch Latenzzeiten auch zu Verzögerungen in der Abarbeitung des Programms. Als Optimierung kann man Caches für die relevantesten Daten im Hauptspeicher halten, um auf diese direkt zugreifen zu können. Häufig sind das die zuletzt zugegriffenen Daten.

Steuerung durch Callback-Methode Die Klasse `LinkedHashMap<K,V>` bietet die folgende Callback-Methode, die beim Einfügen von Elementen aufgerufen wird:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
```

Der Rückgabewert bestimmt, ob das jeweils älteste Element aus der Map entfernt werden soll. Die Defaultimplementierung dieser Methode liefert den Wert `false` und sorgt damit dafür, dass beim Hinzufügen von Elementen kein Element gelöscht wird. Soll dieses Verhalten geändert werden, so muss die Methode überschrieben und für zu löschende Elemente der Wert `true` zurückgegeben werden. Das Löschen geschieht dann automatisch durch die Implementierung der Map selbst.

Im Folgenden betrachten wir zunächst die Realisierung einer Größenbeschränkung, wobei hier das älteste Element anhand der Reihenfolge des Einfügens bestimmt wird. Neuere Daten verdrängen so früher eingefügte.

Hinweis: Aussagekräftige Methodennamen im API

Da die Methode `removeEldestEntry(Map.Entry<K,V> eldest)` kein Element löscht, sondern lediglich bestimmt, ob dies geschehen soll, hätte man sie besser `shouldRemoveEldestEntry(Map.Entry<K,V> eldest)` genannt.

Beispiel: Realisierung einer Größenbeschränkung Mithilfe der gerade vorgestellten Callback-Methode kann man leicht eine in ihrer Größe beschränkte Map implementieren, die ältere Elemente entfernt, wenn eine gewisse Größe überschritten ist und dann Elemente eingefügt werden. Die folgende Klasse `FixedSizeLinkedHashMap<K,V>` zeigt, wie einfach eine derartige Größenbeschränkung zu realisieren ist:

```
public final class FixedSizeLinkedHashMap<K, V> extends LinkedHashMap<K, V>
{
    private final int maxEntryCount;

    public FixedSizeLinkedHashMap(final int maxEntryCount)
    {
        this.maxEntryCount = maxEntryCount;
    }

    @Override
    protected boolean removeEldestEntry(final Map.Entry<K, V> customer)
    {
        return size() > maxEntryCount;
    }
}
```

Um die Größenbeschränkung umzusetzen, muss man lediglich die Methode `removeEldestEntry(Map.Entry<K,V>)` überschreiben und dazu prüfen, ob die Anzahl der gespeicherten Elemente eine zuvor festgelegte Größe übersteigt. Da hier keine anderweitige Parametrisierung der zugrunde liegenden `LinkedHashMap<K,V>` erfolgt, wird das älteste Element anhand der Reihenfolge des Einfügens bestimmt. Bei Überschreiten der angegebenen Größe wird das laut Einfügereihenfolge älteste, d. h. das zuerst eingefügte Element gelöscht und damit die Größenbeschränkung erhalten.

Im nachfolgenden Beispiel wird die Größe des realisierten Containers zu Demonstrationszwecken auf den Wert 3 festgelegt. Anschließend werden fünf Abbildungen von Namen auf `Customer`-Objekte in der Map abgelegt.

```
public static void main(final String[] args)
{
    // Größenbeschränkung auf drei Elemente
    final int MAX_ELEMENT_COUNT = 3;
    final FixedSizeLinkedHashMap<String, Customer> fixedSizeMap =
             new FixedSizeLinkedHashMap<>(MAX_ELEMENT_COUNT);

    // Initial befüllen
    fixedSizeMap.put("Erster", new Customer("Erster", "Stuhr", 11));
    fixedSizeMap.put("Zweiter", new Customer("Zweiter", "Hamburg", 22));
    fixedSizeMap.put("M. Inden", new Customer("Inden", "Aachen", 39));
    printCustomerList("Initial", fixedSizeMap.values());
```

```
    // Änderungen durchführen und ausgeben
    fixedSizeMap.put("New1", new Customer("New_1", "London", 44));
    printCustomerList("After insertion of 'New_1'", fixedSizeMap.values());

    fixedSizeMap.put("New2", new Customer("New_2", "San Francisco", 55));
    printCustomerList("After insertion of 'New_2'", fixedSizeMap.values());
}

private static void printCustomerList(final String title,
                                      final Collection<Customer> customers)
{
    System.out.println(title);
    customers.forEach(System.out::println); // Java-8-Defaultmethode
}
```

Listing 6.18 *Ausführbar als* '**FIXEDSIZELINKEDHASHMAPEXAMPLE**'

Führt man das Programm FIXEDSIZELINKEDHASHMAPEXAMPLE aus, wird die Ersetzungsstrategie deutlich: Die beiden zuerst eingefügten Elemente "Erster" und "Zweiter" werden durch die neu hinzugefügten Elemente "New_1" und "New_2" verdrängt:

```
[...]
After insertion of 'New_1'
Customer [name=Zweiter, city=Hamburg, age=22]
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
After insertion of 'New_2'
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
Customer [name=New_2, city=San Francisco, age=55]
```

Beispiel: Realisierung eines LRU-Caches Statt die Reihenfolge des Einfügens als Verbleibkriterium zu nutzen, ist es oftmals sinnvoller, zu betrachten, welche Elemente zuletzt verwendet wurden. Man realisiert dazu einen sogenannten ***LRU-Cache*** (Least-Recently-Used), der die zuletzt benutzten Objekte zwischenspeichert, indem er die am längsten nicht mehr zugegriffenen Elemente im Cache ersetzt:

```
public final class LruLinkedHashMap<K, V>  extends LinkedHashMap<K, V>
{
    // Kopie der Package-privaten Definitionen aus der Klasse HashMap
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    private static final float DEFAULT_LOAD_FACTOR = 0.75f;
    private static final boolean USE_ACCESS_ORDER = true;

    private final int maxEntryCount;

    public LruLinkedHashMap(final int maxEntryCount)
    {
        // Unschön: Um die Eigenschaft accessOrder anzugeben, müssen wir Werte
        // an den Konstruktor übergeben, die wir nicht spezifizieren wollen
        super(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR, USE_ACCESS_ORDER);
        this.maxEntryCount = maxEntryCount;
    }
```

Bitte beachten Sie Folgendes: Um die Eigenschaft der Zugriffsreihenfolge überhaupt setzen zu können, ist man durch die Konstruktoren der `LinkedHashMap<K,V>` dazu gezwungen, die Werte für Initialkapazität und Füllgrad (Load Factor) anzugeben.

Die Methode `removeEldestEntry(Map.Entry<K,V>)` implementieren wir folgendermaßen:

```
    @Override
    protected boolean removeEldestEntry(final Map.Entry<K, V> customer)
    {
        return size() > maxEntryCount;
    }
}
```

Zur Verdeutlichung der Arbeitsweise der Klasse `LruLinkedHashMap` speichern wir dort wieder einige `Customer`-Objekte. Danach wird dann nur auf drei der vier zuvor gespeicherten Einträge zugegriffen. Durch das Einfügen eines weiteren Eintrags wird der am längsten nicht verwendete Eintrag ersetzt. Im folgenden Beispiel wird der Eintrag »`M. Inden`« durch den Eintrag »`D. Dummy`« ersetzt.

```
public static void main(final String[] args)
{
   // Größenbeschränkung auf vier Elemente
   final int MAX_ELEMENT_COUNT = 4;
   final LruLinkedHashMap<String, Customer> lruMap =
                                   new LruLinkedHashMap<>(MAX_ELEMENT_COUNT);

   lruMap.put("A. Mustermann", new Customer("A. Mustermann", "Stuhr", 16));
   lruMap.put("B. Mustermann", new Customer("B. Mustermann", "Hamburg", 32));
   lruMap.put("C. Mustermann", new Customer("C. Mustermann", "Zürich", 64));
   lruMap.put("M. Inden", new Customer("M. Inden", "Kiel", 32));

   printCustomerList("Initial", lruMap.values());

   // Zugriff auf alle bis auf M. Inden
   lruMap.get("A. Mustermann");
   lruMap.get("B. Mustermann");
   lruMap.get("C. Mustermann");

   // Neuer Eintrag sollte M. Inden ersetzen
   lruMap.put("Dummy", new Customer("D. Dummy", "Oldenburg", 128));

   printCustomerList("Nach Zugriffen", lruMap.values());
}
```

Listing 6.19 *Ausführbar als* '**LRULINKEDHASHMAPEXAMPLE**'

Ein Start des Programms LRULINKEDHASHMAPEXAMPLE produziert die hier gekürzte Ausgabe:

```
[...]
Nach Zugriffen
Customer [name=A. Mustermann, city=Stuhr, age=16]
Customer [name=B. Mustermann, city=Hamburg, age=32]
Customer [name=C. Mustermann, city=Zürich, age=64]
Customer [name=D. Dummy, city=Oldenburg, age=128]
```

Die Klasse TreeMap<K,V>

Die Klasse TreeMap<K,V> ist eine Erweiterung der abstrakten Klasse AbstractMap<K,V> und implementiert das Interface SortedMap<K,V>. Eine TreeMap<K,V> stellt automatisch die Ordnung der gespeicherten Schlüssel her und nutzt dazu entweder das Interface Comparable<T> oder einen im Konstruktor übergebenen Comparator<T>. Außerdem implementiert die Klasse TreeMap<K,V> das Interface NavigableMap<K,V>, das einige nützliche Methoden definiert: Beispielsweise erhält man durch Aufruf der Methode ceilingKey(K) einen passenden Schlüssel, der größer oder gleich dem übergebenen Schlüssel ist. Korrespondierende Methoden floorKey(K), lowerKey(K) und higherKey(K) liefern Schlüssel, die kleiner oder gleich, kleiner und größer als der angegebene Schlüssel sind. Weiterhin kann man dazugehörende Einträge der Map über korrespondierende xyzEntry(K)-Methoden ermitteln, wobei xyz für lower, higher usw. steht.

Beispiel In folgendem Beispiel nutzen wir die genannten Methoden, um eine Abbildung von Namen auf das Alter zu erreichen und passende Schlüssel bzw. Einträge zu einem übergebenen Namenskürzel zu ermitteln:

```
public static void main(final String[] args)
{
    final NavigableMap<String, Integer> nameToAgeMap = new TreeMap<>();
    nameToAgeMap.put("Max", 47);
    nameToAgeMap.put("Moritz", 39);
    nameToAgeMap.put("Micha", 43);

    System.out.println("floor   Ma: " + nameToAgeMap.floorKey("Ma"));
    System.out.println("higher  Ma: " + nameToAgeMap.higherKey("Ma"));
    System.out.println("lower   Mz: " + nameToAgeMap.lowerKey("Mz"));
    System.out.println("ceiling Mc: " + nameToAgeMap.ceilingEntry("Mc"));
}
```

Listing 6.20 *Ausführbar als* **'TREEMAPEXAMPLE'**

Führt man das Programm TREEMAPEXAMPLE aus, so erhält man diese Ausgabe:

```
floor   Ma: null
higher  Ma: Max
lower   Mz: Moritz
ceiling Mc: Micha=43
```

Die Ausgaben verdeutlichen die vorangegangenen Beschreibungen: Der Aufruf von floorKey("Ma") liefert den Vorgängerschlüssel von "Ma". Weil dieser nicht existiert, wird null zurückgeliefert. Ein Aufruf von higherKey("Ma") liefert den Nachfolgerschlüssel von "Ma" und dies ist der Schlüssel "Max". Mit lowerKey("Mz") wird der Vorgänger von "Mz" ermittelt, was hier "Moritz" ist. Schließlich liefert ceilingEntry("Mc") den Nachfolger von "Mc".

6.1.11 Erweiterungen am Beispiel der Klasse `HashMap`

Manchmal soll eine bestehende Containerklasse um Funktionalität erweitert werden: Nachfolgend wollen wir exemplarisch für die Klasse `HashMap<K,V>` die Normierung von Schlüsseln ergänzen. Das dient z. B. dazu, Benutzereingaben dezent zu korrigieren, etwa führende oder abschließende Leerzeichen zu entfernen, damit nach einheitlichen Schlüsseln gesucht werden kann. Stellen Sie sich als Beispiel vor, man würde Bilder basierend auf Eingaben aus einem GUI referenzieren wollen. Werden Benutzereingaben ungeprüft, unbearbeitet und ohne Korrekturen direkt zur Abfrage als Schlüssel einer Map verwendet, so ist es nicht möglich, gespeicherte Werte zuverlässig wiederzufinden. Fehler treten beispielsweise dann auf, wenn man ein Wort oder einen Buchstaben kleinschreibt oder die Eingabe versehentlich ein führendes oder nachfolgendes Leerzeichen enthält. Sollen textuelle Werte als Referenz auf Schlüssel einer Map dienen, ist es daher wichtig, eine konsistente Umwandlung oder Normalisierung (z. B. Abschneiden von Leerzeichen) der eingegebenen Texte in eine festgelegte Darstellungsform (etwa komplett in Großbuchstaben) zu definieren.

Lösungsvarianten

Weil die Schlüssel aus ungeprüften Benutzereingaben stammen, drohen Inkonsistenzen. Das soll durch eine konsistente Normalisierung von Schlüsseln in eine festgelegte Darstellungsform adressiert werden. Weiterhin ist es wünschenswert, dies in der zu erstellenden Containerklasse einmal zentral zu realisieren. Zudem soll die Namensabbildung für nutzende Applikationen unsichtbar und ohne Aufwand einsetzbar sein. Um die gewünschten Erweiterungen umzusetzen, existieren folgende zwei Alternativen:

1. ***Aggregation*** einer Containerklasse und ***Delegation*** an deren Methoden
2. ***Ableitung*** von einer Containerklasse und ***Überschreiben*** von Methoden

Aggregation und Delegation Verwendet man Aggregation und Delegation, so muss die benötigte Funktionalität über Methodenaufrufe an die aggregierte Containerklasse selbst programmiert werden. Eine Realisierung könnte wie folgt aussehen:

```
public final class NameToImageMapUsingDelegation
{
    private final Map<String, Image> nameToImage = new HashMap<>();

    public void put(final String name, final Image image)
    {
        final String key = name != null ? name.toUpperCase().trim() : null;
        nameToImage.put(key, image);
    }

    public Image get(final String name)
    {
        final String key = name != null ? name.toUpperCase().trim() : null;
        return nameToImage.get(key);
    }
```

```
    public void clear()
    {
        nameToImage.clear();
    }
}
```

Diese Art der Realisierung besitzt folgende Auswirkungen:

- Vielfach wird in der Praxis – wie im Beispiel auch – zunächst nur diejenige Funktionalität bereitgestellt, die man initial wirklich benötigt. Ist später mehr Containerfunktionalität erforderlich, so muss diese passend realisiert werden. Im Speziellen kann man aber auch bewusst gewisse Methoden in der Schnittstelle *nicht* anbieten,[14] etwa Löschoperationen.
- Ein derart realisierte Klasse erschwert die Handhabung, weil sie nicht gut mit dem Collections-Framework harmoniert: Da weder ein Basisinterface erfüllt noch eine Basisklasse erweitert wird, etwa `Map<K,V>`, lässt sich die von der Utility-Klasse `Collections` angebotene Funktionalität nicht oder nur eingeschränkt nutzen.
- Weil eine eigene Schnittstelle und kein Basisinterface aus dem Collections-Framework genutzt wird, besteht ein Handicap darin, dass eben nicht die bekannten Methoden angeboten werden oder Methodensignaturen (auch stärker) abweichen. Dies führt aber zu Inkompatibilitäten mit dem Collections-Framework.

Ableitung und Überschreiben Seit JDK 5 kann man typsichere Containerklassen eleganter durch eine Kombination von Ableitung und Einsatz von Generics realisieren und bleibt kompatibel zu allen Funktionalitäten, die durch die Utility-Klasse `Collections` zur Verfügung gestellt werden.

Folgendes Beispiel zeigt den ersten Versuch, die geforderte Funktionalität auf Basis einer typsicheren `HashMap<String,Image>` umzusetzen:

```
// ACHTUNG: Fehlerhafter erster Versuch!
public final class NameToImageMap extends HashMap<String, Image>
{
    @Override
    public Image put(final String name, final Image image)
    {
        return super.put(name.toUpperCase().trim(), image);
    }

    // @Override => nicht möglich da Signatur get(Object)
    public Image get(final String name)
    {
        return super.get(name.toUpperCase().trim());
    }
    // ...
}
```

[14] Bei einer Realisierung als Subklasse kann man dies nur durch Überschreiben und Auslösen von z. B. einer `UnsupportedOperationException` in der Implementierung ausdrücken.

Auf den ersten Blick ist kein Fehler zu erkennen. Tatsächlich enthält diese Art der Umsetzung jedoch folgende Probleme:

1. Die Realisierung unterstützt keine `null`-Werte als Schlüssel, sondern führt stattdessen zu einer `NullPointerException`. ***Damit verstößt diese Umsetzung gegen die Methodenkontrakte und verhält sich nicht korrekt wie eine Subklasse***. Damit verletzt sie das in Abschnitt 3.5.3 vorgestellte LISKOV SUBSTITUTION PRINCIPLE (LSP). In diesem Fall lässt sich das Problem dadurch lösen, dass man eine Hilfsmethode `normalizeKey(String)` erstellt:

```
private String normalizeKey(final String key)
{
    if (key == null)
        return null;

    return key.toUpperCase().trim();
}
```

2. Die oben im Listing gezeigte Methode `put(String,Image)` ist korrekt überschrieben, die Methode `get(Object)` jedoch nicht! Hier findet vielmehr ein versehentliches Überladen von `get(Object)` statt. Ohne Nutzung der Annotation `@Override` kann das leicht passieren, da im Collections-Framework leider einige Methoden mit Parametern vom Typ `Object` statt des Typs `K` des Schlüssels definiert sind. Diese Besonderheit gilt auch für `get()`-Methoden und erfordert Vorsicht, um Typfehler zu vermeiden. Um Fehler beim Überschreiben durch den Compiler aufdecken zu können, bietet es sich an, ***alle Methoden, die man überschreiben möchte, mit der Annotation `@Override` zu kennzeichnen***.
3. Damit sich die Klasse korrekt als Spezialisierung verhält, müssen alle Methoden angepasst werden, die einen Schlüssel als Parameter erwarten. Geschieht dies nicht, kann man ansonsten zwar problemlos Elemente speichern, eine Abfrage über `containsKey(Object)` oder ein Löschen über `remove(Object)` würde jedoch nicht richtig arbeiten. Ohne Anpassungen in einer Subklasse werden lediglich die Methoden der Oberklasse aufgerufen. ***Es wird zuvor eine Umwandlung der Schlüssel benötigt, um garantiert mit passenden Schlüsseln zu suchen***.

Verallgemeinert nutzt man statt des Typs `Image` beliebige Typen mit dem Typkürzel `V`. Zudem werden alle Methoden, die auf Schlüssel zugreifen, entsprechend angepasst, wodurch sich die Klasse wie eine Spezialisierung einer `HashMap<K,V>` verhält, die als Besonderheit jedoch die Schlüssel normalisiert. Folgende Klasse `UpperCaseNormalizedHashMap<V>` behebt die angesprochenen Mängel:

```
public final class UpperCaseNormalizedHashMap<V> extends HashMap<String, V>
{
    @Override
    public V put(final String key, final V value)
    {
        return super.put(normalizeKey(key), value);
    }

    @Override
    public V get(final Object key)
    {
        return super.get(normalizeKey((String) key));
    }

    @Override
    public boolean containsKey(final Object key)
    {
        return super.containsKey(normalizeKey((String) key));
    }

    // ...

    private String normalizeKey(final String key)
    {
        if (key == null)
            return null;

        return key.toUpperCase().trim();
    }
}
```

Diese Klasse erfüllt die Anforderungen, passt sich ins Collections-Framework ein und stellt somit eine gelungenere Realisierung als diejenige durch Aggregation dar.

6.1.12 Erweiterungen im Interface `Map` in JDK 8

Das Interface `Map<K,V>` wurde in JDK 8 erweitert. Anhand einer konkreten Aufgabenstellung wollen wir nachvollziehen, wie wir einige der Methoden im Interface `Map<K,V>` gewinnbringend einsetzen können.

Nehmen wir an, wir müssten für eine Liste von Wörtern deren Häufigkeiten bestimmen. Weil uns die dazu benötigten Testdaten in einigen Listings begleiten werden, zeige ich zunächst einmalig die Methode, die diese Werte bereitstellt:

```
private static List<String> createTestData()
{
    final List<String> wordList = Arrays.asList("Dies", "ist", "eine", "Liste",
                        "Eine", "Liste", "kann", "Worte", "enthalten",
                        "Dies", "ist", "das", "Ende", "der", "Liste");
    return wordList;
}
```

Realisierung mit JDK 7 oder früher

Als Beispiel sollen die Häufigkeiten von Wörtern in einem Text ermittelt und eine Art Histogramm erstellt werden. Zunächst zeige ich, wie man dies herkömmlich mithilfe einer `Map<K,V>` ausprogrammieren könnte:

```
final List<String> wordList = createTestData();

final Map<String, Integer> wordCounts = new TreeMap<>();
for (final String word : wordList)
{
    // Wortvorkommen hoch zählen bzw. anlegen, wenn zuvor nicht existent
    if (wordCounts.containsKey(word))
    {
        final Integer oldValue = wordCounts.get(word);
        wordCounts.put(word, oldValue + 1);
    }
    else
    {
        wordCounts.put(word, 1);
    }
}

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

Wir sehen die Behandlung verschiedener Sonderfälle, beispielsweise wenn kein Wert vorhanden ist, sowie das Auslesen des alten und Setzen des neuen Werts. Das wirkt bereits ein wenig unelegant. Insbesondere problematisch sind zwei Dinge: Erstens muss man diese Funktionalität für andere, ähnliche Anwendungsfälle immer wieder erneut ausprogrammieren, im Speziellen auch dann, wenn lediglich andere Typen für Schlüssel oder Wert genutzt werden. Zweitens ist eine derartige Verarbeitung kritisch, falls zwischenzeitlich durch andere Threads Änderungen erfolgen, sodass im Anschluss ein anderer Zustand existiert als vor der Unterbrechung und z. B. bei der Prüfung. Natürlich kann man durch Synchronisierung für einen kritischen Bereich und die exklusive Ausführung sorgen, dies geschieht jedoch auf Kosten der Möglichkeit zur parallelen Abarbeitung. Eine weitere Alternative wären Locks.

Zusammenfassend lässt sich feststellen, dass man diese Erweiterungen zwar relativ einfach in ihrer Funktionalität nachbauen kann, es jedoch eher schwierig ist, dies Thread-sicher und bei konkurrierenden Zugriffen korrekt hinzubekommen. Umso angenehmer ist es, dass diese Methoden von der für Multithreading und konkurrierende Zugriffe ausgelegten Klasse `ConcurrentHashMap<K,V>` angeboten werden. Nachfolgend wollen wir uns vor allem um Lesbarkeit und Verständlichkeit und weniger um Multithreading kümmern. Lernen wir also einige neue Methoden im Interface `Map<K,V>` kennen.

Die Methode `getOrDefault()`

Oftmals wünscht man sich beim Zugriff auf eine Map, dass ein Defaultwert zurückgeliefert werden kann, falls kein Eintrag zu einem gewünschten Schlüssel existiert. Diese Funktionalität wird in JDK 8 durch die Methode `getOrDefault(Object,V)` realisiert. Man vermeidet dadurch ansonsten notwendige Spezialbehandlungen. Eine Utility-Methode, die analog arbeitet, könnte man wie folgt realisieren:

```
// Achtung: Nur für Singlethreading korrekt
public static <K,V> V getOrDefaultSimplified(final Map<K,V> map, final K key,
                                             final V defaultValue)
{
    if (!map.containsKey(key))
        return defaultValue;

    return map.get(key);
}
```

Die Methoden `putIfAbsent()` und `replace()`

Im Interface `Map<K,V>` konnte man bis JDK 8 durch Aufrufe von `put(K,V)` Werte zu einem Schlüssel sowohl in die Map einfügen als auch einen bereits existierenden Wert überschreiben. In JDK 8 werden nun mit `putIfAbsent(K,V)` und `replace(K,V)` zwei neue, speziellere Funktionen geboten. Wie bereits am Namen zu vermuten ist, fügt `putIfAbsent(K,V)` nur dann einen Wert ein, wenn zuvor noch keiner existierte. Für `replace(K,V)` gilt es andersherum: Mit der Methode `replace(K,V)` werden lediglich schon vorhandene Einträge ersetzt. Existiert kein Wert, passiert nichts.

Beispiel Wörterzählen Die drei Methoden `getOrDefault(Object,V)`, `putIfAbsent(K,V)` und `replace(K,V)` kombinieren wir für das Wörterzählen wie folgt in einem kürzeren Sourcecode, der jedoch möglicherweise nicht intuitiv verständlich ist. Insbesondere muss beim ersten Hochzählen ein wenig getrickst werden:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        // Initialen Wert vorgeben, Achtung 0, weil später Inkrement erfolgt
        wordCounts.putIfAbsent(word, 0);
        // Wert ermitteln, wenn vorhanden
        final Integer value = wordCounts.getOrDefault(word, 0);
        // Wert ersetzen
        wordCounts.replace(word, value + 1);
    }

    System.out.println(wordCounts);
}
```

Listing 6.21 *Ausführbar als* '**WORDCOUNTPUTIFABSENTREPLACEEXAMPLE**'

Die Methoden `computeIfAbsent()` und `computeIfPresent()`

Manchmal soll nicht nur ein ganz bestimmter Wert mit `putIfAbsent(K,V)` in eine Map eingefügt werden, sondern stattdessen eine Berechnung ausgeführt werden. Das kann man unter anderem dazu nutzen, um eine sogenannte Multi Map zu realisieren, bei der für einen Schlüssel mehrere Werte gespeichert werden können. Existiert noch kein Wert für einen Schlüssel, so muss zunächst eine Collection angelegt werden. Das implementieren wir wie folgt:

```
// Achtung: Nur für Singlethreading korrekt
if (!map.containsKey(key))
{
    map.put(new ArrayList<>());
}
```

Die obige Realisierung ist nur für Singlethreading korrekt, bei Multithreading könnten mehrere Threads zeitgleich die Prüfung vornehmen und danach unterbrochen werden. Nachfolgendes Hinzufügen von Elementen wird dann möglicherweise durch frisch initialisierte `ArrayList<E>`-Instanzen wieder zunichtegemacht. Die Standardimplementierung als Defaultmethode `computeIfAbsent(K, Function<? super K, ? extends V>)` im Interface `Map<K,V>` löst dieses Problem zwar nicht, verbessert aber die Lesbarkeit, insbesondere in Kombination mit einem Lambda (vgl. Kapitel 4).

```
map.computeIfAbsent(key, it -> new ArrayList<>());
```

Beispiel Wörterzählen Für das Beispiel des Wörterzählens kann man die Verarbeitung mithilfe von Lambdas deutlich klarer wie folgt schreiben:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        wordCounts.computeIfPresent(word, (str, val) -> val + 1);
        wordCounts.computeIfAbsent(word, (val) -> 1);
        // Alternativ: wordCounts.putIfAbsent(word, 1);
    }

    System.out.println(wordCounts);
}
```

Listing 6.22 *Ausführbar als* **'WORDCOUNTCOMPUTEIFEXAMPLE'**

Zum Erhöhen des Zählers nutzen wir hier im Aufruf von `computeIfPresent()` einen Lambda, der als Eingabe sowohl den Wert des Schlüssels als auch des Werts erhält und Letzteren um eins erhöht zurückgibt. Falls es noch keinen Eintrag für ein Wort gibt, kann man entweder einen Aufruf von `computeIfAbsent()` oder die Methode `putIfAbsent(K,V)` nutzen.

Die Methode `merge()`

Die abschließend vorgestellte Methode `merge(K, V, BiFunction<? super V, ? super V, ? extends V>)` realisiert eine Funktionalität, ähnlich zu `computeIfAbsent()`, die einen existierenden Eintrag mit einer übergebenen Funktion verknüpft: Der neue Wert wird aus dem alten Wert und einer binären Operation ermittelt. Für den Fall, dass es für den Schlüssel noch keinen Wert gibt, wird der an die Methode `merge()` übergebene Startwert vom Typ `V` in der Map gespeichert. Zur Verdeutlichung möchte ich dies für das Beispiel des Wörterzählens wie folgt nutzen:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        wordCounts.merge(word, 1, Integer::sum); // JDK 8: Methodenreferenz
    }

    System.out.println(wordCounts);
}
```

Listing 6.23 *Ausführbar als* **'WORDCOUNTMERGEEXAMPLE'**

Im Listing sehen wir die Methodenreferenz `Integer::sum` (vgl. Abschnitt 4.3), die auf die Methode `sum()` der Klasse `Integer` verweist und eine Addition wie folgender Lambda ausführt: `(int x, int y) -> x + y`.

Fazit

An der schrittweisen Weiterentwicklung des Beispiels erkennen wir sehr schön, dass man sich von der imperativen Programmierung hin zu einem deklarativen Programmierstil bewegt. Die erste Variante hat den Algorithmus mit 15 Zeilen umgesetzt. Zum Schluss benötigt man nur noch fünf Zeilen. Neben der Kürze kommuniziert die obige Lösung vor allem die gewünschte Funktionalität viel besser.

Die zuvor vorgestellten Methoden sind für viele Anwendungsfälle praktisch und erleichtern die tägliche Arbeit. Man kann sich dadurch mehr auf das zu lösende Problem als auf die Details der Zugriffe auf die Map konzentrieren. Insgesamt sinkt durch den deklarativen Ansatz und durch die im Framework implementierten Algorithmen die Wahrscheinlichkeit für Flüchtigkeitsfehler – erschwerend dazu kann beim imperativen Ansatz ein kleiner Fehler viel schneller zu unerwarteten Resultaten führen.

6.1.13 Collection-Factory-Methoden in JDK 9

Das Erzeugen von Collections für ein paar vordefinierte Werte ist in Java mitunter etwas umständlich. Sprachen wie Groovy oder Python bieten dafür eine spezielle Syntax, sogenannte ***Collection-Literale***. Bereits 2009 hat man auch für Java über eine Integration einer einfacheren Schreibweise zur Erzeugung von und zum Zugriff auf Collections nachgedacht. Leider wurde nichts Derartiges realisiert, obwohl es einige vielversprechende Vorschläge gab.

Vorschlag: Collection-Literale und Datenzugriff

Während man auf Arrays indiziert mit eckigen Klammern zugreift, muss man für Listen die Methode `get(int)` und für Maps die Methode `get(K)` nutzen. Wertzuweisungen erfolgen durch Aufruf von `set(int,E)` bzw. `put(K,V)`.

Für indizierte Zugriffe und Wertzuweisungen auf Listen bzw. Zugriffe auf Schlüssel und Werte bei Maps war für Java eine Notation analog zu indizierten Array-Zugriffen vorgesehen, wodurch explizite Methodenaufrufe überflüssig geworden wären:

```
// Collection-Literale leider weder mit JDK 8 noch JDK 9 umgesetzt
newStyleList[0] = "newValue";
final String valueAtPos0 = newStyleList[0]; // => newValue
```

Realisierung mit JDK 9

In Java 9 gibt es leider keine Collection-Literale in der zuvor beschriebenen Form. Stattdessen wurde eine Armada an Factory-Methoden mit den Namen `of()` in die Interfaces `List<E>`, `Set<E>` und `Map<K,V>` integriert, die sich wie folgt zur Erzeugung von Collections nutzen lassen:

```
final List<String> names = List.of("Mike", "Tim", "Tom");
final Set<Integer> primeNumbers = Set.of(2, 3, 5, 7, 11);
final Map<Integer, String> numberMapping = Map.of(5, "five", 6, "six");
```

Schauen wir uns dazu ein Beispiel für jede der Methoden an, das darüber hinaus die zusätzliche Methode `ofEntries()` im Interface `Map<K,V>` und deren Nutzung verdeutlicht:[15]

```
public static void main(final String[] args)
{
    // Collection Factory Methods
    final List<String> names = List.of("MAX", "MORITZ", "MIKE");
    names.forEach(name -> System.out.println(name)); // oder System.out::println

    final Set<Integer> numbers = Set.of(1, 2, 3);
    numbers.forEach(number -> System.out.println(number));
```

[15] Zur besseren Lesbarkeit erfolgt dazu ein statischer Import von `java.util.Map.entry`. Zudem sieht man, dass `ofEntries()` eine klarere Gruppierung von Schlüssel und Wert erlaubt, was vor allem dann hilfreich ist, wenn diese vom selben Typ sind.

```
    final Map<Integer, String> mapping = Map.of(5, "five", 6, "six");
    final Map<Integer, String> mapping2 = Map.ofEntries(entry(5, "five"),
                                                        entry(6, "six"));
    mapping.forEach((key, value) -> System.out.println(key + ":" + value));
    mapping2.forEach((key, value) -> System.out.println(key + ":" + value));
}
```

Listing 6.24 *Ausführbar als* '**COLLECTIONFACTORYMETHODSEXAMPLE**'

Das Programm COLLECTIONFACTORYMETHODSEXAMPLE gibt Folgendes aus:

```
MAX
MORITZ
MIKE
1
2
3
6:six
5:five
6:six
5:five
```

Die Reihenfolge der Elemente bei Sets und Maps ist bei der Nutzung der Collection-Factory-Methoden allerdings nicht definiert und insbesondere wird die Einfügereihenfolge oftmals nicht beibehalten. Auch die Reihenfolge bei einer Iteration ist zufällig, sodass es durchaus etwa zu folgenden Ausgaben kommen kann:

```
MAX
MORITZ
MIKE
3
2
1
5:five
6:six
5:five
6:six
```

Besonderheiten bei der Konstruktion In den jeweiligen Collection-Interfaces sind die Collection-Factory-Methoden explizit für 0 bis 10 Parameter sowie als Vararg-Variante definiert. Allerdings führt diese performanceoptimierte Realisierung zu einer Menge an Methoden. Zudem gibt es für den Fall mit dem Vararg-Parameter noch einen speziellen `switch-case`, der die spezialisierten Methoden aufruft:

```
static <E> List<E> of() {
    return ImmutableCollections.List0.instance();
}

static <E> List<E> of(E e1) {
    return new ImmutableCollections.List1<>(e1);
}

...
```

```
@SafeVarargs
@SuppressWarnings("varargs")
static <E> List<E> of(E... elements) {
    switch (elements.length) {
        case 0:
            return new ImmutableCollections.List0<>();
        case 1:
            return new ImmutableCollections.List1<>(elements[0]);
        case 2:
            return new ImmutableCollections.List2<>(elements[0], elements[1]);
        default:
            return new ImmutableCollections.ListN<>(elements);
    }
}
```

Glücklicherweise sieht man diese Implementierungsdetails nur dann, wenn man in den JDK-Sourcecode schaut. Allerdings scheint diese Unschönheit zumindest in der IDE bei der Auto-Completion durch, die eine Horde gleichnamiger Methoden anbietet.

Space Efficiency Das obige Listing zeigt, dass spezielle innere Klassen aus der Klasse `java.util.ImmutableCollections` erzeugt werden. Neben ihrer Unveränderlichkeit und auf Arrays basierenden Spezialimplementierungen zeichnen sich die durch die Collection-Factory-Methoden erzeugten Collections dadurch aus, dass sie viel weniger Speicherplatz verbrauchen als diejenigen aus dem Collections-Framework.

Betrachten wir ein `java.util.HashSet<String>` mit zwei Elementen bzw. ein mit den neuen Collection-Factory-Methoden erzeugtes `Set<String>`. Die folgende Variante für JDK 8 benötigt über 150 Bytes, wogegen diejenige für JDK 9 lediglich um die 20 Bytes verbraucht:[16]

```
final Set<String> namesSet = new HashSet<>();
names.add("MAX");
names.add("TIM");
final Set<String> unmodifiableNamesJDK8 = Collections.unmodifiableSet(namesSet);

final Set<String> unmodifiableNamesJDK9 = Set.of("MAX", "TIM");
```

Besonderheiten bei Duplikaten

Beim Einsatz der Collection-Factory-Methoden sollte man ein Detail für `Set<E>` und `Map<K,V>` kennen: Bekanntermaßen modelliert ein `Set<E>` das mathematische Konzept einer Menge und enthält somit keine Duplikate. Das gilt auch für die Schlüssel in Maps. Diese Eigenschaft wurde bei den bisherigen Collections automatisch sichergestellt, indem beim Einfügen von Elementen gegebenenfalls Duplikate aussortiert wurden. Das erfordert, dass die steuernden Methoden wie `equals()`, `hashCode()` usw. korrekt implementiert sind (vgl. Abschnitt 6.1.9).

[16] Die Zahlen basieren auf Stuart Marks' Vortrag »New Collections APIs for Java 9«, der online unter `https://www.youtube.com/watch?v=OJrIMv4dAek` verfügbar ist.

Die Collection-Factory-Methoden weisen allerdings eine nicht überraschungsfreie Besonderheit auf: ***Für Sets wird zum Konstruktionszeitpunkt die Duplikatfreiheit geprüft. Ist diese nicht gegeben, so wird eine Exception ausgelöst. Gleiches gilt auch für Duplikate bei den Schlüsseln von Maps.*** Bei Listen findet dagegen – wie erwartet – keine Duplikatsprüfung statt. Schauen wir uns ein Beispiel an:

```
final Set<String> names = Set.of("MAX", "Moritz", "MAX");
```

Bei dieser Variablendefinition kommt es zur Laufzeit zu folgender Exception:

```
java.lang.IllegalArgumentException: duplicate element: MAX
    at java.base/java.util.ImmutableCollections$SetN.<init>(ImmutableCollections.
          java:462)
```

Weil die Collections direkt anhand der übergebenen Werte konstruiert werden, kann man jedoch auch einen Grund für dieses Verhalten finden, nämlich die Vermeidung von Inkonsistenzen durch Flüchtigkeitsfehler in Form einer Mehrfachangabe von Werten.

Hinweis: Spezifische Typen erzeugen

Die durch die inneren Klassen von `ImmutableCollections` erzeugten Collections erfüllen die jeweiligen Interfaces aus dem Collections-Framework. Wenn man jedoch eine Sortierung wie bei `java.util.TreeSet<E>` und `java.util.TreeMap<K,V>` wünscht, wird dies nicht direkt unterstützt.

Fazit

Die Collection-Factory-Methoden können mich nicht vollständig überzeugen. Für die Definition kleinerer Wertbestände gefallen Sie mir eigentlich recht gut, auch wenn sie nicht ganz so elegant in der Schreibweise sind, wie es Collection-Literale wären. Positiv können die Collection-Factory-Methoden ihre enorme Speichereffizienz ins Feld führen. Oftmals ist zudem die Unveränderlichkeit von Vorteil, etwa wenn man diese Collections an andere Programmteile weitergibt.

Als potenziellen Negativpunkt sehe ich, dass die Duplikatbehandlung für `Set<E>` zumindest nicht überraschungsfrei ist – die ausgelöste Exception ist meiner Meinung nach sogar kontraintuitiv. Auch sollte man nicht zu sehr hinter die Kulissen der Collection-Factory-Methoden schauen, denn deren Realisierung als eine große Menge überladener, statischer Methoden in Interfaces ist eher fragwürdig.

Insgesamt lässt sich festhalten, dass die zuvor genannten Negativpunkte für den Einsatz in der Praxis eine untergeordnete Rolle spielen. Als Nutzer fällt vor allem die Vereinfachung bei der Konstruktion von Collections positiv ins Gewicht.

6.1.14 Unveränderliche Kopien von Collections mit Java 10

Bevor wir uns die Neuerung zum Erzeugen unveränderlicher Kopien von Collections in Java 10 anschauen, wollen wir zunächst einen Blick zurück wagen. Gerade zuvor wurden die sogenannten Collection-Factory-Methoden eingeführt, die es erlauben, unveränderliche Collections zu erzeugen. Dazu dienen verschiedene statische `of()`-Methoden sowie für Maps zusätzlich eine `ofEntries()`-Methode.

Einführung in die Thematik

Nachfolgend ist ein Beispiel zum Erzeugen eines unveränderlichen Sets dargestellt:

```
private Set<String> createImmutableSetJdk9Style()
{
    return Set.of("Tim", "Tom", "Jerry");
}
```

Alternativ kann man mit `asList()` basierend auf einer festgelegten Wertefolge eine korrespondierende unveränderliche Liste erzeugen:

```
final List<String> names = Arrays.asList("Tim", "Tom", "Jerry");
```

Jedoch erlaubt diese Form nur keine Modifikationen an der Listenstruktur, einzelne Elemente können dagegen problemlos – wenn auch nicht immer überraschungsfrei – geändert werden:

```
final List<String> names1 = Arrays.asList("Tim", "Tom", "Mike", "XYZ");
names1.set(3,  "PETER");
System.out.println(names1); // [Tim, Tom, Mike, PETER]
```

Neben diesem Fallstrick existieren auch keine korrespondierenden Methoden für Sets und auch nicht für Maps. Zudem war es in keinem Fall möglich, von einem bereits existierenden Container eine unveränderliche Kopie zu erzeugen. Als Abhilfe wurde oftmals folgendes Idiom eines Konstruktoraufrufs mit Übergabe einer Collection genutzt:

```
final List<String> newCopyOfCollection = new ArrayList<>(names);
```

Basierend auf den Eigenschaften einer `ArrayList<E>` ist die so erzeugte Kopie veränderlich – selbst dann, wenn die ursprüngliche Datenstruktur unveränderlich war.

Auch der bis einschließlich Java 8 verfügbare »Umweg« über die Utility-Funktionalität in Form der überladenen Methoden `unmodifiableXyz()` (`Xyz` = `List`, `Set`, `Map`) führt nicht zu einer unveränderlichen Kopie: Es entsteht dadurch lediglich ein Wrapper um die ursprüngliche Datenstruktur. Zudem ist recht viel Schreibaufwand nötig,

Erweiterung in Java 10

Für den Fall, dass man eine unveränderliche Kopie einer Liste benötigt, macht die in Java 10 neu eingeführte Methode `copyOf()` das Leben deutlich einfacher als noch mit Java 9. Das zeigt folgender Aufruf eindrucksvoll:

```
final List<String> newImmutableCopy = List.copyOf(names);
```

Selbstverständlich gibt es Analogien für Sets und Maps. Das sind die Methoden `Set.copyOf(originalSet)` und `Map.copyOf(originalMap)`.

Beispiel

Betrachten wir die neue Kopierfunktionalität für die jeweiligen Datenstrukturen:

```
List<String> names = new ArrayList<>(List.of("Tim", "Tom", "Jerry"));
List<String> copyOfNames = List.copyOf(names);
System.out.println("copyOfList: " + copyOfNames.getClass());

Set<String> colors = new HashSet<>(Set.of("Red", "Green", "Blue"));
Set<String> copyOfColors = Set.copyOf(colors);
System.out.println("copyOfSet:  " + copyOfColors.getClass());

Map<String, Long> personAgeMapping = new HashMap<>(
                  Map.of("Tim", 47L, "Tom", 12L, "Mike", 47L, "Max", 25L));
Map<String, Long> copyOfMap = Map.copyOf(personAgeMapping);
System.out.println("copyOfMap:  " + copyOfMap.getClass());
```

Durch die obigen Anweisungen kommt es zu folgenden Ausgaben, die zeigen, dass hier die mit Java 9 eingeführten Immutable-Collection-Klassen verwendet werden, wie dies auch beim Aufruf der Collection-Factory-Methoden der Fall ist.

```
copyOfList: class java.util.ImmutableCollections$ListN
copyOfSet:  class java.util.ImmutableCollections$SetN
copyOfMap:  class java.util.ImmutableCollections$MapN
```

Wir erhalten hier unveränderliche Kopien. Wird trotzdem eine Modifikation versucht, so wird diese verhindert und eine `UnsupportedOperationException` ausgelöst. Probieren Sie es mal mit folgenden Zeilen aus:

```
final List<String> names1 = Arrays.asList("Tim", "Tom", "Mike", "XYZ");
names1.set(3,  "PETER");
System.out.println(names1); // [Tim, Tom, Mike, PETER]

final List<String> names2 = List.of("Tim", "Tom", "Mike", "XYZ");
// Exception in thread "main" java.lang.UnsupportedOperationException
// names2.set(3,  "PETER");

final List<String> names3 = List.copyOf(names1);
// Exception in thread "main" java.lang.UnsupportedOperationException
// names3.set(3,  "PETER");
```

6.1.15 Entscheidungshilfe zur Wahl von Datenstrukturen

Nachfolgend möchte ich für die beschriebenen Containerklassen eine Entscheidungshilfe ableiten, weil die adäquate Wahl der für ein Problem geeigneten Datenstruktur große Auswirkungen sowohl auf die Lesbarkeit und Verständlichkeit als auch auf die Performance haben kann. Als Faustregel gilt, dass sowohl die `ArrayList<E>` als auch das `HashSet<E>` sowie die `HashMap<K,V>` die erste Wahl sind, unter anderem auch weil sie oftmals gute Performance (vgl. Abschnitt 22.2.1) liefern. Arrays dienen z. B. zur Verwaltung primitiver Typen. Insbesondere bei Mehrdimensionalität stellt ein Array-Zugriff vielfach die natürlichste Zugriffsvariante dar.

Abbildung 6-8 bietet eine Entscheidungshilfe zur Auswahl einer geeigneten Datenstruktur aus dem Collections-Framework und greift Ideen aus dem Buch »Java 2 – Designmuster und Zertifizierungswissen« [16] von Friedrich Esser auf, beschränkt sich dabei aber aus Gründen der Übersichtlichkeit auf die zuvor besprochenen Klassen.

Abbildung 6-8 *Entscheidungshilfe zur Wahl von Datenstrukturen*

Auswirkungen von Fehlverwendungen

Nutzt man beispielsweise ein Array, obwohl Listenfunktionalität benötigt wird, so kommt es durch diese für das Problem unpassend gewählte Datenstruktur zu mehr Sourcecode und mehr Komplexität: Die Listenfunktionalität muss dann jeweils an der einsetzenden Stelle vom Entwickler selbst programmiert werden. Die Wahrscheinlichkeit für Fehler steigt, weil Eigenimplementierungen weniger ausgereift und gut getestet sind als die Klassen des JDKs. Ähnliches gilt, wenn man etwa eine Liste nutzt, obwohl man eine Menge benötigt. Auch in diesem Fall müsste der verwendende Sourcecode jeweils die Mengenfunktionalitäten sicherstellen. Wird dies an nur einer Stelle vergessen oder unterlassen, drohen Inkonsistenzen und Programmfehler.

6.2 Suchen und Sortieren

Nachdem wir bisher hauptsächlich die Verwaltung von Daten in Containern betrachtet haben, wollen wir uns im Folgenden mit den Themen Suchen und Sortieren beschäftigen. Das sind zwei elementare Themen der Informatik im Bereich der Algorithmen und Datenstrukturen. Das Collections-Framework setzt beide um und nimmt einem dadurch viel Arbeit ab. Allerdings ist bis JDK 8 eine wichtige und in der Praxis häufig benötigte Funktionalität nicht enthalten: das Filtern. Das ändert sich mit Java 8. Dessen mächtige Möglichkeiten zur Filterung mit dem Filter-Map-Reduce-Framework werden in Abschnitt 7.2 beschrieben.

Zunächst betrachten wir das Suchen in Abschnitt 6.2.1. Danach behandeln die Abschnitte 6.2.2 sowie 6.2.3 das Sortieren von Arrays und Listen sowie das Sortieren mit Komparatoren. In Abschnitt 6.2.4 gehe ich dann auf Erweiterungen im Interface `Comparator<T>` mit JDK 8 ein.

6.2.1 Suchen

Praktischerweise besitzen alle Containerklassen Methoden, mit denen man nach Elementen suchen kann, und zudem gibt es eine Prüfung, ob Elemente enthalten sind.

Suchen mit `contains()`

Wenn Containerklassen über den allgemeinen Typ `Collection<E>` angesprochen werden, so lässt sich durch Aufruf der Methode `contains(Object)` ermitteln, ob ein gewünschtes Element vorhanden ist. Darüber hinaus kann man mit `containsAll(Collection<?>)` prüfen, ob eine Menge von Elementen enthalten ist. Dabei wird über die Elemente der Collection iteriert, und jedes einzelne wird basierend auf `equals(Object)` auf Gleichheit mit dem übergebenen Element bzw. den übergebenen Elementen geprüft. Für Maps existieren korrespondierende Methoden `containsKey(Object)` und `containsValue(Object)`.

Tipp: Signatur von `contains()`

Eine Anmerkung noch: Wieso verwendet `contains()` nicht den generischen Typ, sondern `Object`? Laut Methodenkontrakt wird das hereingereichte Objekt eines beliebigen Typs mittels `equals(Object)` mit den Elementen der Collection verglichen. Aufgrund der Signatur von `equals(Object)` ist der konkrete Typ nur indirekt durch die Implementierung und deren interne Typprüfung von Bedeutung.

Suchen mit `indexOf()` und `lastIndexOf()`

Für Listen gibt es ergänzend zu der Prüfung auf Existenz mit `contains(Object)` die Methoden `indexOf(Object)` und `lastIndexOf(Object)`, um die Position eines gesuchten Elements zu ermitteln. Die erste Methode beginnt die Suche am Anfang einer

Liste und die zweite beginnt an deren Ende. Auf diese Weise kann, sofern vorhanden, entweder das erste bzw. letzte Vorkommen ermittelt werden. Gleichheit wird wiederum durch `equals(Object)` überprüft.

Suchen mit `binarySearch()`

Neben den gerade genannten Suchmethoden, die iterativ so lange alle Elemente der Datenstruktur betrachten, bis sie fündig geworden sind, wird für die Datenstrukturen Array und `List<E>` im JDK eine effiziente Suche, die sogenannte ***Binärsuche***, angeboten. ***Diese setzt allerdings zwingend sortierte Daten voraus*** – idealerweise durch die Wahl einer entsprechenden Datenstruktur wie beispielsweise eines `TreeSet<E>`s, das die Elemente immer sortiert hält. Wenn man Daten explizit erst sortieren muss, dann ist der Vorteil gegenüber einer linearen Suche gerade bei kleinen Datenbeständen kaum gegeben.

Bei größeren Datenvolumina ist die logarithmische Laufzeit einer Binärsuche allerdings deutlich besser als diejenige einer linearen Suche: Die geringe Laufzeit wird dadurch erreicht, dass der Algorithmus die jeweils zu verarbeitenden Bereiche halbiert und danach im passenden Teilstück weitersucht. Der beschriebene Ablauf bei der Binärsuche wird im JDK durch die überladene Methode `binarySearch()` in den Utility-Klassen `Arrays` bzw. `Collections` realisiert. Abbildung 6-9 stellt den prinzipiellen Ablauf dar, wobei aussortierte Teilstücke grau markiert sind.

Abbildung 6-9 *Schematischer Ablauf bei der Binärsuche*

In der Abbildung zeigt der Pfeil im ersten Schritt zwischen die Elemente – je nach Implementierung von `binarySearch()` wird bei gerader Anzahl das zur Mitte direkt benachbarte obere bzw. untere Element zum Vergleich genutzt.

Analogie: Binärsuche im realen Leben

Für einige eher abstrakte Algorithmen lassen sich schöne Analogien aus dem realen Leben finden. Dies gilt auch für die Binärsuche. Betrachten wir dazu die Suche nach einer Telefonnummer in einem Telefonbuch.

Wollten wir die Telefonnummer von Michael Inden finden, so würden wir das Telefonbuch etwa in der Mitte aufschlagen, da wir wissen, dass die Einträge alphabetisch nach Nachnamen geordnet sind und demnach Inden ziemlich mittig zu finden sein sollte – jedenfalls wenn man in etwa eine Gleichverteilung der Nachnamen und deren Anfangsbuchstaben annimmt. Möglicherweise haben wir nicht exakt getroffen und sind bei Max Mustermann gelandet. Dann gehen wir etwas nach vorne und schlagen dort auf. Wir landen etwa bei Nachnamen mit dem Anfangsbuchstaben H. Dann gehen wir wieder ein Stück nach hinten und landen etwa beim Buchstaben J. Das setzen wir fort, bis wir den Autor gefunden haben. Nach diesem Prinzip arbeitet die Binärsuche und ist damit deutlich effizienter, als die Suche beim Buchstaben A zu starten und sich sukzessive bis I vorzuarbeiten.

Binärsuche in Arrays und Listen Betrachten wir als Beispiel eine Suche in einer `List<Integer>`, die einige Primzahlen in aufsteigender Ordnung speichert. Dort suchen wir nach den Zahlen 7 und 14. Im Anschluss daran kehren wir die Reihenfolge der Elemente um und wiederholen die Suche. Das Ganze wird folgendermaßen realisiert:

```
public static void main(final String[] args)
{
    final List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13);

    System.out.println(primes); // [2, 3, 5, 7, 11, 13]

    // Suche nach 7 und 14
    System.out.println(Collections.binarySearch(primes, 7));  // 3
    System.out.println(Collections.binarySearch(primes, 14)); // -7

    // Anderes Sortierkriterium anwenden, um Versagen der Suche zu provozieren
    Collections.sort(primes, Collections.reverseOrder());
    System.out.println(primes); // [13, 11, 7, 5, 3, 2]

    // Suche nach 7 und 14
    System.out.println(Collections.binarySearch(primes, 7));  // 2
    System.out.println(Collections.binarySearch(primes, 14)); // -7
}
```

Listing 6.25 *Ausführbar als* **'BINARYSEARCHEXAMPLE'**

Führt man das Programm BINARYSEARCHEXAMPLE aus, so wird der Wert 7 wie erwartet an Position drei (0-basiert) gefunden. Der gesuchte Wert 14 ist jedoch nicht gespeichert. Die Methode `binarySearch()` liefert als Ergebnis den Wert -7. Sind die Rückgabewerte der Methode `binarySearch()` größer oder gleich 0, so entspricht dies dem 0-basierten Index, an dem der gesuchte Wert gefunden wurde. Ein negativer Wert bedeutet, dass das gesuchte Element nicht in der Datenstruktur gespeichert ist. Über den Rückgabewert wird die Position beschrieben, an der der gesuchte Wert in den Container eingefügt werden kann, ohne die Sortierung zu zerstören. Demnach codiert der Rückgabewert die Position wie folgt: $Position = -Rückgabewert - 1$. Die Zahl 14 könnte demnach an Position 6 eingefügt werden, also hinter der 13.

Bis jetzt scheint alles recht einfach zu sein. Um auf Fallstricke hinzuweisen, wird für das Beispiel anschließend eine absteigende Sortierung durch einen Aufruf der statischen Methode `Collections.sort(List<T>, Comparator<? super T>)` hergestellt. Dazu nutzen wir einen Komparator, den man durch Aufruf der statischen Methode `reverseOrder()` erzeugt. Nun wird die Suche auf der absteigend sortierten Liste wiederholt. Der Wert 7 wird dann an Position 2 gefunden, was jedoch ein Zufallstreffer ist, der durch die Teilung der Eingabewerte und die Mittenposition des Werts 7 entsteht.[17] Auch der Rückgabewert -7 für die Suche nach dem Wert 14 verwundert, denn in der umsortierten Liste wäre der Wert -1 die richtige Rückgabe für Position 0 gewesen, an der 14 eingefügt werden müsste (siehe nachfolgenden Kasten »Fallstrick«).

Fallstrick: Unterschiedliche Reihenfolgen bei Sortierung und Suche

Wie im Beispiel gesehen, ist es elementar wichtig, dass ***sowohl die zur Sortierung genutzte als auch die bei der Suche verwendete Reihenfolge identisch sind***.[a] Ansonsten ist das Ergebnis einer solchen Suche nicht definiert, liefert jedoch häufig den Wert -1. Da die Binärsuche ohne Angabe eines Sortierkriteriums von einer Sortierung gemäß der natürlichen Ordnung »ausgeht«, kommt es im obigen Beispiel durch die abweichende Sortierung aber auch zu der falschen Angabe von -7 für eine Suche der Zahl 14 in der absteigend sortierten Liste von Primzahlen.

[a] Dies ist eine »beliebte« Fangfrage bei OCPJP-Prüfungen.

Binärsuche in Sets und Maps? Intuitiv fragt man sich: Wie sucht man effizient in Sets und Maps? Mit dem bisher erlangten Wissen können wir diese Frage leicht beantworten: Die hashbasierten Container bieten aufgrund der Speicherung in einer Hashtabelle bereits einen extrem performanten Zugriff (sofern die Hashfunktion gut und nicht zu aufwendig zu berechnen ist), sodass eine zusätzliche *eigenständige* Realisierung einer derartigen Suche nicht sinnvoll ist.[18] Das Gleiche gilt für die sortierten Container `TreeSet<E>` und `TreeMap<K,V>`. Wie in Abschnitt 6.1.8 beschrieben, verwenden diese Datenstrukturen einen balancierten Baum, in dem die Suche nach dem gleichen Prinzip wie bei der Binärsuche erfolgt.

6.2.2 Sortieren von Arrays und Listen

Im Collections-Framework kann man neben verschiedenen Realisierungen der automatisch sortierenden Container `SortedSet<E>` und `SortedMap<K,V>` bei Bedarf auch Arrays und Listen durch einen Aufruf der jeweiligen Methode `sort()` aus den Utility-Klassen `Arrays` bzw. `Collections` sortieren. Seit Java 8 bietet das Interface `List<E>` vereinfachend eine Methode `sort()`.

[17] Erweitert man die Eingabemenge der Primzahlen um die Werte 17 und 19, so wird als Folge die Zahl 7 nicht mehr gefunden und die Binärsuche liefert die Rückgabe -1.

[18] Zudem ist eine Realisierung aufgrund der unsortierten Speicherung in einer Hashtabelle nicht möglich.

Bevor wir im nächsten Abschnitt Komparatoren, die die Reihenfolge der Elemente steuern, genauer betrachten, wollen wir anhand der manuellen Sortierung von Listen den Einsatz von Komparatoren rekapitulieren und danach auch einen Blick auf die Sortierung von Arrays werfen.

Warnhinweis: `null`-Prüfungen in Komparatoren

Um die nachfolgenden Ausführungen zu den Komparatoren nicht mit `null`-Prüfungen unübersichtlich zu machen, werde ich darauf (weitestgehend) verzichten. In der Praxis ist es jedoch oftmals sinnvoll, diese Prüfungen vorzunehmen. Noch besser ist es, die Eingabedaten `null`-frei zu halten, sofern dies möglich ist.

Beispiel: Sortieren einer Liste vor JDK 8

Vor JDK 8 können wir zum Sortieren von Listen die Methode `Collections.sort()` nutzen und die Sortierreihenfolge mithilfe der Interfaces `Comparable<T>` und `Comparator<T>` festlegen:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Tim", "Stefan");

   // Comparable<String>-basierte Sortierung
   Collections.sort(names);
   System.out.println(names);

   final Comparator<String> byLength = new Comparator<String>()
   {
      @Override
      public int compare(final String str1, final String str2)
      {
         return Integer.compare(str1.length(), str2.length());
      }
   };

   // Comparator<String>-bestimmte Sortierung
   Collections.sort(names, byLength);
   System.out.println(names);
}
```

Listing 6.26 *Ausführbar als* **'LISTSORTEXAMPLE'**

Führen wir das Programm LISTSORTEXAMPLE aus, so zeigt die Ausgabe zunächst eine alphabetische Sortierung und dann eine nach der Länge der Strings:

```
[Andy, Michael, Stefan, Tim]
[Tim, Andy, Stefan, Michael]
```

Vereinfachungen mit JDK 8 JDK 8 bietet Lambdas, Methodenreferenzen und Defaultmethoden (vgl. Kapitel 4), die bei Sortierungen nützlich sind. Auch das Interface `Comparator<T>` wurde erweitert, wodurch sich Komparatoren mit sehr wenig Aufwand erstellen lassen. Schließlich offeriert das Interface `List<E>` eine Methode `sort()`. All dies wird nachfolgend eingesetzt, um obiges Beispiel zu modernisieren:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Tim", "Stefan");

   // Comparable<String>-basierte Sortierung
   names.sort(Comparator.naturalOrder());
   System.out.println(names);

   // Comparator<String>-bestimmte Sortierung mit Lambda
   final Comparator<String> byLength =
         (str1, str2) -> Integer.compare(str1.length(), str2.length());

   names.sort(byLength);
   System.out.println(names);
}
```

Beispiel: Sortieren eines Arrays

Analog zu dem zuvor gezeigten Sortieren von Listen kann man Arrays sortieren. Dafür nutzt man die Methode `Arrays.sort()`. Im folgenden Beispiel zeige ich sowohl die auf `Comparable<T>` basierende natürliche Ordnung (vgl. Abschnitt 6.1.8) als auch die Variante mit einem Komparator, der in Form eines Lambdas definiert ist:

```
public static void main(final String[] args)
{
    final String[] names = { "Andy", "Michael", "Tim", "Stefan" };

    // Comparable-basierte Sortierung
    Arrays.sort(names);
    System.out.println(Arrays.toString(names));

    // Comparator-bestimmte Sortierung mit Lambda
    final Comparator<String> byLength =
          (str1, str2) -> Integer.compare(str1.length(), str2.length());

    Arrays.sort(names, byLength);
    System.out.println(Arrays.toString(names));
}
```

Listing 6.27 *Ausführbar als* **'ARRAYSSORTJDK8EXAMPLE'**

Führen wir das Programm ARRAYSSORTJDK8EXAMPLE aus, so zeigt die Ausgabe zunächst eine alphabetische Sortierung und dann eine nach der Länge der Strings:

```
[Andy, Michael, Stefan, Tim]
[Tim, Andy, Stefan, Michael]
```

6.2.3 Sortieren mit Komparatoren

Im Folgenden wird der Einsatz von Komparatoren etwas genauer betrachtet, um verschiedene Sortierungen und Kombinationen davon zu realisieren. In JDK 8 wurden Erweiterungen für Komparatoren eingeführt, die die Arbeit enorm erleichtern. Diese Neuerungen schauen wir uns in Abschnitt 6.2.4 an.

Für die kommenden Beispiele stellen wir uns vor, wir hätten eine Liste mit `Person`-Objekten, die nach verschiedenen Kriterien sortiert werden sollen, etwa nach Name, Wohnort oder Alter. ***Um die Beispiele einfach halten zu können, gehe ich davon aus, dass die zu sortierenden Collections keine `null`-Werte enthalten und man daher auf `null`-Prüfungen verzichten kann***. Wie man elegant mit `null`-Werten umgehen kann, stelle ich ebenfalls in Abschnitt 6.2.4 dar.

Definition eines Komparators (nach Name)

Zunächst wollen wir Personen nach Namen sortieren. Dazu implementieren wir eine Klasse `PersonNameComparator`, die zur Realisierung die Methode `compareTo(String)` der Klasse `String` verwendet:

```
public final class PersonNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        return person1.getName().compareTo(person2.getName());
    }
}
```

Das hier verwendete Prinzip ist für Komparatoren im Grunde immer wieder das gleiche, und der konkrete Ablauf variiert oftmals lediglich in der Extraktion der Attribute.

Definition eines Komparators (nach Alter)

Schnell kommt der Wunsch auf, Personen auch nach deren Alter sortieren zu können. Für primitive Datentypen existieren keine `compareTo()`-Methoden. Häufig sieht man dann folgende Art der Realisierung des Vergleichs von Hand:

```
public final class PersonAgeComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        if (person1.getAge() < person2.getAge())
        {
            return -1;
        }
        if (person1.getAge() > person2.getAge())
        {
            return 1;
        }
        return 0;
    }
}
```

Diese Implementierung löst das Problem, erfordert allerdings einige Zeilen Sourcecode. Zum Teil sieht man auch eine Subtraktion der zu vergleichenden Werte:

```
return person1.getAge() - person2.getAge();
```

Diese Form der Berechnung scheint zunächst kürzer und einfacher zu sein. ***Sie ist jedoch zu vermeiden, da sie nicht das zu lösende Problem widerspiegelt***. Darüber hinaus kann diese Art des Vergleichs zu Problemen durch Wertebereichsverletzungen führen, wenn die Differenz zweier Werte aus dem Wertebereich des verwendeten Datentyps `int` herausfällt. Als sichere und besser lesbar und verständliche Alternative ist der Einsatz der Methode `Integer.compare(int, int)`, die einen Rückgabewert kompatibel zu dem von Komparatoren liefert, empfehlenswert:

```
return Integer.compare(person1.getAge(), person2.getAge());
```

Umkehren der Sortierreihenfolge per Komparator

Für Tabellen ist es ein üblicher Anwendungsfall, nach beliebigen Kriterien (Spalten) sortieren zu können. Häufig möchte man dabei zudem die Sortierreihenfolge zwischen aufsteigend und absteigend wechseln können.

Eine Realisierungsidee für den Wechsel der Sortierreihenfolge ist, den Rückgabewert eines Komparators zu negieren. Das funktioniert in der Regel.[19] Einfacher und semantisch klarer als durch Negation lässt sich die Umkehrung der Reihenfolge durch die Bibliotheksklasse `ReverseComparator` und die zwei überladenen Hilfsmethoden `reverseOrder()` ausdrücken. Eine davon basiert auf dem Interface `Comparable<T>` und erlaubt, die natürliche Sortierung umzukehren. Die zweite Version ist für das Interface `Comparator<T>` ausgelegt.

Kombination von Sortierkriterien

Sind Sortierungen nach unterschiedlichen Kriterien möglich, so ist der Wunsch nach einer Kombination verschiedener Sortierungen eine naheliegende Folge, etwa eine Sortierung erst nach Alter und dann nach Name.

Beim Sortieren anhand verschiedener Kriterien gibt es vorrangige und nachrangige Sortierkriterien. Nur wenn das erste (vorrangige) Kriterium Gleichheit ergibt, wird anhand des nächsten (nachrangigen) Kriteriums weiter sortiert. Im Falle der Sortierung zuerst nach Alter und dann nach Name heißt dies, dass der Name nur dann zum Tragen kommt, wenn das Alter gleich ist. Verallgemeinert gilt, dass bei Gleichheit eines Sortierkriteriums so lange das jeweils folgende Kriterium betrachtet werden muss, bis entweder alle Sortierkriterien ausgewertet sind oder vorher eine Ungleichheit fest-

[19] Die Ausnahme bildet der Wert `Integer.MIN_VALUE`. Für diesen gilt die Anomalie `-Integer.MIN_VALUE == Integer.MIN_VALUE` (vgl. Abschnitt 5.2).

gestellt wird. Betrachten wir zur Verdeutlichung eine Realisierung der Kombination der Sortierungen nach Alter und Name:

```
public final class PersonAgeAndNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        int ret = Integer.compare(person1.getAge(), person2.getAge());

        if (ret == 0)
        {
            ret = person1.getName().compareTo(person2.getName());
        }

        return ret;
    }
}
```

Häufig sind Anforderungen nicht präzise formuliert. Nehmen wir an, es bestände zusätzlich der Wunsch, auch in der anderen Kombination, d. h. zuerst nach Name und dann nach Alter, sortieren zu können. Die Realisierung dieser Funktionalität ist einfach:

```
public final class PersonNameAndAgeComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        int ret = person1.getName().compareTo(person2.getName());

        if (ret == 0)
        {
            ret = Integer.compare(person1.getAge(), person2.getAge());
        }

        return ret;
    }
}
```

Schnell erkennt man, dass die hier eingesetzten Vergleichsoperationen denen der beiden eingangs vorgestellten Komparatoren sehr ähneln. Diese Duplikation ist hier zwar nicht ausgeprägt, aber trotzdem nicht optimal, wenn man qualitativ hochwertigen Sourcecode schreiben möchte. Was kann man also verbessern?

Wie man das mit Java 7 bewerkstelligen kann, beschreibe ich in den älteren Auflagen dieses Buchs. Hier wollen wir uns zur Lösung aber mit modernem Java und seinen vielfältigen Möglichkeiten im Bereich der Komparatoren beschäftigen.

6.2.4 Erweiterungen im Interface `Comparator` mit JDK 8

Bekanntermaßen entsteht bei der Realisierung eines `Comparator<T>` in Form einer Klasse schnell einiges an Boilerplate-Code. Zudem möchte man oftmals nicht den Typ `T` selbst, sondern einige seiner Attribute vergleichen. Denken Sie beispielsweise an Komparatoren für eine Liste von Personen. Naheliegend sind die Sortierungen nach Vor- bzw. Nachnamen. Diese kann man durch einen `Comparator<Person>` durchführen, dessen `compareTo(Person, Person)`-Methode aus den zwei übergebenen `Person`-

Objekten den jeweiligen Namensbestandteil extrahiert und dann vergleicht. Seit JDK 8 lässt sich dies hier exemplarisch für Nachnamen mithilfe eines Lambdas prägnant wie folgt schreiben:

```
final Comparator<Person> compareByName = (person1, person2) ->
{
    return person1.getName().compareTo(person2.getName());
};
```

Erweiterungen in JDK 8

Weil derartige Vergleiche ein recht häufiger Anwendungsfall sind, wurde das Interface `Comparator<T>` mit JDK 8 um einige nützliche Methoden ergänzt, die zunächst aufgezählt und danach anhand kleiner Beispiele genauer vorgestellt werden:

- `comparing()` – Definiert einen Komparator basierend auf der Extraktion eines Werts eines Attributs, das sich mithilfe des Interface `Comparable<T>` vergleichen lässt.
- `comparingInt()/-Long()/-Double()` – Definiert einen `Comparator<T>`, wobei der `int-/long-/double`-Wert eines speziellen Attributs des Typs `T` zum Vergleich genutzt wird.
- `thenComparing()` und `thenComparingInt()/-Long()/-Double()` – Hiermit wird die Hintereinanderschaltung von Komparatoren ermöglicht.
- `naturalOrder()`, `reverseOrder()` und `reversed()` – Diese Methoden stellen Komparatoren gemäß der natürlichen, der dazu entgegengesetzten sowie einer zu einem bestehenden Komparator inversen Sortierung bereit.

`comparing()` – auf `Comparable<T>` basierende Komparatoren

Mithilfe der statischen Methode[20] `Comparator.comparing(Function<? super T,? extends U>)` wird ein Komparator erzeugt. Dazu wird als Parameter ein sogenannter Key-Extractor übergeben, der bestimmt, wie das Attribut aus den zu sortierenden Instanzen, im Beispiel den `Person`-Objekten, ermittelt wird. Der Gewinn dabei ist, dass der fixe Ablauf des Vergleichs basierend auf `Comparable<T>` nicht selbst realisiert werden muss, sondern nur der variable Anteil (die Extraktion) einer klassischen Lösung als Parameter übergeben wird. Nachfolgendes Beispiel zeigt, wie man einen `Comparator<Person>`, der den Namensbestandteil extrahiert und vergleicht, durch Aufruf von `comparing()` erzeugen kann:

```
// Varianten mit Comparator.comparing()
Comparator<Person> byName1 = Comparator.comparing(person -> person.getName());
Comparator<Person> byName2 = Comparator.comparing(Person::getName);
```

[20] Hier sieht man ein nützliches Beispiel für eine statische Methode in einem Interface. Es wäre aber ebenso gut möglich gewesen, dies durch eine separate Utility-Klasse bereitzustellen.

Hinweis: Internationalisierung und Umlaute

Der zuvor dargestellte Vergleich von Namen basiert auf `compareTo(String)` und vergleicht zwei Strings textuell (basierend auf deren Zeichencode), ohne auf Umlaute und regionale Besonderheiten Rücksicht zu nehmen. Als Abhilfe lässt sich jedoch ein sogenannter Collator vom Typ `java.text.Collator` nutzen. Das ist eine Spezialisierung des Typs `Comparator<T>`, die sprachspezifische Eigenheiten beim Vergleich beachtet, z. B. die Umlaute ä, ö, ü korrekt einsortiert. Weitere Details finden Sie in Abschnitt 13.1.8.

Verarbeitung primitiver Typen

Im Interface `Comparator<T>` finden sich spezialisierte Varianten der `comparing()`-Methode für die primitiven Typen `int`, `long` und `double`. Damit kann man einen Komparator für `Person`-Objekte und deren Attribut `age` als `int`-Wert mithilfe von `comparingInt()` folgendermaßen konstruieren:

```
Comparator<Person> byAge = Comparator.comparingInt(Person::getAge);
```

`thenComparing()` – Hintereinanderschaltung von Komparatoren

Mitunter ist eine Sortierung nach nur einem Kriterium nicht ausreichend. Für Personen gilt im Speziellen, dass diese durchaus den gleichen Nachnamen besitzen können. Somit wäre ein zweites Sortierkriterium wünschenswert, um bei Menschen mit gleichen Nachnamen eine eindeutige Reihenfolge zu erzielen. Dazu werden wir mehrere Komparatoren hintereinander ausführen. Als Basisbausteine definieren wir zunächst verschiedene Komparatoren für einzelne Attribute und kombinieren diese dann durch Aufrufe der Methode `thenComparing(Comparator<? super T>)`:

```
// Komparatoren für ein spezielles Attribut
Comparator<Person> byFirstname = Comparator.comparing(Person::getFirstname);
Comparator<Person> byName = Comparator.comparing(Person::getName);
Comparator<Person> byAge = Comparator.comparing(Person::getAge);

// Kombination von Komparatoren
Comparator<Person> byNameAndFirstname = byName.
                                        thenComparing(byFirstname);
Comparator<Person> byNameAndAge = byName.thenComparing(byAge);
```

Im Listing ist gezeigt, wie einfach sich Komparatoren zu komplexeren Komparatoren kombinieren lassen. Allerdings erfolgt im Beispiel für die Alterswerte jedoch ständig ein Auto-Boxing aus einem `int` in ein `Integer`-Objekt, da `thenComparing()` einen auf `Comparable<Integer>` basierenden Komparator erzeugt, `getAge()` aber den Rückgabetyp `int` besitzt. Insbesondere bei der Verarbeitung sehr großer Datenmengen oder auf kritischen Pfaden kann der Auto-Boxing-Automatismus unerwünscht sein

und sich leicht ungünstig auf die Performance auswirken. Als Abhilfe haben wir gerade schon die auf primitive Typen spezialisierten Varianten am Beispiel `int` kennengelernt.

Spezielle Ordnungen

Manchmal besteht das Bedürfnis, die Sortierreihenfolge zu beeinflussen, z. B. um die Reihenfolge der Elemente umzudrehen. Das gilt etwa für das Sortieren von Tabellenspalten. Das Interface `Comparator<T>` bietet seit JDK 8 die Methode `reversed()` an, die hier genutzt wird, um eine absteigende Sortierung nach Namen zu definieren:

```
final Comparator<Person> byName = Comparator.comparing(Person::getName);
final Comparator<Person> byNameDescending = byName.reversed();
```

Zum Teil möchte man eine natürliche Ordnung mit einem Komparator abbilden. Dazu dient die Methode `naturalOrder()`, die für Strings eine alphabetische und für Zahlen eine aufsteigende Sortierung liefert. Die entgegengesetzte Sortierung erhält man mit der Methode `reverseOrder()`. Wenn man diese umkehrt, erhält man wieder die natürliche Ordnung. Im Listing sind die entsprechenden Methodenaufrufe gezeigt und die resultierende Sortierung durch fett geschriebene Kommentare angedeutet:

```
public static void main(final String[] args)
{
    final Integer[] primes = { 1, 7, 3, 13, 11, 5, 17, 19 };

    // aufsteigend
    final Comparator<Integer> naturalOrder = Comparator.naturalOrder();
    // absteigend
    final Comparator<Integer> reverseOrder = Comparator.reverseOrder();
    // aufsteigend
    final Comparator<Integer> naturalOrderAgain = reverseOrder.reversed();

    sortAndPrint("naturalOrder     ", primes, naturalOrder);
    sortAndPrint("reverseOrder     ", primes, reverseOrder);
    sortAndPrint("naturalOrderAgain", primes, naturalOrderAgain);
}

private static void sortAndPrint(final String name, final Integer[] primes,
                                 final Comparator<Integer> sortOrder)
{
    Arrays.sort(primes, sortOrder);
    System.out.println(name + ": " + Arrays.toString(primes));
}
```

Listing 6.28 *Ausführbar als* **'NATURALORDEREXAMPLE'**

Das obige Programm NATURALORDEREXAMPLE produziert folgende Ausgaben:

```
naturalOrder     : [1, 3, 5, 7, 11, 13, 17, 19]
reverseOrder     : [19, 17, 13, 11, 7, 5, 3, 1]
naturalOrderAgain: [1, 3, 5, 7, 11, 13, 17, 19]
```

Behandlung von `null`-Werten

Ab und zu sieht man sich der Herausforderung gegenüber, dass die zu sortierenden Datensätze potenziell auch `null`-Werte enthalten. Die üblicherweise genutzten Komparatoren lösen dann ohne spezielle Behandlung – konform zum Kontrakt – eine `NullPointerException` aus. Für einige Anwendungsfälle ist es jedoch erforderlich, `null`-Werte am Anfang oder am Ende einzusortieren. Dazu gibt es die beiden Methoden `nullsFirst(Comparator<? super T>)` und `nullsLast(Comparator<? super T>)`, die wir nachfolgend verwenden, um eine mit `null`-Werten durchsetzte Liste mit Namen zu sortieren:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("A", null, "B", "C", null, "D");

    // Null-sichere Komparatoren zur Dekoration bestehender Komparatoren
    final Comparator<String> naturalOrder = Comparator.naturalOrder();
    final Comparator<String> nullsFirst = Comparator.nullsFirst(naturalOrder);
    final Comparator<String> nullsLast = Comparator.nullsLast(naturalOrder);

    names.sort(nullsFirst);
    System.out.println("nullsFirst: " + names);
    names.sort(nullsLast);
    System.out.println("nullsLast: " + names);
}
```

Listing 6.29 *Ausführbar als* **'NULLSAFECOMPARATOREXAMPLE'**

Führt man das Programm NULLSAFECOMPARATOREXAMPLE aus, kommt es zu folgenden Ausgaben, die die beschriebene Arbeitsweise zeigen:

```
nullsFirst: [null, null, A, B, C, D]
nullsLast: [A, B, C, D, null, null]
```

Oftmals sind in einer Liste komplexere Objekte gespeichert, und die Sortierung basiert auf einem oder mehreren Attributen. Wenn man dafür einen `null`-sicheren Komparator definieren möchte, muss man achtsam sein: Für die Klasse `Person` und ein über die Methode `getFavoriteColor()` ermitteltes optionales Attribut vom Typ `String` müssen die Komparatoren wie folgt zusammengebaut werden:

```
Comparator<Person> byFavoriteColor = Comparator.comparing(
                                     Person::getFavoriteColor,
                                     Comparator.nullsFirst(String::compareTo));
```

Hier kommt eine spezielle Variante der Methode `comparing()` zum Einsatz, bei der als erster Parameter der bekannte Key-Extractor übergeben wird. Der zweite Parameter ist ein Komparator, allerdings einer, der für den Typ des zu extrahierenden Attributs spezialisiert ist, hier also für den Typ `String`.

6.3 Utility-Klassen und Hilfsmethoden

Das Collections-Framework bietet mit den Klassen `Collections` und `Arrays` zwei mächtige Utility-Klassen an, die diverse Algorithmen und Erweiterungen bereitstellen, von denen einige wichtige im Folgenden kurz vorgestellt werden.

6.3.1 Nützliche Hilfsmethoden

In diesem Abschnitt wollen wir einen Blick auf verschiedene Methoden rund um die Erzeugung von Collections werfen.

Listen mit Inhalt befüllen und `Arrays.asList()`

Manchmal soll eine Liste aus einer Menge an vordefinierten Werten erstellt werden. Eine zur Initialisierung gebräuchliche Realisierung ist folgende:

```
final List<String> names = new ArrayList<>();
names.add("Max");
names.add("Michael");
names.add("Carsten");
```

Diese Schreibweise ist recht geschwätzig, da der Name der Collection-Variable immer wieder beim Hinzufügen angegeben werden muss. Etwas eleganter kann man das Ganze schreiben, wenn man zwei Java-Techniken kombiniert, nämlich die Definition einer anonymen Klasse und einen Initializer-Block:

```
final List<String> names = new ArrayList<String>()
{{
    add("Max");
    add("Michael");
    add("Carsten");
}};
```

Damit erspart man sich, den Namen der Collection ständig wiederholen zu müssen. Noch besser lesbar kann man diese Befüllung mit Daten durch Aufruf der statischen Methode `Arrays.asList(T...)` folgendermaßen schreiben:

```
final List<String> names = Arrays.asList("Max", "Michael", "Carsten");
```

Hier nutzt man, dass Java seit Version 5 das Sprachfeature Varargs bietet, wodurch die kommaseparierte Angabe beliebig vieler Werte als Parameter möglich wird. Allerdings muss man einige Besonderheiten beachten, auf die ich nun eingehe.

Alternative mit JDK 9 Seit Java 9 bieten sich auch die in Abschnitt 6.1.13 beschriebenen Collection-Factory-Methoden an:

```
final List<String> names = List.of("Max", "Michael", "Carsten");
```

Hinweis: Syntaktische Besonderheiten im Beispiel

Zur Konstruktion einer Liste mit einer Menge an vordefinierten Werten haben wir eben verschiedene Varianten gesehen. Auf die zweite möchte ich noch einmal explizit eingehen:

```
// Zwei Besonderheiten
final List<String> names = new ArrayList<>()
{{ // #1
    add("Max");      // statt names.add("Max"); #2
    add("Michael");
    add("Carsten");
}}; // #1
```

Dieses kurze Beispiel enthält tatsächlich zwei syntaktische Besonderheiten, über die Sie sich eventuell schon beim Betrachten gewundert haben.

1. Durch die spezielle Doppel-Klammer-Syntax mit `{{ }}` entsteht durch die äußere Klammerung eine anonyme innere Klasse mit dem Basistyp `ArrayList<E>`.
2. Das innere Paar `{}` stellt einen Instance Initializer dar, der während der Konstruktion ausgeführt wird. Zudem können die Aufrufe von `add(E)` ohne den Variablennamen der Collection erfolgen.

Besonderheiten von `Arrays.asList()` Die von der Methode `Arrays.asList(T...)` erzeugte Liste weist im Gegensatz zu einer gewöhnlichen `ArrayList<E>` einige Unterschiede auf. Es wird nämlich eine unveränderliche Liste erzeugt, wodurch Änderungen an der strukturellen Zusammensetzung (die Änderungen der Länge bewirken), etwa über die Methoden `add(E)` bzw. `remove(Object)`, zu einer `UnsupportedOperationException` führen:

```
public static void main(final String[] args)
{
    final String[] valuesArray = { "Value 1", "Value 2", "Value 3" };

    // Änderungen an der Liste (Inhalt, Zusammensetzung)
    final List<String> valuesAsList = Arrays.asList(valuesArray);
    valuesAsList.set(1, "Value 7"); // Inhalt ändern
    // valuesAsList.add("Value 4"); // UnsupportedOperationException

    System.out.println("valuesArray:  " + Arrays.toString(valuesArray));
    System.out.println("valuesAsList: " + valuesAsList);

    // Änderungen am Inhalt des Arrays
    valuesArray[0] = "Michael changed";
    valuesArray[1] = "Value 1 & 2";

    System.out.println("valuesArray:  " + Arrays.toString(valuesArray));
    System.out.println("valuesAsList: " + valuesAsList);
}
```

Listing 6.30 *Ausführbar als* '**ARRAYSASLISTEXAMPLE**'

Das Programm ARRAYSASLISTEXAMPLE produziert folgende Ausgaben:

```
valuesArray:  [Value 1, Value 7, Value 3]
valuesAsList: [Value 1, Value 7, Value 3]
valuesArray:  [Michael changed, Value 1 & 2, Value 3]
valuesAsList: [Michael changed, Value 1 & 2, Value 3]
```

Als Besonderheit sehen wir, dass Werte inhaltlich geändert werden können (z. B. durch Aufrufe von `set()`) und sich dies jeweils in der anderen Datenstruktur widerspiegelt.

Erweiterungen in der Klasse `Arrays` in JDK 9

Die Utility-Klasse `java.util.Arrays` wurde in JDK 9 erweitert. Das betrifft vor allem Funktionalitäten die auch eine Bereichseinschränkung ermöglichen. Dabei werden folgende Signaturen geboten, bei denen `XYZ` jeweils für verschiedene Typen stehen, etwa `Object`, `int`, `long`, `double` usw., die Methoden also mehrfach überladen sind:

- `boolean equals(XYZ[] a, int aFrom, int aTo, XYZ[] b, int bFrom, int bTo)` – Untersucht die beiden Arrays `a` und `b` jeweils elementweise auf Gleichheit, berücksichtigt dabei die jeweiligen Grenzen.
- `int mismatch(XYZ[] a, XYZ[] b)`
  `int mismatch(XYZ[] a, int aFrom, int aTo, XYZ[] b, int bFrom, int bTo)` – Prüft, ob die zwei Arrays `a` und `b` sich unterscheiden, und liefert die Position der ersten Differenz. Durch die Angabe von Indizes können auch Teilbereiche von Arrays untersucht werden. Wird kein Unterschied gefunden, so ist die Rückgabe `-1`.
- `int compare(XYZ[] a, XYZ[] b)`
  `int compare(XYZ[] a, int aFrom, int aTo, XYZ[] b, int bFrom, int bTo)` – Hierbei werden die beiden Arrays `a` und `b` jeweils elementweise verglichen und dabei die natürliche Ordnung beachtet. Der Rückgabewert folgt dem eines Komparators: Bei Gleichheit 0, ist Element `a[i]` kleiner als Element `b[i]`, dann wird eine negative Zahl und ansonsten ein positiver Wert zurückgeliefert. Die verglichenen Werte stammen aus den jeweiligen angegebenen Grenzen.

Erweiterungen bei `equals()` Die Methode `Arrays.equals()` existiert schon lange im JDK, aber man konnte den Vergleich leider nicht auf spezielle Bereiche einschränken, was mit JDK 9 möglich wird.

Schauen wir uns dies für die Texte `ABC-DEF-GHI` und `DEF` an. Offensichtlich stimmen diese nicht über ein. Wenn man jedoch nur den mittleren Teil zwischen den Minuszeichen betrachtet, so sollte dieser Teilbereich gleich sein:

```
final byte[] string1 = "ABC-DEF-GHI".getBytes();
final byte[] string2 = "DEF".getBytes();

System.out.println(Arrays.equals(string1, string2));
System.out.println(Arrays.equals(string1, 4, 7, string2, 0, 3));
```

Erwartungskonform erhält man folgende Ausgaben:

```
false
true
```

Neuerung `mismatch()` Die Methode `Arrays.mismatch()` prüft auf Abweichungen im Array – bei Bedarf auf spezielle Bereiche eingeschränkt.

Beginnen wir mit einem Vergleich zweier `int[]`. Danach schauen wir uns an, wie wir nicht nur eine Abweichung, sondern auch darauffolgende ermitteln können. Dazu nutzen wir einen Text, der jeweils eine Abweichung in der Groß- und Kleinschreibung an spezifischen Positionen aufweist:

```
final int[] values1 = { 1, 2, 3, 4, 5, 13 };
final int[] values2 = { 1, 2, 3, 4, 5, 6, 7 };
System.out.println("Array-Mismatch: " + Arrays. mismatch(values1, values2));

//                      01234012345678901234
final byte[] string1 = "ABC-DEF-GHIJKLMNOPQT".getBytes();
final byte[] string2 = "ABC-dEF-GHIJKLMNOPqT".getBytes();

System.out.println("First Diff: " +
                   Arrays.mismatch(string1, string2));
System.out.println("Next Diff:  " +
                   Arrays.mismatch(string1, 5, 20, string2, 5, 20));
```

Erwartungskonform weicht das erste Array an Position 5 ab und im komplexeren Fall ermitteln wir die Positionen 4 und 13 als Abweichungen:

```
Array-Mismatch: 5
First Diff: 4
Next Diff:  13
```

Neuerung `compare()` Die Methode `Arrays.compare()` vergleicht die Werte zweier Arrays gemäß einem `Comparator<T>`. Auch diesen Vergleich kann man auf spezielle Bereiche einschränken.

Zur Demonstration nutze ich wieder die Texte `ABC-DEF-GHI` und `DEF`. Auch hier gilt wieder, dass diese offensichtlich nicht übereinstimmen, allerdings sollte bei Einschränkung auf den mittleren Teil zwischen den Minuszeichen Gleichheit festgestellt werden. Werden nur die letzten drei Buchstaben des ersten Texts mit dem zweiten Text verglichen, so sollten diese als größer betrachtet werden:

```
final byte[] string1 = "ABC-DEF-GHI".getBytes();
final byte[] string2 = "DEF".getBytes();

System.out.println("compare(all):      " +
                   Arrays.compare(string1, string2));
System.out.println("compare(DEF, DEF): " +
                   Arrays.compare(string1, 4, 7, string2, 0, 3));
System.out.println("compare(GHI, DEF): " +
                   Arrays.compare(string1, 8, 11, string2, 0, 3));
```

Diese Zeilen liefern folgende Ausgaben, die die zuvor beschriebene Funktionsweise verdeutlichen:

```
compare(all):      -3
compare(DEF, DEF): 0
compare(GHI, DEF): 3
```

Einelementige Collections und `Collections.singletonList()`

Wenn man eine Liste übergeben oder zurückliefern soll, jedoch nur ein einzelnes Datenelement hat, so kann man dieses durch Aufruf der statischen Methode `Collections.singletonList(T)` in eine Liste mit einem einzigen gespeicherten Element umwandeln, die zudem unmodifizierbar ist:

```
final List<Image> thumbnailImages = Collections.singletonList(thumbnailImage);
```

Häufig sieht man jedoch zur Konstruktion einelementiger Listen folgende Zeilen:

```
final List<Image> thumbnailImages = new ArrayList<>();

thumbnailImages.add(thumbnailImage);
```

Nutzt man stattdessen die Methode `Collections.singletonList(T)` wird die Lesbarkeit des Sourcecodes besser. Es wird zudem klarer kommuniziert, dass es sich um eine einelementige, unveränderliche Liste handelt, der nachträglich keine weiteren Elemente hinzugefügt werden können.

Tipp: API-Unschönheit

Im Collections-Framework sind zum Erzeugen einelementiger Collections folgende generische und typsichere Methoden definiert:

- `Set<T> singleton(T o)`
- `List<T> singletonList(T o)`
- `Map<K,V> singletonMap(K key, V value)`

Beachten Sie die Inkonsistenz: Die erste Methode zum Erzeugen eines einelementigen Sets heißt nicht, wie man erwarten dürfte, `singletonSet()`, sondern lediglich `singleton()`.

Null-Objekte mit `Collections.emptyList()`/`emptyIterator()` usw.

Über die Methoden `emptyList()`, `emptyMap()`, `emptySet()` werden typsichere Konstanten für leere Container bereitgestellt. Es lassen sich auf diese Weise leere, unveränderliche Container realisieren.

Mit JDK 7 wurde die Utility-Klasse `Collections` um folgende Null-Objekte für Iteratoren bzw. Enumerationen erweitert:

```
public static <T> Iterator<T> emptyIterator()
public static <T> ListIterator<T> emptyListIterator()
public static <T> Enumeration<T> emptyEnumeration()
```

6.3.2 Dekorierer `synchronized` und `unmodifiable`

Im Kontext von Multithreading ist es oftmals hilfreich, Containerklassen unveränderlich zu machen und Zugriffe darauf zu synchronisieren. Das Collections-Framework stellt für beides spezifische Hilfsklassen bereit. Diese Erweiterungen erfolgen gemäß dem DEKORIERER-Muster (vgl. Abschnitt 18.2.3) und sind dadurch kombinierbar.

`synchronized`-Collections

Die in diesem Kapitel vorgestellten Containerklassen sind nicht Thread-sicher, unter anderem, weil ihre Methoden unsynchronisiert realisiert sind. Dadurch kann es bei Multithreading zu Inkonsistenzen durch konkurrierende Zugriffe kommen. Zur Abhilfe lässt sich jeder einzelne Methodenzugriff mittels der `synchronized`-Wrapper um Synchronisierung erweitern.

Allerdings wird leicht der Fehler gemacht, derart ummantelte Container bzw. die von Hause aus synchronisierten Container `Vector<E>` und `Hashtable<K,V>` als vollständig Thread-sicher zu betrachten. Jeder einzelne Methodenzugriff erfolgt zwar synchronisiert, wodurch er für sich gesehen geschützt ist und nach jedem Methodenaufruf den Container in einem gültigen Zustand hinterlässt. Trotzdem kann es zu Inkonsistenzen kommen, ***weil beim Multithreading die Kombination Thread-sicherer Methoden keine Thread-Sicherheit garantiert***. Das klingt zunächst paradox. Sie werden aber an den folgenden zwei einfachen Beispielen die Problematik schnell erkennen. Weitere Details beschreibt Kapitel 10.

Zugriff auf Elemente Betrachten wir eine typische Situation beim Zugriff auf eine Collection. Zunächst wird mithilfe der Methode `size()` die Anzahl der Elemente geprüft und anschließend per `get(int)` indiziert zugegriffen:

```
final Vector<Person> personsAsVector = new Vector<>(getAllMembers());
final int elementCount = personsAsVector.size();
// Sicherstellen, dass Elemente vorhanden sind
if (elementCount > 0)
{
    // IndexOutOfBoundsException wenn parallel clear() aufgerufen wird
    final Person first = personsAsVector.get(0);

    // IndexOutOfBoundsException wenn irgendeine Veränderung erfolgt,
    // etwa durch Aufruf von removeAt() oder clear()
    final Person last = personsAsVector.get(elementCount - 1);
}
```

Nachdem über die Methode `size()` die Anzahl der Elemente ermittelt wurde, kann man bei Multithreading nicht in jedem Fall davon ausgehen, dass der Wert zum Zeit-

punkt des nächsten eigenen Zugriffs auf die Collection unverändert ist. Dies liegt daran, dass die Ausführung eines laufenden Threads nahezu jederzeit unterbrochen werden kann. Im ungünstigsten Fall geschieht dies nach Abfrage der Größe und bevor man lesend per `get(int)` zugreift. Wird die Collection von einem parallel laufenden Thread verändert, so kommt es beim Zugriff entweder zu Dateninkonsistenzen (z. B. durch falsche Indizes) oder zu Exceptions.

IDIOM: THREAD-SICHERE ITERATION Während ein Thread eine Iteration über eine synchronisiert ummantelte Collection durchführt, sind konkurrierende Zugriffe durch andere Threads nicht ausgeschlossen, da, wie bereits erwähnt, nur jeder Zugriff für sich, nicht aber eine Folge geschützt abläuft. Wird demnach ein anderer Thread während der Iteration aktiv, so könnte er die Datenstruktur ändern und dadurch beim nächsten Zugriff des Iterators eine `ConcurrentModificationException` auslösen. Man nutzt dann das folgende IDIOM DER THREAD-SICHEREN ITERATION und sichert mit dem Schlüsselwort `synchronized` die Iteration als Ganzes:

```
final List<Person> persons = getAllMembers();
final List<Person> syncPersons = Collections.synchronizedList(persons);
synchronized (syncPersons)
{
    final Iterator<Person> it = syncPersons.iterator();
    while (it.hasNext())
    {
        final Person person = it.next();
        // Aktionen auf dem Person-Objekt
    }
}
```

Das `synchronized (syncPersons)` verhindert, dass die Anweisungen innerhalb des Blocks gleichzeitig von anderen Threads ausgeführt werden können. Allerdings werden so auch alle anderen über `syncPersons` synchronisierten Zugriffe auf die Datenstruktur für die Dauer der Iteration blockiert. Dadurch kann eine Parallelverarbeitung (selbst parallele, rein lesende Zugriffe) empfindlich gestört werden. Als Abhilfe können die in Abschnitt 11.1 beschriebenen Concurrent Collections zum Einsatz kommen.

`unmodifiable`-Collections

Die `unmodifiable`-Wrapper dienen dazu, eine unveränderliche Sicht, also ein Read-only-Interface, auf eine Collection bereitzustellen. Im Gegensatz zu den zuvor vorgestellten `synchronized`-Wrappern, die Synchronisierungsfunktionalität hinzufügen, werden durch die `unmodifiable`-Wrapper Veränderungsmöglichkeiten »entfernt«.[21] Dies kann zum einen gewünscht sein, um eine Collection nach ihrer Erstellung unveränderlich zu machen. Zum anderen hilft es, Konsistenz im Zusammenspiel verschiedener Komponenten zu wahren. Klienten besitzen lediglich eine unveränderliche Sicht auf die Daten. Die eigene Klasse hat weiter vollen Zugriff. Aufgrund der bereits ausführlich

[21] Natürlich können Methoden nicht im Nachhinein aus einem Interface entfernt werden. Stattdessen löst ein Aufruf solcher Methoden eine `UnsupportedOperationException` aus.

diskutierten Referenzsemantik kann allerdings nur ein Teil der durch den Namen suggerierten Intention erfüllt werden: Nur die Collection selbst wird so vor Veränderungen geschützt, dort gespeicherte Elemente sind jedoch potenziell weiterhin veränderlich.

Beispiel Betrachten wir zum Verständnis folgendes Beispiel eines Datenmodells `DataModel`, das ein Read-only-Interface `IDataAccessRO` zum lesenden Datenzugriff anbietet. In diesem Modell soll eine beliebige Anzahl von Objekten des Typs `ModelElement` verwaltet werden können. Abbildung 6-10 zeigt das UML-Klassendiagramm.

Abbildung 6-10 *Read-only-Datenmodell*

Die Zugriffsmethode `getAllElements()` liefert eine Referenz auf eine Liste von `ModelElement`-Objekten. Somit können aber Nutzer die interne Zusammensetzung des Datenmodells beliebig durch Aufruf von Methoden des `List<ModelElement>`-Interface verändern. So können etwa unerwartet `ModelElement`-Objekte neu eingefügt oder gelöscht werden, obwohl das Datenmodell für Klienten lediglich Methoden mit rein lesendem Zugriff bereitstellt. Wie bereits in Kapitel 3 besprochen, kann als Abhilfe entweder eine Kopie der Liste erstellt oder die Technik READ-ONLY-INTERFACE für Container angewendet werden. Das Collections-Framework bietet dazu verschiedene Klassen. Zur Korrektur bzw. zur Vermeidung von Veränderlichkeit nutzen wir einen Aufruf von `unmodifiableList()` wie folgt:

```
public List<ModelElement> getAllElements()
{
    return Collections.unmodifiableList(modelElements);
}
```

Ein Versuch, die zurückgegebene Liste zu modifizieren, würde nun eine Exception auslösen. Dieser Schutz vor Modifikationen ist für einige Anwendungsfälle bereits ausreichend. Aufgrund der Referenzsemantik kann jedoch nur ein Teil der Intention einer unveränderlichen Datenstruktur erfüllt werden. Die gespeicherten Elemente sind weiterhin veränderlich, sofern sie nicht als Immutable-Klasse realisiert sind. Das wurde bereits ausführlich in Abschnitt 3.4.1 diskutiert.

Es gibt aber noch einen unerwarteten Effekt. Es wird nämlich eine unveränderliche Sicht angeboten, die jedoch Änderungen an den Ausgangsdaten reflektiert.

```
public static void main(final String[] arguments)
{
    final List<String> originalStrings = new ArrayList<>();
    originalStrings.add("Tim");
    originalStrings.add("Tom");
    originalStrings.add("Peter");

    // JDK und Guava für unmodifizierbare Collections nutzen
    final List<String> unmodifiables =
                        Collections.unmodifiableList(originalStrings);
    final ImmutableList<String> immutables =
                                ImmutableList.copyOf(originalStrings);
    System.out.println("Initial List of Strings:      " + unmodifiables);

    // Peter im Original entfernen ... indirekt auch in unmodifiableList()
    originalStrings.remove("Peter");

    System.out.println("Original List of Strings:     " + originalStrings);
    System.out.println("Unmodifiable List of Strings: " + unmodifiables);
    System.out.println("Immutable List of Strings:    " + immutables);

    // Entfernen nicht erlaubt
    unmodifiables.remove("Tom"); // java.lang.UnsupportedOperationException
}
```

Listing 6.31 *Ausführbar als* '**UNMODIFIABLEUNEXPECTEDEXAMPLE**'

Nach dem Start des Programms UNMODIFIABLEUNEXPECTEDEXAMPLE wird man – ohne Kenntnis des einleitenden Satzes – ein wenig verwundert sein, dass die als unveränderlich konstruierte Liste durch Änderungen am Original auch modifiziert wird, hier symbolisch durch das Entfernen vom Eintrag "Peter" dargestellt:

```
Initial List of Strings:      [Tim, Tom, Peter]
Original List of Strings:     [Tim, Tom]
Unmodifiable List of Strings: [Tim, Tom]
Immutable List of Strings:    [Tim, Tom, Peter]
Exception in thread "main" java.lang.UnsupportedOperationException
  at java.util.Collections\$UnmodifiableCollection.remove(Collections.java:1118)
```

In der Regel ist das nicht konform zu der Erwartungshaltung einer unveränderlichen Datenstruktur. Wenn man wirklich eine unveränderliche Kopie der Listenzusammensetzung benötigt, dann kann man Google Guava einsetzen, wie es im obigen Beispiel mit der `com.google.common.collect.ImmutableList` gezeigt ist. Aufgrund der Referenzsemantik sind die einzelnen Listenelemente aber auch hier nach wie vor veränderlich, sofern sie nicht selbst wieder als Immutable-Klasse realisiert sind.

Seit Java 9 bieten sich auch die in Abschnitt 6.1.13 beschriebenen Collection-Factory-Methoden wie `List.of()` an.

Tipp: Einbinden von Google Guava

Google Guava enthält eine Menge nützlicher Utility-Klassen. Zum Einbinden in unsere Projekte notieren wir die Abhängigkeit in der Datei `build.gradle` wie folgt:

```
compile 'com.google.guava:guava:29.0-jre'
```

6.3.3 Vordefinierte Algorithmen in der Klasse `Collections`

Das Collections-Framework stellt durch die Klasse `java.util.Collections` diverse Algorithmen bereit, die bei der täglichen Arbeit gewinnbringend einsetzbar sind.

Anhand der Speicherung und Verarbeitung von `Person`-Objekten wollen wir einen kurzen Blick auf einige der angebotenen Algorithmen werfen. Nehmen wir dazu an, dass die Klasse `Person` das Interface `Comparable<Person>` implementiert.

Die Methoden `nCopies()` und `frequency()`

Durch Aufruf der Methode `nCopies(int,T)` wird eine `List<T>` erzeugt, die n-mal das übergebene Objekt vom Typ `T` enthält. Mit einem Aufruf der Methode `frequency(Collection<?>,Object)` kann man die Anzahl der gemäß `equals(Object)` gleichen Objekte innerhalb einer Collection bestimmen. Ich persönlich hätte `count()` als Methodennamen verständlicher gefunden.

Wir definieren zwei Listen, einmal mit den Werten von 1 bis 7 und dann eine mit viermal der Zahl 7 – für Letzteres nutzen wir die Methode `nCopies()`. Zum Zählen der Einträge verwenden wir einen Aufruf von `frequency()`:

```
public static void main(final String[] args)
{
    final List<Integer> firstNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
    final List<Integer> sevens = Collections.nCopies(4, 7);

    System.out.println("count1: " + Collections.frequency(firstNumbers, 7));
    System.out.println("count2: " + Collections.frequency(sevens, 7));
}
```

Listing 6.32 *Ausführbar als* '**ALGORITHMSEXAMPLE**'

Wie erwartet, kommt es zu folgender Ausgabe auf der Konsole:

```
count1: 1
count2: 4
```

Die Methoden `min()` und `max()`

In der Klasse `Math` gibt es die statischen Methoden `min()` und `max()`. Diese ermitteln den kleinsten bzw. größten Wert zweier Zahlen. Gleichnamige Methoden sind in der Klasse `Collections` zur Bestimmung minimaler und maximaler Elemente innerhalb einer Collection definiert. Diese Methoden arbeiten standardmäßig mit der natürlichen Ordnung für Zahlen primitiver Typen bzw. basierend auf `Comparable<T>` für Objekte, können aber auch selbst definierte Komparatoren als Argument nutzen.

Für ein Beispiel fügen wir in eine `List<Person>` drei `Person`-Objekte ein. Dann nutzen wir zunächst die natürliche Ordnung, also `Comparable<Person>`. Diese ist durch die Implementierung der Klasse `Person` so realisiert, dass hier nach Name, Ort und Alter sortiert wird. Damit werden die Methoden `min()` und `max()` ausgeführt.

Anschließend kommt ein selbst definierter Komparator zum Einsatz, der die Städtenamen vergleicht und daraus das Maximum bestimmt:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Anton", "Tirol", 11));
    persons.add(new Person("Micha", "Zürich", 43));
    persons.add(new Person("Stefan", "Kiel", 43));

    // Bestimmung min() und max() mit Comparable
    final Person min = Collections.min(persons);
    final Person max = Collections.max(persons);
    System.out.println("Min: " + min);
    System.out.println("Max: " + max);

    // Bestimmung max() mit eigenem Komparator
    final Comparator<Person> cityComparator =
                             Comparator.comparing(Person::getCity);
    final Person maxCity = Collections.max(persons, cityComparator);
    System.out.println("Max city: " + maxCity);
}
```

Listing 6.33 *Ausführbar als* '**ALGORITHMSEXAMPLEMINMAX**'

Das Programm ALGORITHMSEXAMPLEMINMAX produziert folgende Ausgaben:

```
Min: Person: Name='Anton' City='Tirol' Age='11'
Max: Person: Name='Stefan' City='Kiel' Age='43'
Max city: Person: Name='Micha' City='Zürich' Age='43'
```

Die Methoden `shuffle()` und `replaceAll()`

Für einige Anwendungsfälle ist es praktisch, die Reihenfolge innerhalb einer Collection umzuordnen, etwa zur Präsentation eines zufälligen Vorschlags aus einer Bestenliste wie aus den beliebtesten Pizzen des Monats oder den Top-10-Bestsellern. Das kann über die Methode `shuffle(List<?>)` zufallsbasiert geschehen. Darüber hinaus lassen sich über `replaceAll(List<T>,T,T)` Elemente ersetzen.

Folgendes Listing zeigt den Einsatz beider Methoden. Zunächst ersetzen wir die Zahlen 2 und 3 mit dem Wert 7 und danach durchmischen wir die Elemente:

```
public static void main(final String[] args)
{
    final List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    // Ersetzungen
    Collections.replaceAll(numbers, 2, 7);
    Collections.replaceAll(numbers, 3, 7);
    System.out.println("All numbers after replace: " + numbers);

    // Umordnen
    Collections.shuffle(numbers);
    System.out.println("All numbers after shuffle: " + numbers);
    System.out.println("#7: " + Collections.frequency(numbers, 7));
}
```

Listing 6.34 *Ausführbar als* '**ALGORITHMSEXAMPLESHUFFLEREPLACEALL**'

Die Ausgaben des obigen Programms zeigen, dass es in der Liste durch die Ersetzung keine 2 und 3 mehr gibt, aber dreimal die 7. Zudem sieht man gut die Umordnung der Werte durch `shuffle()`:

```
All numbers after replace: [1, 7, 7, 4, 5, 6, 7, 8, 9, 10]
All numbers after shuffle: [5, 6, 7, 7, 7, 8, 10, 9, 1, 4]
#7: 3
```

6.4 Containerklassen: Generics und Varianz

Um einige Besonderheiten im Kontext von Generics und Polymorphie kennenzulernen, kommen wieder grafische Figuren mit einer gemeinsamen Basisklasse `BaseFigure` als Beispiel zum Einsatz – ähnliche Klassenhierarchien wurden auch schon in verschiedenen Abschnitten im Kapitel 3 zu OO-Design genutzt.

Bisher haben wir Generics und Collections ohne viel Nachdenken verwendet und dabei ganz natürlich Zuweisungen wie Folgende geschrieben:

```
final List<String> names = new ArrayList<>();
final Set<BaseFigure> figures = new HashSet<>();
```

In diesem Abschnitt wollen wir uns ein paar Details von Generics widmen, um mögliche Probleme beim Einsatz generischer Typen im Zusammenhang mit Collections und der Verarbeitung von Typen einer Vererbungshierarchie zu vermeiden.

Im obigen Beispiel sind die Containerklassen auf einen bestimmten Typ fixiert. In einer solchen Collection lassen sich trotzdem auch Objekte von Subtypen speichern, z. B. statt vom Typ `BaseFigure` auch solche vom Typ `Circle`. Für Arrays kann man problemlos folgende Konstrukte verwenden:

```
final Object[] names = new String[10];
final BaseFigure[] figures = new Circle[10];
```

Problematisch wird es aber, diese mit Generics abzubilden. Intuitiv könnte man Folgendes schreiben wollen:

```
// Achtung: Kompilierfehler
final List<Object> names = new ArrayList<String>();
final Set<BaseFigure> figures = new HashSet<Circle>();
```

Diese kovariante Definition ist so für generische Container nicht erlaubt und führt zu Kompilierfehlern. Das Ganze werde ich nun genauer behandeln.

Generics, Invarianz, Kovarianz und Polymorphie

Nachfolgendes Beispiel zeigt die gewohnte, schon genutzte Definition typsicherer Containerklassen: Man kann eine Instanz eines spezifischeren Typs eines Containers, hier `HashSet<BaseFigure>`, einer Variablen eines allgemeineren Typs eines Containers, hier `Set<BaseFigure>`, zuweisen. Dabei sind die Typparameter invariant und die Typen der Container kovariant:

```
final Set<BaseFigure> figures = new HashSet<BaseFigure>();
//     ^      ^                          ^          ^
//     |      |_____ Typparameter invariant ____|
//     |                                |
//     |_______ Typen kovariant _______|
```

Mit ***Invarianz*** ist für Generics gemeint, dass ***die Typparameter bei Deklaration und Definition exakt übereinstimmen.*** Diese Forderung ist sehr restriktiv. ***Kovarianz*** bedeutet, dass die Typen der Vererbungshierarchie folgen. Dies ist hier für das Interface `Set<E>` und die konkrete Realisierung `HashSet<E>` gegeben.

Bemerkenswert ist, dass sich die Polymorphie durch Kovarianz bei generischen Definitionen nur auf den Typ der Containerklasse bezieht und nicht auf die Typparameter (und hat damit eigentlich gar nichts mit Generics zu tun)!

Polymorphie für Typen So weit scheint die obige Definition selbstverständlich zu sein. Allerdings muss man bei Generics penibel darauf achten, dass der Typparameter invariant ist. Weil dies – wie wir später noch ausführlicher sehen werden – ein häufiger Stolperstein beim Einsatz von Generics ist, schauen wir uns das nun etwas genauer an. Dazu betrachten wir die Ableitungshierarchie `RectFigure extends BaseFigure`. Man könnte annehmen, dass Folgendes erlaubt wäre:

```
// Compile-Error
final Set<BaseFigure> figures = new HashSet<RectFigure>();
//            ^                          ^
//            |____ Typparameter kovariant _____|
```

Für eine solche Definition wären die gezeigten Typparameter nicht mehr invariant, sondern kovariant, da der Typ `RectFigure` von `BaseFigure` abgeleitet ist. Diese Schreibweise scheint intuitiv korrekt, birgt aber Probleme bei der Typsicherheit.

Mögliche Probleme durch Kovarianz

Erinnern wir uns daran, dass Arrays kovariant und Generics invariant arbeiten. Beginnen wir die Betrachtung mit Arrays und schauen, was dort mit Kovarianz gemeint ist.

Wie schon aus Kapitel 3 bekannt, kann ein `Object[]` beispielsweise sowohl ein `String[]` als auch ein `BaseFigure[]` repräsentieren, jedoch nicht gleichzeitig.

```
// kovariante Zuweisung an Object[]; jetzt Referenz auf String[]
Object[] objects = new String[] { "Test1", "Test2" };
objects[0] = "Some string content";
```

Obwohl im Listing das `Object[]` zunächst mit einem `String[]` initialisiert wird, kann problemlos später eine Zuweisung mit dem Typ `BaseFigure[]` erfolgen:

```
// kovariante Zuweisung an Object[]; jetzt Referenz auf BaseFigure[]
objects = new BaseFigure[10];
// Laufzeitfehler: java.lang.ArrayStoreException: java.lang.String
objects[0] = "This is not a BaseFigure";
```

Im Listing sehen wir, dass an Position 0 des Arrays ein String zugewiesen wird, ohne dass dies einen Fehler beim Kompilieren auslöst: Semantisch ist die gezeigte Aktion natürlich nicht sinnvoll, jedoch finden für Arrays mit dem Basistyp `Object[]` zur Kompilierzeit keine Typprüfungen statt. Eine derartige Array-Referenz kann zur Laufzeit demnach beliebige, nicht primitive Array-Typen repräsentieren, etwa `String[]` oder auch `BaseFigure[]`. ***Typverstöße oder mögliche Zuweisungsfehler werden erst zur Laufzeit erkannt***: Dazu benötigen und besitzen Arrays zur Laufzeit entsprechende Typinformationen über die in ihnen gespeicherten Elemente. Liegen tatsächlich Inkompatibilitäten bei Zuweisungen vor, so wird eine `ArrayStoreException` ausgelöst. Allerdings habe ich eine solche in über 20 Berufsjahren – außer für Beispiele provoziert – bisher nicht gesehen.

Trotzdem war Sun beim Entwurf der Generics angetreten, eine möglichst wasserdichte Typprüfung zu erreichen. Nehmen wir einmal an, wir könnten auch für Generics kovariante Definitionen vornehmen. Damit wäre dann etwa Folgendes möglich:

```
// Compile-Error
final List<Object> names = new ArrayList<String>();
//            ^                            ^
//            |__ Typparameter kovariant ___|

names.add(new RectFigure());  // Typinkompatibilität
```

Ebenso wie für Arrays sollte dies dann zur Laufzeit zu einem Fehler führen, wenn man einer `List<Object>` mal eine `List<String>` oder `List<BaseFigure>` zuweist und Elemente hinzufügt. Um den Typverstoß aber erkennen zu können, müsste man für die Containerklassen des JDKs analog zu Arrays zur Laufzeit Typinformationen zu den gespeicherten Elementen besitzen. Nun wurde aber beim Entwurf von Generics die Entscheidung getroffen, Generics mithilfe von Type Erasure (vgl. Abschnitt 3.7.2) umzusetzen. Aufgrund dessen ist eine Typprüfung zur Laufzeit keine Option, da nach dem Kompilieren keine Typinformationen mehr vorhanden sind, die ausgewertet werden könnten. Wäre obige kovariante Definition also erlaubt, so könnte der Compiler hier nicht mehr sicherstellen, dass zur Laufzeit keine Typinkompatibilitäten auftreten. Daher führt die gezeigte Definition bereits beim Kompilieren zu einem Fehler.

Weil Kovarianz als potenziell gefährlich eingestuft wurde und es ein wesentliches Ziel beim Entwurf von Generics war, Laufzeitfehler zu vermeiden, gilt hier eine strengere Forderung, nämlich die nach ***Invarianz*** der Typparameter. Manchmal ist Kovarianz allerdings nützlich, wie wir nun sehen werden.

Warum ist Kovarianz so wünschenswert?

Kovarianz ist immer dann wünschenswert, wenn eine »is-a«-Beziehung zwischen den zu speichernden Typen besteht. Die mit Generics standardmäßig realisierte Invarianz erschwert jedoch den Umgang.

Betrachten wir das anhand eines Beispiels und nutzen zur Datenspeicherung sowohl ein Array `BaseFigure[]` als auch eine Liste `List<BaseFigure>` wie folgt:

```
final BaseFigure[] arrayOfFigures = new BaseFigure[10];
final List<BaseFigure> listOfFigures = new ArrayList<BaseFigure>();
```

Nehmen wir weiterhin an, eine Methode `printInfo(BaseFigure)` zur Ausgabe von Informationen zu einer Figur sei in einer Utility-Klasse `FigureUtilities` für einzelne Elemente vom Typ `BaseFigure` folgendermaßen definiert:

```
public void printInfo(final BaseFigure figure)
{
    figure.printDetails();
}
```

Schauen wir uns nun an, wie einfach es ist oder auf welche Probleme man stößt, wenn man die Utility-Klasse so erweitern soll, dass eine überladene Methode für Arrays sowie auch für Collections von Figuren genutzt werden kann. Die geforderte Funktionalität lässt sich für Arrays und Listen wie folgt leicht realisieren:

```
public void printInfo(final BaseFigure[] figures)
{
    for (final BaseFigure figure : figures)
        printInfo(figure);
}

public void printInfo(final List<BaseFigure> figures)
{
    for (final BaseFigure figure : figures)
        printInfo(figure);
}
```

In beiden Varianten werden alle Elemente durchlaufen und für jede Figur polymorph die Methode `printInfo(BaseFigure)` aufgerufen. Aber es gibt einen funktionalen Unterschied, wie wir nachfolgend sehen werden.

Beginnen wir damit, die Methoden mit unterschiedlichen Eingabewerten aufzurufen. Dazu erzeugen wir Objekte unterschiedlicher Figurenklassen. Betrachten wir dies für folgende drei Arrays mit Kreisen, Rechtecken und einem Mix daraus:

```
public static void main(final String[] args)
{
    // Definitionen von Arrays unterschiedlichen Typs
    final CircleFigure[] circles = { new CircleFigure(), new CircleFigure() };
    final RectFigure[] rects = { new RectFigure(), new RectFigure() };
    final BaseFigure[] figures = { new CircleFigure(),  new RectFigure() };

    // Ausgabe über die Methode printInfo(BaseFigure[])
    printInfo(circles);
    printInfo(rects);
    printInfo(figures);
}
```

Listing 6.35 *Ausführbar als* **'GENERICSARRAYPOLYMORPHYEXAMPLE'**

Für Arrays des jeweiligen Figurentyps bzw. des Basistyps `BaseFigure` kann die zuvor definierte Methode `printInfo(BaseFigure[])` problemlos aufgerufen werden. Beim Start des Programms GENERICSARRAYPOLYMORPHYEXAMPLE kommt es erwartungskonform zu folgenden Ausgaben:

```
Circle
Circle
Rect
Rect
Circle
Rect
```

Dadurch motiviert, versuchen wir nun, das Ganze auf generische Collections zu übertragen. Intuitiv könnte man auf die im folgenden Beispiel dargestellte Umsetzung kommen, die analog zur vorherigen Realisierung arbeitet, jedoch mithilfe von `Arrays.asList(T...)` zunächst eine Umwandlung der typisierten Arrays in typsichere Listen vornimmt, um dann die Methode `printInfo(List<BaseFigure>)` aufzurufen:

```
public static void main(final String[] args)
{
    // Identische Definitionen
    final CircleFigure[] circles = { new CircleFigure(), new CircleFigure() };
    final RectFigure[] rects = { new RectFigure(), new RectFigure() };
    final BaseFigure[] figures = { new CircleFigure(), new RectFigure() };

    // Umwandlung Array -> Liste (Arrays.asList(T...))
    final List<BaseFigure> figureList = Arrays.asList(figures);
    printInfo(figureList);

    // Compile-Error: Type mismatch: cannot convert from
    // List<CircleFigure> to List<BaseFigure>
    // final List<BaseFigure> circleList = Arrays.asList(circles);

    // Compile-Error: The method printInfo(List<BaseFigure>) in the type
    // GenericsArrayPolymorphy2Example is not applicable
    // for the arguments (List<RectFigure>)
    // printInfo(Arrays.asList(rects));
}
```

Listing 6.36 *Ausführbar als* **'GENERICSARRAYPOLYMORPHY2EXAMPLE'**

Das Programms GENERICSARRAYPOLYMORPHYEXAMPLE produziert folgenden Ausgaben:

```
Circle
Rect
```

Daran erkennt man das nur das in eine Liste umgewandelte `BaseFigure[]` problemlos den Aufruf der Methode `printInfo(List<BaseFigure>)` erlaubt, da hier der Typparameter invariant ist. Bei den beiden Arrays der spezielleren Typen kommt es dagegen zu Kompilierfehlern. Dabei ist es unbedeutend, ob zunächst eine Umwandlung in eine `List<BaseFigure>` erfolgt oder ein direkter Aufruf. Es stellen sich die folgenden Fragen: »Wie kommt es dazu und wie lösen wir das Problem?«

Problemlösung Um die mitunter wünschenswerte Kovarianz für Typparameter bei Generics zu ermöglichen, existiert eine spezielle Notation. Allerdings darf aufgrund der Type Erasure und der fehlenden Möglichkeit einer Typprüfung zur Laufzeit nur eine »sichere Form« der Kovarianz unterstützt werden, bei der garantiert wird, dass der verwendete Typ alle Operationen des Basistyps implementiert. Gleiches gilt für Kontravarianz. Für beide kommt eine sogenannte Wildcard zum Einsatz. Die Wildcard '`?`' erlaubt beliebige Typen und kann zur Kennzeichnung von Ko- und Kontravarianz wie folgt genutzt werden:

1. **Kovarianz** – Die Wildcard '`? extends` **`basetype`**' wird ***Upper Type Bound*** genannt und ermöglicht ***Kovarianz***. Es dürfen dadurch generische Klassen verwendet werden, die Subtypen von `basetype` als Typparameter nutzen.[22]
2. **Kontravarianz** – Die Wildcard '`? super` **`subtype`**' heißt ***Lower Type Bound*** und ermöglicht ***Kontravarianz***. Dadurch sind alle diejenigen generischen Klassen erlaubt, die als Typparameter einen Basistyp von `subtype` besitzen. Es werden für die Typparameter also *alle* Basistypen akzeptiert, jedoch *keine* Subtypen mehr.

Kovarianz – Upper Type Bound

Weil durch die Type Erasure zur Laufzeit keine Typinformationen existieren, ist Kovarianz bei Generics nur mit Einschränkungen möglich. Probleme durch inkompatible Typen müssen bereits zur Kompilierzeit ausgeschlossen werden. Typparameter sind dazu mit der Syntax '`? extends basetype`' explizit als ***kovariant*** zu kennzeichnen.

Nutzen wir dieses Wissen, um das vorherige Beispiel alternativ sowohl für Figuren vom Typ `CircleFigure` als auch `RectFigure` ohne Fehler kompilieren zu können:

```
final List<? extends BaseFigure> figureList1 = new ArrayList<CircleFigure>();
final List<? extends BaseFigure> figureList2 = new ArrayList<RectFigure>();
```

[22] Für Interfaces existiert keine Notation mit `implements`, sondern es wird hier, analog zu den bereits bekannten Typeinschränkungen, das Schlüsselwort `extends` genutzt.

Einer derart definierten `List<? extends BaseFigure>` lässt sich problemlos eine `ArrayList<CircleFigure>` oder eine `ArrayList<RectFigure>` zuweisen. Worin besteht denn dann der Unterschied zu der Kovarianz bei Arrays? Die Antwort ist einfach: In der Angabe des Typparameters! Zuweisungsfehler können bei Arrays erst zur Laufzeit erkannt werden. ***Für Generics werden Schreibzugriffe auf Elemente bereits beim Kompilieren ausgeschlossen und problematische kovariante Zuweisungen vom Compiler verboten***.

Einfügen von Elementen Eingangs haben wir erkannt, dass Schreibzugriffe in eine kovariant definierte Collection problematisch sein können. ***Um Typsicherheit zu garantieren, werden Schreibzugriffe vom Compiler unterbunden und führen beim Kompilieren zu Fehlern***. Dadurch vermeidet man beispielsweise, dass sich in einer als `ArrayList<RectFigure>` konstruierten Liste (fälschlicherweise) ein `CircleFigure`-Objekt speichern lässt. Als einzige Ausnahme kann immer `null` verarbeitet werden, da dieser Wert zu allen Referenztypen zuweisungskompatibel ist.

Zugriff auf Elemente Für den Zugriff auf Elemente gilt, dass es immer problemlos möglich ist, Referenzen vom Typ `BaseFigure` auszulesen, da die gespeicherten Elemente garantiert mindestens diesen Typ besitzen:

```
final BaseFigure value = figureList.get(index);
```

Das Auslesen eines Subtyps von `BaseFigure`, z. B. `CircleFigure`, ist jedoch nicht möglich, da die durch '`? extends BaseFigure`' repräsentierte, konkrete Klasse unbekannt ist. Damit wird etwa verhindert, dass aus einer `ArrayList<RectFigure>` ein `CircleFigure`-Objekt gelesen werden kann:

```
// Compile-Error: Type mismatch: cannot convert from capture#3-of ? extends
// BaseFigure to RectFigure
final RectFigure value = figureList.get(index);
```

Kontravarianz – Lower Type Bound

Die Forderung nach Kovarianz ist relativ natürlich und basiert auf dem Substitutionsprinzip. Der Einsatz von Kontravarianz ist weniger eingängig, weil diese eine Kompatibilität entgegengesetzt zur Vererbungshierarchie der Typparameter ermöglicht. Kontravarianz bei Generics wird durch die Notation '`? super subtype`' ausgedrückt:

```
final List<? super BaseFigure> figureList = new ArrayList<BaseFigure>();
```

Im Speziellen ist sogar folgende Zuweisung möglich:

```
final List<? super BaseFigure> figureList = new ArrayList<Object>();
```

Durch die Beschränkung des Typs nach »unten« führt folgende Definition zu einem Fehler beim Kompilieren:

```
// Compile-Error
// cannot convert ArrayList<RectFigure> to ArrayList<? super BaseFigure>
final List<? super BaseFigure> figureList = new ArrayList<RectFigure>();
```

Einfügen von Elementen Für eine kontravariant deklarierte Liste ist das Hinzufügen von Elementen möglich. Im nachfolgenden Beispiel nutzen wir als begrenzenden Typ `BaseFigure` und fügen mit `RectFigure` eine Spezialisierung davon ein:

```
final List<? super BaseFigure> figureList = new ArrayList<BaseFigure>();
figureList.add(new RectFigure());
```

In diesem Beispiel fällt eine Besonderheit auf, die für Verwirrung sorgen kann. Obwohl die Liste mit dem Typparameter `BaseFigure` definiert wurde, kann tatsächlich (zunächst irritierenderweise) jeder Subtyp von `BaseFigure` gespeichert werden, wie dies bereits im obigen Beispiel durch den Aufruf von `figureList.add(new RectFigure())` gezeigt wurde. Folgendes ist wichtig zu wissen: ***Die Varianz bezieht sich auf die Typparameter der generischen Klasse (`ArrayList<E>`) und nicht auf die Typen der tatsächlich gespeicherten Elemente.***

Zugriff auf Elemente Beim Einsatz von Kontravarianz kann der Compiler nicht feststellen, von welchem Typ die Elemente der Collection tatsächlich sind. ***Aus einer solchen Collection kann deshalb nicht typsicher gelesen werden***.

```
// Compile-Error: Type mismatch: cannot convert from capture#3-of ? super
// BaseFigure to RectFigure
final RectFigure value = list.get(index);
```

Allerdings besitzen die enthaltenen Elemente immer den Basistyp `Object`, wodurch folgender Lesezugriff möglich ist:

```
final Object obj = list.get(index);
```

Invarianz

Bekanntermaßen sind bei der Invarianz für Generics die Typparameter bei Deklaration und Definition gleich. Im folgenden Beispiel ist dies für eine `ArrayList` von `BaseFigure`-Objekten gezeigt:

```
final List<BaseFigure> graphicObjects = new ArrayList<BaseFigure>();
```

Eine auf diese Weise definierte `ArrayList<BaseFigure>` lässt sich so nutzen, als ob die Signaturen der Methoden tatsächlich für `BaseFigure`-Objekte definiert wären:

Beim Auslesen mit `get(int)` liefert sie ein Objekt vom Typ `BaseFigure`. Beim Hinzufügen lässt sich mit `add(BaseFigure)` ein Element vom Typ `BaseFigure` bzw. sogar Subtypen davon speichern:

```
final BaseFigure baseFigure = graphicObjects.get(index);
graphicObjects.add(new RectFigure());
```

Info: Invarianz und Auto-Boxing

Für Collections von Wrapper-Klassen, etwa `ArrayList<Long>`, erfolgt automatisch Auto-Boxing/-Unboxing beim Hinzufügen oder Auslesen von Elementen:

```
final List<Integer> integerList = new ArrayList<>();
integerList.set(7, 4711);                          // Auto-Boxing
final int valueAtPosition7 = integerList.get(7);   // Auto-Unboxing
```

Auswirkungen der Varianzformen

Die Vorstellung der Varianzformen war möglicherweise ein wenig theoretisch. Der Nutzen in der Praxis lässt sich gut an einem Beispiel verdeutlichen.

Nehmen wir an, die bereits kurz genannte Utility-Klasse `FigureUtilities` soll um eine Methode `copy()` erweitert werden, mit der man grafische Figuren aus einer Liste in eine andere Liste kopieren kann. Eine erste Variante wäre, die Methoden folgendermaßen zu realisieren:

```
public static void copy(final List<BaseFigure> src,
                        final List<BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);
}
```

So plausibel diese Realisierung zunächst auch aussehen mag, sie besitzt einige Einschränkungen und ist wenig hilfreich. Betrachten wir mögliche Probleme und was man dagegen unternehmen kann.

Problem Invarianz Das erste Problem der gezeigten Realisierung besteht in der invarianten Definition der Typparameter von Quelle und Ziel:

```
final List<BaseFigure> src = new ArrayList<BaseFigure>();
// Füllen der Liste ...
final List<BaseFigure> dest = new ArrayList<BaseFigure>();

FigureUtilities.copy(src, dest);
```

Dadurch sind wir darauf beschränkt, beim Kopieren jeweils nur Listen mit identischem Typparameter zu nutzen. Das ist jedoch störend, wenn man Subtypbeziehungen und Polymorphie nutzt.

Rekapitulieren wir kurz das Verhalten für Referenzen und Arrays: Man kann z. B. eine Referenz `CircleFigure` an den Typ `BaseFigure` zuweisen und so auf allgemeinerer Ebene arbeiten. Für Arrays wäre es auch möglich, einem `BaseFigure[]` ein spezielleres `CircleFigure[]` zuzuweisen. Ähnliches wäre auch für generische Container wünschenswert: Ein möglicher Nutzer der Utility-Klasse könnte dann eine `List<CircleFigure>` in eine Liste mit dem allgemeineren Typ `BaseFigure` übertragen. Das ist jedoch mit der zuvor gezeigten `copy()`-Methode nicht möglich! Stattdessen kommt es zu einem Kompilierfehler, wie dies folgende Zeilen zeigen:

```
final List<CircleFigure> src = new ArrayList<CircleFigure>();
// Füllen der Liste ...
final List<BaseFigure> dest = new ArrayList<BaseFigure>();

// Compile-Error: The method copy(List<BaseFigure>, List<BaseFigure>)
// in the type FigureUtilities is not applicable for the arguments
// (List<CircleFigure>, List<BaseFigure>)
FigureUtilities.copy(src, dest);
```

Motivation für Kovarianz für den Parameter der Quelle Als Abhilfe nutzen wir eine kovariante Definition folgendermaßen:

```
public static void copy(final List<? extends BaseFigure> src,
                        final List<BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);
}
```

Damit lässt sich die eben noch an einem Kompilierfehler scheiternde Kopieraktion von einer `ArrayList<CircleFigure>` in eine `ArrayList<BaseFigure>` durchführen.

Kovarianz auch für den Parameter des Ziels? Intuitiv fragt man sich: »Wäre es nicht praktisch, auch den Parameter für das Ziel kovariant zu definieren?« Das klingt zunächst eigentlich ganz vernünftig. Probieren wir es aus. Damit ergäbe sich folgende Realisierung:

```
public static void copy(final List<? extends BaseFigure> src,
                        final List<? extends BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);   // Compile-Error
}
```

Tatsächlich kommt es dadurch zu einem Kompilierfehler bei `add(BaseFigure)`. Das ist auch gut so, denn so verhindert man möglicherweise problematische Schreibzugriffe, wie wir dies bereits bei der Diskussion der Kovarianz bei Arrays besprochen haben. Nehmen wir kurz an, diese kovariante Methodendefinition wäre erlaubt, dann wäre z. B. folgende Kopie einer Liste von Kreisen in eine Liste von Rechtecken möglich:

```
final List<CircleFigure> src = new ArrayList<>();
final List<RectFigure> dest = new ArrayList<>();

FigureUtilities.copy(circles, rects);
```

Das würde aber zu Typfehlern führen, weil man Elemente vom Typ `CircleFigure` in eine Liste, festgelegt auf Elemente vom Typ `RectFigure`, einfügen würde. Um Derartiges grundsätzlich zu verhindern, sind Schreibzugriffe für kovariant definierte Containerklassen nicht erlaubt.

Motivation für Kontravarianz für den Parameter des Ziels Die initial genutzte Invarianz bot wenig Flexibilität. Auch für Kovarianz haben wir erkannt, dass es keinen Sinn ergibt, das Kopierziel derart zu definieren. Was ist mit Kontravarianz?

Beim Kopieren wäre es teilweise durchaus wünschenswert, ein allgemeineres Ziel angeben zu können, also statt einer `List<BaseFigure>` etwa eine `List<Object>`. Es ist typsicher, wenn man etwa eine Liste von Rechteckfiguren in einer Liste speichert, deren Typparameter `BaseFigure` bzw. `Object` ist. Durch Kontravarianz wird es möglich, auch eine auf einen allgemeineren Typ festgelegte generische Containerklasse als Kopierziel zu nutzen. Das erlaubt es, die Zieldatenstruktur flexibler angeben zu können. Betrachten wir die neue Variante der `copy()`-Methode, die dies nutzt:

```
public static void copy(final List<? extends BaseFigure> src,
                        final List<? super BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);
}
```

Mit dieser Methode haben wir eine flexible Kopierfunktionalität umgesetzt: Wir können mithilfe der obigen `copy()`-Methode eine typsichere Kopie erstellen und dabei sowohl speziellere Eingabe- als auch allgemeinere Rückgabetypen nutzen. Dadurch wird die Handhabung für Klienten vereinfacht, sodass nun etwa folgende Aufrufe möglich sind:

```
// Definition von Zieldatenstrukturen
final List<BaseFigure> baseFigures = new ArrayList<>();
final List<Object> objects = new ArrayList<>();

// Kovariante Eingabe und kontravarianter Zielparameter
FigureUtilities.copy(circles, baseFigures);
FigureUtilities.copy(rects, objects);
```

Sind wir damit wirklich am Ziel? Nein, aber fast. Was fehlt denn? Bei der gesamten Diskussion um Super- und Subtypen haben wir einen Fall bisher nicht betrachtet: Man sollte jede Liste eines speziellen Typs, etwa `CircleFigure`, selbstverständlich auch wieder in eine `List<CircleFigure>` kopieren können. Obwohl das banal klingt, ist es mit der realisierten `copy()`-Methode doch nicht möglich! Stattdessen kommt es zu dem im nachfolgenden Listing gezeigten Kompilierfehler:

```
final List<CircleFigure> circlesSrc = new ArrayList<>();
// Füllen der Liste ...
final List<CircleFigure> circlesDest = new ArrayList<>();

// Compile-Error: The method copy(List<? extends BaseFigure>,
// List<? super BaseFigure>) in the type FigureUtilities is not
// applicable for the arguments List<CircleFigure>, List<CircleFigure>)
FigureUtilities.copy(circlesSrc, circlesDest);
```

Korrekterweise wird erkannt, dass der Typ `List<CircleFigure>` des Ziels nicht dem Typ `List<? super BaseFigure>` zugewiesen werden kann. Es ist aber eine natürliche Forderung und wünschenswert, Elemente aus Listen des gleichen Typs kopieren zu können. Um dies zu ermöglichen, nutzen wir die bereits vorgestellte Syntax für Typeinschränkungen für statische Methoden wie folgt:

```
public static <T extends BaseFigure> void copy(final List<T> src,
                                               final List<? super T> dest)
{
    for (final T figure : src)
        dest.add(figure);
}
```

Nach dieser Modifikation haben wir nun unter Einsatz von Ko- und Kontravarianz eine typsichere, generische Kopiermethode implementiert.

Schlussfolgerung In diesem Beispiel wurden diverse kleinere Fallstricke und Besonderheiten beim Einsatz von Generics und Collections, aber auch Möglichkeiten zur Lösung vorgestellt. Damit haben Sie ein recht gutes Wissen erlangt, um eigene Experimente zu starten. Fassen wir noch mal zusammen: Der Einsatz von Kovarianz bietet sich an, um Eingaben allgemeiner zu gestalten. Kontravarianz erlaubt dies für Rückgaben bzw. Zuweisungen. Um Typsicherheit zu gewährleisten, sind Schreibzugriffe bei Kovarianz verboten, und durch den Compiler wird ein Read-only-Umgang forciert. Bei Kontravarianz sind Lesezugriffe verboten, aber Schreibzugriffe erlaubt. In Tabelle 6-1 sind die obigen Aussagen zusammengefasst.

Tabelle 6-1 *Konsequenzen der Varianzformen*

Varianzform	Lesezugriff	Schreibzugriff
Kontravarianz	- (nur `Object`)	✓
Kovarianz	✓	- (nur `null`)
Invarianz	✓	✓

Info: Generics und Wildcards

Nachdem wir die Verwendung von Wildcards kennengelernt und diese bereits in einigen Beispielen eingesetzt haben, ist uns aufgefallen, dass deren Verwendung mit einigen Merkwürdigkeiten in der Schreibweise verbunden ist: Dies gilt im Speziellen, wenn man komplexere generische Typen oder Methoden definiert, die eine Kombination aus `extends` und `super` nutzen, etwa wie folgt:

```
static <T> int binarySearch(List<? extends Comparable<? super T>> list,
                            T key)
```

Eine interessante Diskussion finden Sie unter `http://www.artima.com/weblogs/viewpost.jsp?thread=222021`. Einen Videovortrag von Joshua Bloch zu diesem Thema kann man sich unter `http://beta.parleys.com/#id=116&st=5&sl=37` anschauen.

Varianz nur für Eingabeparameter, niemals für Rückgabewerte Joshua Bloch gibt in »Effective Java« [6] den Hinweis, die beiden Varianzformen

- ? extends T (Kovarianz)
- ? super T (Kontravarianz)

nur für Eingabeparameter, niemals aber für Rückgabetypen zu verwenden.

Der Grund ist folgender: Für Methodeneingaben erreicht man für mögliche Aufrufer mehr Flexibilität. Für Rückgabewerte gilt dies nicht. Im Gegenteil: Es erschwert sogar die Handhabung, da die Wildcards dann auch in den Sourcecode der Klienten aufgenommen werden müssen.

Spezielle Syntax bei Generics

Wir haben mit Ko- und Kontravarianz und der ?-Notation schon eine recht kryptische Syntax bei Generics kennengelernt. Leider gibt es sogar noch eine Steigerung. Diese kommt dann zum Einsatz, wenn man Methoden mit generischen Parametern aufrufen möchte, etwa folgendermaßen:

```
public void print(final String title,
                  final List<TimeSeriesData> timeSeriesData);
```

Diese Signatur sieht harmlos aus, aber was passiert, wenn wir eine leere Liste als Parameter übergeben wollen? Dazu nutzen wir `Collections.emptyList()` wie folgt:

```
print("title", Collections.emptyList());
```

Auch wenn dieser Aufruf korrekt und logisch aussieht, so kommt es durch die Type Erasure zu folgender Fehlermeldung: »`The method print(String, List<TimeSeriesData>) in the type XYZ is not applicable for the arguments`

(String, List<Object>)«. Als Abhilfe muss man den Typ direkt angeben, und zwar mithilfe folgender kovarianter Syntax:

```
print("title", Collections.<TimeSeriesData>emptyList());
```

Unterschied zwischen List, List<Object> und List<?>

Nachdem nun die Formen der Varianz bekannt sind, möchte ich abschließend auf einige Unterschiede in den Deklarationen List, List<Object> und List<?> eingehen. Obwohl alle recht ähnlich aussehen, besitzen sie doch voneinander abweichende Eigenschaften, die wir nun erkunden.

List vs. List<Object> Etwas vereinfachend kann man sagen, dass die Deklarationen List und List<Object> nahezu gleich sind. Derart definiert können Elemente beliebigen Typs gespeichert werden. Der Unterschied besteht lediglich darin, welche Zuweisungen an die so definierten Referenzvariablen vorgenommen werden können: Eine Deklaration List<Object> erlaubt tatsächlich nur, dass exakt so definierte Listen zugewiesen werden. Das folgt aufgrund der Invarianz:

```
// beim Auslesen nicht typsicher
final List plainList = new ArrayList();
plainList.add(new Integer(4711));
plainList.add("Test");
final Integer sum = plainList.get(0) + // Laufzeitfehler, weil
                    plainList.get(1);  // Integer und String enthalten sind

// ähnlich wie List, total generisch
final List<Object> objectList = new ArrayList<Object>();
// final List<Object> objectList = new ArrayList<String>(); // Compile-Error
objectList.add("Test");
objectList.add(new Integer(4711));
```

List<Object> vs. List<?> Im Gegensatz zu den gerade betrachteten Deklarationen bestehen zwischen den Deklarationen List<Object> und List<?> sehr große Unterschiede. Wie eben erwähnt, können in einer List<Object> Elemente beliebigen Typs gespeichert werden, aufgrund der Invarianz jedoch nur Listen zugewiesen werden, die auch exakt den generischen Typ Object nutzen.

Für die List<?> gilt das Gegenteil: Die Wildcard '?' erlaubt beliebige Typen und somit können Listen mit beliebigen Typparametern zugewiesen werden:

```
final List<?> anyTypeList1 = new ArrayList<Person>();
final List<?> anyTypeList2 = new ArrayList<CircleFigure>();

// typsichere Liste, bei der der Typ unbekannt ist
final List<?> anyTypeList = new ArrayList<String>();

// Compile-Error: The method add(capture#1-of ?) in the type
// List<capture#1-of ?> is not applicable for the arguments (String)/(Object)
// anyTypeList.add("Test");
// anyTypeList.add("Object");

anyTypeList.add(null);    // erlaubt
```

Allerdings kann aufgrund der fehlenden Typinformation der zu speichernden Elemente keine Typsicherheit garantiert werden. ***Für `List<?>` ist daher ein Einfügen von Elementen generell nicht möglich***. Es kommt zu der im Listing gezeigten Fehlermeldung.

Zusammenfassung Fassen wir alles noch mal zusammen: Wie eingangs erwähnt, verhält sich eine `List<Object>` nahezu wie der nicht typisierte Raw Type `List` und definiert eine ***heterogene Liste***. Allerdings kann einer `List<Object>` aufgrund der Forderung nach Invarianz *keine* `List<String>` zugewiesen werden; es können der `List<Object>` aber durchaus Stringobjekte hinzugefügt werden.

Mit einer `List<?>` wird eine ***homogene Liste*** beschrieben: Es kann zwar eine `List<String>` oder `List<Person>` zugewiesen werden; es können aber *keine* Stringobjekte bzw. `Person`-Objekte hinzugefügt werden. Dies klingt nach einer starken Einschränkung. Trotzdem ist `List<?>` aber für Utility-Methoden nützlich, die nur auf der Listenfunktionalität unabhängig vom Typ arbeiten sollen, etwa `size(List<?>)`.

6.5 Die Klasse `Optional<T>`

Für Berechnungen kann mitunter kein Ergebnis ermittelt werden, beispielsweise bei einer erfolglosen Suche oder bei der Bestimmung des Maximums für eine leere Menge. Dann benötigt man eine Möglichkeit zur Modellierung optionaler oder nicht vorhandener Werte. Dies geschah bis JDK 8 in Form von `null` oder mithilfe des NULL-OBJEKT-Musters (vgl. Abschnitt 18.3.2). Allerdings sind Methoden, die potenziell `null` zurückgeben, in der Handhabung immer wieder problematisch. Ein Aufrufer weiß möglicherweise nichts von diesem Fall (oder hat den Hinweis im Javadoc einfach ignoriert) oder behandelt den Fall einfach nicht – was dann irgendwann einmal zu einer `NullPointerException` führt. Fast noch schlimmer sind jedoch übervorsichtige, ständige `null`-Prüfungen nach Aufrufen von Methoden, die niemals `null` zurückliefern – einfach aus Unwissenheit. Das bläht den Sourcecode nur auf und macht ihn schwer verständlich.

Um Optionalität besser ausdrücken und kommunizieren zu können, bietet sich seit JDK 8 der Einsatz der Klasse `java.util.Optional<T>` an, die einen Container für

Werte vom Typ `T` oder aber für den Wert `null` darstellt. Dadurch können optionale Werte klar ausgedrückt und `NullPointerException`s leichter vermieden werden.

6.5.1 Grundlagen zur Klasse `Optional`

Schauen wir uns zum Einstieg eine Suche nach einem Kunden über den Namen an. Eine sehr gebräuchliche Variante liefert einen Referenz auf eine `Customer`-Instanz zurück, sofern ein passender Kunde gefunden wurde. Solche Methoden besitzen Implementierungen ähnlich zu folgender:

```
public Customer findCustomerByNameOldStyle(final String name)
{
    for (final Customer customer : customers)
    {
        if (customer.getName().equals(name))
        {
            return customer;
        }
     }

     return null;
}
```

Diese Realisierung ist gut verständlich. Leicht problematisch wird sie, wenn kein passender Datensatz existiert. Dies wird vielfach – wie auch hier – über die Rückgabe des Werts `null` ausgedrückt. Allerdings lässt sich allein anhand der Signatur nicht feststellen, in welcher Form ein nicht gefundenes Element an einen Aufrufer kommuniziert wird. Demzufolge ist ein entsprechender Javadoc-Kommentar mit dem Hinweis auf eine mögliche Rückgabe von `null` sehr empfehlenswert.

Erzeugung eines `Optional<T>`

Durch Einsatz der Klasse `Optional<T>` als Rückgabe der gerade implementierten Suchmethode wird das API sofort verständlich und verdeutlicht, dass mitunter kein Treffer geliefert werden kann. Dazu müssen wir nun aber eine `Customer`-Instanz bzw. einen `null`-Wert in ein `Optional<T>` wandeln. Das geschieht durch Aufruf der Methode `of()` bzw. `ofNullable()`. Letztere nutzt man, falls ein umzuwandelnder Wert `null` sein kann. Ist ein solcher sicher `null`, bietet sich hier der Aufruf von `Optional.empty()` an:

```
public Optional<Customer> findCustomerByNameNewStyle(final String name)
{
    for (final Customer customer : customers)
    {
        if (customer.getName().equals(name))
        {
             return Optional.of(customer);
        }
   }

   return Optional.empty();
}
```

Durch die geänderte Rückgabe ist man als Aufrufer durch das Typsystem dazu gezwungen, mit dem Typ `Optional<T>` zu arbeiten bzw. genauer: Man muss erst durch verschiedene Aktionen sicherstellen, dass ein Wert vorhanden ist. Das schauen wir uns nun genauer an.

Verarbeitung von Ergebnissen

Nehmen wir an, es wäre eine Liste mit Kunden, unter anderem mit einem namens Tim, definiert.

In der nutzenden Applikation soll zunächst nur für erfolgreiche Suchen eine Aktion ausgeführt werden. Das lässt sich mit `ifPresent(Consumer<? super T>)` ausdrücken:

```
findCustomerByNameNewStyle("Tim").ifPresent(System.out::println);
findCustomerByNameNewStyle("UNKNOWN").ifPresent(System.out::println);
```

Zuerst wird nach einem Kunden mit dem Namen Tim gesucht, um die Ausführung im Positivfall mit `ifPresent(Consumer<? super T>)` zu zeigen. Die anschließende Suche nach `UNKNOWN` verläuft erfolglos, sodass man den Rückgabewert von `Optional.empty()` als Ergebnis erhält. Deswegen wird im zweiten Fall die Aktion nicht ausgeführt. Daher kommt es erwartungsgemäß zu folgenden Ausgaben:

```
Tim
```

Im obigen Beispiel wird deutlich, dass für die Suche nach dem nicht vorhandenen Wert `UNKNOWN` keine Warnmeldung erfolgt. Oftmals soll aber auch für den Fall, dass eine Suche erfolglos war, eine Aktion ausgeführt werden. Das lässt sich mit JDK 8 leider nicht mehr so elegant formulieren. Vielmehr muss man die Fallunterscheidung selbst programmieren und zudem im Positivfall den Ergebniswert per `get()` aus dem `Optional<T>` wie folgt ermitteln:

```
final Optional<String> optCustomer1 = findCustomerByNameNewStyle("Tim");
if (optCustomer1.isPresent())
{
    System.out.println("found: " + optCustomer1.get());
}
else
{
    System.out.println("not found");
}

final Optional<String> optCustomer2 = findCustomerByNameNewStyle("UNKNOWN");
if (optCustomer2.isPresent())
{
    System.out.println("found: " + optCustomer2.get());
}
else
{
    System.out.println("not found");
}
```

Mit dieser (schon recht aufwendigen) Modifikation wird der Eintrag `UNKNOWN` zwar wiederum nicht gefunden. Jedoch kommt es diesmal zu einer Warnmeldung:

```
found: Tim
not found
```

Die Methode `ifPresentOrElse()` Mit Java 9 erlaubt es uns die Methode `ifPresentOrElse()`, das Ganze deutlich kompakter und prägnanter zu schreiben – neben der Angabe zweier Aktionen profitiert man insbesondere davon, dass im Positivfall direkt der Wert zugreifbar ist:

```
final Optional<String> optCustomer1 = findCustomerByNameNewStyle("Tim");
optCustomer1.ifPresentOrElse(str -> System.out.println("found: " + str),
                             () -> System.out.println("not found"));

final Optional<String> optCustomer2 = findCustomerByNameNewStyle("UNKNOWN");
optCustomer2.ifPresentOrElse(str -> System.out.println("found: " + str),
                             () -> System.out.println("not found"));
```

Wiederum kommt es zu den folgenden Ausgaben:

```
found: Tim
not found
```

Verarbeitung von primitiven Werten

Ergänzend zu der generischen Klasse `Optional<T>` gibt es im JDK auf die primitiven Typen `int`, `long` und `double` spezialisierte Implementierungen. Das sind die Klassen `OptionalInt`, `OptionalLong` und `OptionalDouble`.

Das folgende Beispiel zeigt Berechnungen zur Ermittlung von Maximum und von Minimum sowie des Durchschnitts für ein Array des primitiven Typs `int`, das in einen `IntStream` gewandelt wird – Streams lernen wir im Detail in Kapitel 7 kennen. Hier genügt das Wissen, dass damit Folgen von Werten bereitgestellt werden.

```
public static void main(final String[] args) {

    final int[] sampleValues = {1, 3, 5, 7, 11, 13, 17};

    final OptionalInt min = IntStream.of(sampleValues).min();
    final OptionalInt max = IntStream.of(sampleValues).max();
    final OptionalDouble avg = IntStream.of(sampleValues).average();

    System.out.println("min: " + min);
    System.out.println("max: " + max);
    System.out.println("avg: " + avg);
}
```

Listing 6.37 *Ausführbar als* **'OPTIONALPRIMITIVESEXAMPLE'**

Für `min()` und `max()` werden die Zahlen gemäß ihrer natürlichen Ordnung (vgl. Abschnitt 6.1.8) sortiert. Zudem sieht man, dass die Durchschnittsberechnung mit

average() keinen int, sondern einen double liefert, weil Durchschnittswerte oftmals keine Ganzzahlen sind. Aufgrund seiner Optionalität wird der Durchschnittswert in Form eines OptionalDouble-Objekts zurückgegeben:

```
min: OptionalInt[1]
max: OptionalInt[17]
avg: OptionalDouble[9.5]
```

Weitere Behandlung von Alternativen

Häufig möchte man bei Existenz eines optionalen Werts eine Aktion oder alternativ bei Nichtexistenz eine Behandlung von Alternativen ausführen. Dazu werden die Methoden orElse(), orElseGet() und orElseThrow() genutzt. Mit der Methode orElse() kann man einen Fallback-Wert vorgeben, mit orElseGet() eine Berechnung durchführen und mit orElseThrow() eine Exception auslösen:

```
public static void main(final String[] args)
{
    // Minimum für leeren Eingabe-Stream ermitteln
    final OptionalInt min = IntStream.empty().min();

    // Alternativen Wert liefern, wenn nicht vorhanden
    System.out.println(min.orElse(-1));

    // Berechne Ersatzwert, wenn nicht vorhanden
    final IntSupplier randomGenerator = () -> (int)(100 * Math.random());
    System.out.println(min.orElseGet(randomGenerator));

    // Löse eine Exception aus, wenn nicht vorhanden
    min.orElseThrow(() -> new NoSuchElementException("there is no minimum"));
}
```

Listing 6.38 *Ausführbar als* **'OPTIONALALTERNATIVESEXAMPLE'**

An diesem Beispiel erkennt man neben der Verarbeitung mit OptionalInt auch die Erleichterung bei Werten primitiver Typen. Welchen Rückgabewert sollte man hier ansonsten wählen? Bei Referenzen ist null noch einigermaßen sprechend für kein Ergebnis, aber wie drückt man das für primitive Typen aus? Oft sieht man Werte wie -1, 0 oder MIN_VALUE. Doch keiner davon ist so sprechend wie der Einsatz von OptionalInt.empty().

Startet man das Programm OPTIONALALTERNATIVESEXAMPLE, so kommt es (durch die Zufallszahlenberechnung) in etwa zu folgenden, hier gekürzten Konsolenausgaben:

```
-1
73
Exception in thread "main" java.util.NoSuchElementException: there is no minimum
```

Offenbar wird der alternative Wert -1 ausgegeben sowie die alternative Berechnung einer Zufallszahl. Abschließend wird durch orElseThrow() die gezeigte Exception ausgelöst.

6.5.2 Weiterführendes Beispiel und Diskussion

Bisher haben wir das API der Klasse `Optional<T>` kennengelernt und ein erstes Gespür für die Möglichkeiten zur Verarbeitung optionaler Werte erhalten. Nun betrachten wir das Ganze mit einem stärkeren Praxisbezug. Wie schon zuvor wollen wir aus einer Liste von Kunden denjenigen mit einem bestimmten Namen ermitteln und für diesen verschiedene Detailinformationen ausgeben. Dabei werden wir uns aber mehr auf die Aufruferseite konzentrieren. Nehmen wir an, eine Applikation solle nach der Suche zu dem gefundenen Kunden diverse Detailinformationen ausgeben, natürlich nur, sofern ein solcher Kunde existiert.

Umsetzung mit JDK 7 oder früher

Betrachten wir eine Implementierung, wie sie mit JDK 7 typisch ist. Lassen Sie uns eine unbekannte Person wie folgt suchen und danach die Detailinformationen ausgeben:

```
final String desiredName = "Unknown";

// findCustomerByNameOldStyle() kann null liefern
final Customer customer = findCustomerByNameOldStyle(desiredName);

printCustomerDetails(customer); // potenziell NullPointerException
```

Die Suche haben wir schon als `findCustomerByNameOldStyle()` kennengelernt. Nehmen wir nun für die Methode `printCustomerDetails()` an, sie greife zur Ausgabe auf verschiedene Attribute analog zu folgendem Ausschnitt zu:

```
void printCustomerDetails(final Customer customer)
{
    final String hometown = customer.getHometown();
    // ...
}
```

Die gewählte Implementierung erfüllt ihre Aufgabe, jedoch erfordert sie beim Aufrufer zusätzliche Prüfungen für den Fall, dass das gesuchte Element nicht in der Liste enthalten ist. Wenn die Suche nämlich den Wert `null` zurückliefert, so wird bei Aufrufen der korrespondierenden `get()`-Methoden in `printCustomerDetails(Customer)` eine `NullPointerException` ausgelöst. Oftmals findet man dann als vermeintliche Abhilfe eine `null`-Prüfung wie folgt:

```
final String desiredName = "Unknown";

final Customer customer = findCustomerByNameOldStyle(desiredName);
if (customer != null)
{
    printCustomerDetails(customer);
}
```

Zwar erhalten wir nun keine `NullPointerException` mehr, allerdings auch keinen Hinweis darauf, dass der gesuchte Kunde nicht gefunden wurde. Auch vom Design her

gibt es eine Schwachstelle: Wir können als Methodenbereitsteller die späteren Nutzer nicht dazu bewegen, mögliche null-Werte geeignet abzufragen.

Lassen Sie uns kurz weiter überlegen. Im Allgemeinen ist eine Rückgabe von null vieldeutig: Heißt es »nicht gefunden« oder »es ist ein Fehler aufgetreten« oder aber etwas ganz anderes? Im obigen Beispiel wird der erste Fall ausgedrückt und wir könnten folgende Spezialbehandlung ergänzen:

```
final String desiredName = "Unknown";

final Customer customer = findCustomerByNameOldStyle(desiredName);
if (customer != null)
{
    printCustomerDetails(customer);
}
else
{
    showWarnMessage("No customer with name '" + desiredName + "'");
}
```

Beim Betrachten der Methode selbst wie auch des aufrufenden Sourcecodes kann man sich berechtigterweise fragen, ob sich das Ganze nicht schöner realisieren lässt. Wir wissen bereits, dass man seit JDK 8 dazu die Klasse Optional<T> nutzen kann.

Vorteile durch den Einsatz der Klasse Optional<T>

Weil die Methode findCustomerByNameNewStyle() ein Optional<Customer> zurückgibt, ändert sich der Sourcecode an der Aufrufstelle wie folgt:

```
final String desiredName = "Unknown";

final Optional<Customer> optCustomer = findCustomerByNameNewStyle(desiredName);
optCustomer.ifPresentOrElse(customer -> printCustomerDetails(customer),
            () -> showWarnMessage("No customer with name '" + desiredName + "'")
                 );
```

Der entscheidende Punkt wird dann deutlich, wenn man sich den Sourcecode nochmals genauer anschaut und ein wenig überlegt:

- Die Methode findCustomerByNameNewStyle() drückt durch Rückgabe von Optional<Customer> klar aus, dass ein optionaler Wert zurückgeliefert wird.
- Die Methode printCustomerDetails(Customer) erwartet ein Objekt vom Typ Customer. Wenn man im Sourcecode konsequent Optional<T> nutzt, ist offensichtlich, dass hier niemals der Wert null übergeben werden darf. Auch ist es nicht möglich, versehentlich ein Optional<T> zu übergeben, wie man es als Ergebnis einer Suche erhält.

Gerade der letzte Punkt zeigt, dass sich der konsequente Einsatz von Optional<T> positiv auswirken kann und unerwartete Übergaben von null recht unwahrscheinlich macht. Das bedeutet auch, dass nach der Einführung von Optional<T> der Wert null nur noch selten verwendet werden sollte.

6.5.3 Verkettete Methodenaufrufe

Schauen wir uns noch eine Variante zur Verhinderung von `NullPointerException`s an. Oftmals würden wir zwar gerne die Klasse `Optional<T>` für Rückgaben nutzen, aber wir haben entweder den Sourcecode nicht im Zugriff oder dürfen diesen nicht ändern.

In der Praxis findet man immer wieder einmal Aufrufketten wie diese:

```
final String version = computer.getGraphicscard().getFirmware().getVersion();
```

Das ist in der Regel schlechtes Design: Neben einem Verstoß gegen das Law of Demeter (vgl. Abschnitt 3.5.2) ist diese Variante auch fehleranfällig: Wenn eine der obigen Methoden `null` als Rückgabe liefert, da damit optionale Daten ausgedrückt werden, so wird eine `NullPointerException` ausgelöst.

Lösungsvariante 1

Oftmals erfolgt dann ein Workaround mit diversen `if`-Prüfungen und tiefen Verschachtelungen:

```
String version = "UNKNOWN";
if (computer != null)
{
    final Graphicscard graphicscard = computer.getGraphicscard();
    if (graphicscard != null)
    {
        final Firmware firmware = graphicscard.getFirmware();
        if (firmware != null)
        {
            version = firmware.getVersion();
        }
   }
}
```

Obwohl das die `NullPointerException`s verhindert, ist das Ganze doch schlecht lesbar und irgendwie nicht wirklich schön.

Lösungsvariante 2

Seit Java 8 ist man versucht, statt eines `null`-Werts mit der Klasse `Optional<T>` zu arbeiten. Unter Einsatz von Methodenreferenzen könnte man dann folgende, kürzere, aber eher nicht direkt intuitiv verständliche Konstruktion nutzen:

```
final String version = computer.flatMap(Computer::getGraphicscard).
                                flatMap(Graphicscard::getFirmware).
                                map(Firmware::getVersion).orElse("UNKNOWN");
```

Genau wie Variante 1 ist auch diese Umsetzung etwas kompliziert und unschön. Darüber hinaus ist sie für viele Entwickler vermutlich sogar schlechter verständlich.

Lösungsvariante 3

Wir können aber das Beste aus beiden Welten (Kürze, Lesbarkeit und Verständlichkeit) erreichen, wenn wir uns eine Hilfsmethode `safeResolve()` basierend auf dem Interface `Supplier<T>` schreiben. Dabei habe ich mich von einem Blog von Benjamin Winterberg inspirieren lassen.[23]

Schauen wir uns zunächst die Verwendung und dann die Realisierung der Hilfsmethode an. Die Verwendung erfolgt nahezu so, wie man sie auch ursprünglich aufgerufen hätte, nur als Lambda und als Parameter der Methode `safeResolve()`:

```
final Optional<T> optValue = safeResolve(
                  () -> computer.getGraphicscard().getFirmware().getVersion());

optValue.ifPresent(System.out::println);
```

Praktischerweise können wir weiterhin die intuitiv verständliche `.`-Notation nutzen, ummanteln diese aber mit einem Aufruf an unsere Hilfsmethode, die beim Auftreten einer `NullPointerException` diese in ein `Optional.empty()` umwandelt:

```
public static <T> Optional<T> safeResolve(final Supplier<T> resolver)
{
    try
    {
        final T result = resolver.get();
        return Optional.ofNullable(result);
    }
    catch (final NullPointerException npe)
    {
        return Optional.empty();
    }
}
```

6.5.4 Erweiterungen in der Klasse `Optional<T>` in JDK 9

Neben der bereits genutzten Erweiterung in Form der Methode `ifPresentOrElse(Consumer<? super T>, Runnable)`, die eine Ausführung einer Aktion im Positiv- oder im Negativfall ermöglicht, finden wir in Java 9 folgende Ergänzungen in `Optional<T>`:

- `stream()` – Wandelt das `Optional<T>` in einen `Stream<T>` um.
- `or(Supplier<Optional<T>> supplier)` – Ermöglicht auf elegante Weise die Verknüpfung mehrerer Berechnungen.

[23] http://winterbe.com/posts/2015/03/15/avoid-null-checks-in-java

Die Methode `stream()`

Manchmal ist es praktisch, ein `Optional<T>` in einen `Stream<T>` umzuwandeln. Das wird nun durch die Methode `stream()` möglich.

Deren Nutzen lässt sich gut für einen Stream von optionalen Werten verdeutlichen, falls dort nur diejenigen Einträge mit gültigen Werten behalten werden sollen. Das erreicht man durch die Kombination der Methoden `flatMap()` und `stream()`. Hierbei wird jedes `Optional<T>` in einen Stream überführt und durch `flatMap()` zu einem einzigen Stream mit Werten kombiniert. Das Verfahren zeige ich am Beispiel eines Streams, der aus `Optional<String>`-Elementen besteht, beispielsweise als Folge einer parallelen Suche. Am Ende sollen die Ergebnisse konsolidiert werden. Diese Anforderung realisieren wir wie folgt:

```
public static void main(final String[] args)
{
    final Stream<Optional<String>> streamOfOptionalNames = Stream.of(
                                   Optional.of("Tim"), Optional.of("Tom"),
                                   Optional.empty(), Optional.of("Mike"),
                                   Optional.empty(), Optional.of("Andy"));

    final Stream<String> streamOfNames =
                         streamOfOptionalNames.flatMap(Optional::stream);

    streamOfNames.forEach(value -> System.out.println("found: " + value));
}
```

Listing 6.39 *Ausführbar als* **'OPTIONALSTREAMEXAMPLE'**

Das Programm OPTIONALSTREAMEXAMPLE produziert folgende Ausgaben:

```
found: Tim
found: Tom
found: Mike
found: Andy
```

Man erkennt, dass alle leeren Elemente (`Optional.empty()`) entfernt und die Werte aus den `Optional<String>`-Instanzen extrahiert wurden. Als Ergebnis entsteht ein `Stream<String>`. Das geschieht, weil die Methode `flatMap()` ineinander verschachtelte Streams zu einem flachen Stream zusammenfasst. Wenn man `Optional::stream` nutzt, wird aus einem `Optional.empty()` ein leerer Stream. Der Aufruf von `flatMap()` entfernt diesen dann automatisch.

Die Methode or()

Nach den beiden grundlegenden Erweiterungen schauen wir abschließend auf die so unscheinbar wirkende Methode `or(Supplier<? extends Optional<? extends T>>)`. Mit deren Hilfe lassen sich Methoden bzw. Aufrufketten mit Fallback-Strategien auf lesbare und verständliche Art beschreiben, wie es die Methode `multiFindCustomer(String)` eindrucksvoll zeigt:[24]

```
public static void main(final String[] args)
{
    final Optional<String> optCustomer = multiFindCustomer("Tim");
    optCustomer.ifPresentOrElse(str -> System.out.println("found: " + str),
                                () -> System.out.println("not found"));
}

private static Optional<String> multiFindCustomer(final String customerId)
{
    return findInCache(customerId)
           .or(() -> findInMemory(customerId))
           .or(() -> findInDb(customerId));
}
```

Listing 6.40 *Ausführbar als* '**OPTIONALORE**XAMPLE'

Zwei der aufgerufenen Suchmethoden sind bewusst sehr simpel und kurz realisiert und liefern lediglich `Optional.empty()` zurück. Damit ergibt sich folgende Realisierung für `findInCache(String)` und `findInDb(String)`:

```
private static Optional<String> findInCache(final String customerId)
{
    System.out.println("findInCache");
    return Optional.empty();
}

private static Optional<String> findInDb(final String customerId)
{
    System.out.println("findInDb");
    return Optional.empty();
}
```

In der Methode `multiFindCustomer(String)` werden nacheinander drei Abfragen ausgeführt, sofern nicht zuvor ein Treffer gefunden wird. Das ist hier für den zweiten Aufruf `findInMemory(String)` der Fall, weil diese Methode eine Suche in einem `Stream<String>` wie folgt ausführt:

```
private static Optional<String> findInMemory(final String customerId)
{
    System.out.println("findInMemory");
    final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");

    return customers.filter(name -> name.contains(customerId))
                    .findFirst();
}
```

[24] Bei der Beschreibung für die Methode `or()` habe ich mich von einem Beispiel des Blogs `http://blog.codefx.org/java/dev/java-9-optional/` inspirieren lassen.

Startet man das Programm OPTIONALOREXAMPLE, so kommt es zu folgenden Ausgaben, die die Arbeitsweise verdeutlichen:

```
findInCache
findInMemory
Tim
```

Zunächst wird nach Tim im Cache durch `findInCache(String)` gesucht, jedoch ohne Erfolg. Das zurückgelieferte `Optional.EMPTY` führt automatisch dazu, dass die zweite Methode in der Aufrufkette, also hier `findInMemory(String)`, ausgeführt wird. Weil die Suche nach Tim dort erfolgreich ist, bricht die Verarbeitung ab und es wird das Suchergebnis als `Optional<String>` zurückgegeben.

6.5.5 Erweiterung in `Optional<T>` in JDK 10 und 11

Die Klasse `java.util.Optional<T>` war eine enorme Bereicherung in Java 8. Mit Java 9 wurde das API durch die drei Methoden `ifPresentOrElse()`, `or()` und `stream()` vervollständigt. Damit schien das API recht komplett.

Erweiterung in JDK 10

Allerdings hat man sich bei Oracle an einer Unschönheit gestört: an der Methode `get()`. Wieso? Die Antwort ist einfach: Die Methode `get()` zum Zugriff auf den Wert eines `Optional<T>` sieht zu harmlos aus. Vermutlich dadurch findet man in der Praxis mitunter einen Aufruf von `get()` ohne vorherige Prüfung auf Existenz eines Werts mit `isPresent()`. Das führt dann aber bei einem nicht vorhandenen Wert zu einer `java.util.NoSuchElementException`. Normalerweise erwartet man beim Aufruf einer `get()`-Methode allerdings nicht unbedingt, dass diese eine Exception auslöst. Um diesen Sachverhalt im API direkt ausdrücken zu können, wurde die Methode `orElseThrow()` als Alternative zu `get()` in das JDK aufgenommen, die bei Vorhandensein den Wert liefert und ansonsten eben eine `NoSuchElementException` auslöst:

```
Optional<String> optValue = Optional.of("ABC");
String value = optValue.orElseThrow(); // => value = "ABC"

Optional<String> optEmpty = Optional.empty();
optEmpty.orElseThrow();  // => java.util.NoSuchElementException thrown: No value
       present
```

Erweiterung in der Klasse `Optional<T>` in JDK 11

Java 11 bringt mit `isEmpty()` noch eine weitere Methode, um für mehr Konsistenz zu anderen APIs wie denjenigen von `Collection` und `String` zu sorgen. Die Methode `isEmpty()` liefert ihrem Namen gemäß `true`, falls kein Wert im `Optional<T>` enthalten ist, und ansonsten `false`.

Die Verbesserung in der Lesbarkeit des Sourcecodes durch Verzicht der Negation verdeutlicht folgendes Beispiel:

```
final Optional<String> optEmpty = Optional.empty();

if (!optEmpty.isPresent())
    System.out.println("check for empty JDK 10 style");

if (optEmpty.isEmpty())
    System.out.println("check for empty JDK 11 style");
```

Hinweis: Anmerkung zur Sinnhaftigkeit und Konsistenz

Ob die Methode `orElseThrow()` wirklich benötigt wird, kann man infrage stellen. Zum einen ist es auch beim `java.util.Iterator<T>` so, dass ein Aufruf von `next()` ohne vorherigen Aufruf von `hasNext()` eine Exception auslöst. Zum anderen hätte der Name besser `getOrElseThrow()` lauten sollen, um Verwechslungen mit `orElse()` oder `orElseGet()` zu vermeiden.

Erneut kann man hinterfragen, ob die Methode `isEmpty()` wirklich benötigt wird. Tatsächlich bringt sie ein wenig mehr Lesbarkeit. Jedoch sollte man überlegen, ob das Gegenteil von `isPresent()` wirklich `isEmpty()` ist – vermutlich wäre `isAbsent()` klarer, aber nicht mehr in Analogie zu den Containern des Collections-Frameworks.

An dieser kurzen Darstellung sieht man schon, wie schwierig es ist, ein API rundum gelungen zu gestalten.

6.6 Fallstricke im Collections-Framework

Beim Collections-Framework handelt es sich um eine gelungene und umfangreiche Sammlung gebräuchlicher Algorithmen und Datenstrukturen. Allerdings existieren doch kleinere Fallstricke, die man kennen sollte, um nicht über sie zu stolpern.

6.6.1 Wissenswertes zu Arrays

Arrays und Vergleiche

Die für verschiedene primitive Typen und den Typ `Object[]` überladene Methode `Arrays.equals()` verhält sich für verschachtelte Arrays nicht so, wie man es intuitiv erwarten würde. Folgendes Listing zeigt das Problem: Es werden zwei Arrays verglichen, die jeweils Arrays mit inhaltlich identischen Stringliteralen speichern. In diesem Beispiel werden zur Verdeutlichung die Indexpositionen der Arrays als Wert gespeichert:

```
public static void main(final String[] args)
{
    final String[][] array1 = { { "0.0", "0.1" }, { "1.0", "1.1" } };
    final String[][] array2 = { { "0.0", "0.1" }, { "1.0", "1.1" } };

    final boolean arrayEquals = Arrays.equals(array1, array2);
    final boolean deepEquals = Arrays.deepEquals(array1, array2);

    System.out.println("equals = " + arrayEquals);      // false !!!
    System.out.println("deepEquals = " + deepEquals);   // true
}
```

Listing 6.41 *Ausführbar als* '**ARRAYCOMPAREEXAMPLE**'

Ganz intuitiv erwartet man bei einem Vergleich von Arrays mit offensichtlich gleichem Inhalt auch das Feststellen von Gleichheit. Tatsächlich werden die obigen Arrays als unterschiedlich erkannt. Dies ist dadurch bedingt, dass die Implementierung von `Arrays.equals(Object[], Object[])` die enthaltenen Subarrays nicht rekursiv vergleicht. Das wurde aus Performance-Gründen so realisiert. ***Zum korrekten Vergleich verschachtelter Arrays ist daher immer die Methode `Arrays.deepEquals(Object[], Object[])` zu verwenden***.[25]

Ähnliche Fallstricke lauern bei der Berechnung des Hashwerts über `Arrays.hashCode(Object[])` und bei der Erzeugung einer Stringrepräsentation mit `Arrays.toString(Object[])`. Es existieren in der Utility-Klasse `Arrays` korrespondierende Methoden `deepHashCode(Object[])` und `deepToString(Object[])`.

Arrays und `clone()`

Arrays bieten eine öffentliche `clone()`-Methode, die eine flache Kopie eines Arrays erzeugt: Das bedeutet, dass jedoch lediglich ein neues Array mit der Kopie der ersten Ebene der Elemente erzeugt wird. Die Subarrays werden zwischen den Arrays »geteilt«. Betrachten wir dazu ein Beispiel:

```
public final class ArrayCloneExample
{
    private static final String[][] original = { { "0.0", "0.1" },
                                                 { "1.0", "1.1" } };

    // Kopie des Arrays erzeugen
    private static final String[][] clone = original.clone();

    public static void main(final String[] args)
    {
        // Protokolliere Ausgangszustand
        System.out.println("before modification:");
        System.out.println("original: " + Arrays.deepToString(original));
        System.out.println("clone:    " + Arrays.deepToString(clone));
```

[25] Meiner Ansicht nach ist das wenig intuitiv und mitunter sogar problematisch. Joshua Bloch wollte in einer persönlichen Diskussion auf der Java One 2009 nicht ganz so weit gehen. Allerdings war ihm die Problematik bereits beim Entwurf des Collections-Frameworks bewusst und daher existiert `Arrays.deepEquals(Object[], Object[])`.

```
        // Verändere Einträge im Clone
        System.out.println("after modification:");
        clone[0][1] = "New 0.1";
        clone[1] = new String[] { "New Sub-Array 1" };

        System.out.println("original: " + Arrays.deepToString(original));
        System.out.println("clone:    " + Arrays.deepToString(clone));
    }
}
```

Listing 6.42 *Ausführbar als* '**ARRAYCLONEEXAMPLE**'

Es kommt zu folgender Ausgabe – hier muss jedoch die Methode `Arrays.deepToString(String[])` verwendet werden, da es sich um verschachtelte `String`-Arrays handelt:

```
before modification:
original: [[0.0, 0.1], [1.0, 1.1]]
clone:    [[0.0, 0.1], [1.0, 1.1]]
after modification:
original: [[0.0, New 0.1], [1.0, 1.1]]
clone:    [[0.0, New 0.1], [New Sub-Array 1]]
```

Das Klonen führt zu einer Kopie der ersten Array-Ebene. In diesem Beispiel sind demnach die Arrays `original` und `clone` verschieden, aber die Subarrays sind gleich. Dadurch kann man mit einer Zuweisung an `clone[1]` unabhängig einen anderen Wert setzen. Ein Schreibzugriff auf die zweite Ebene `clone[0][1]` verändert allerdings beide Arrays. Abbildung 6-11 zeigt dies.

Abbildung 6-11 *Arbeitsweise der Methode* `clone()` *für Arrays*

Rückgabe von Arrays und die Gefahr von Inkonsistenzen

Zur Bereitstellung von Berechnungsergebnissen oder einer Menge vordefinierter Werte setzt man mitunter Arrays ein, ohne sich darüber allzu viele Gedanken zu machen. Allerdings besteht bei der Rückgabe von Arrays auf interne Daten die Gefahr von Inkonsistenzen, weil gespeicherte Werte unerwartet verändert werden können.

Folgendes Beispiel macht dies deutlich und zeigt mögliche Probleme beim Zugriff auf ein Array `CAPITAL_CITIES` vom Typ `String[]`. Es sind folgende zwei Zugriffsmethoden `getCities()` und `getCityIterator()` definiert:

```
public final class BesserIteratorAlsArray
{
    private static final String[] CAPITAL_CITIES = new String[]
                                { "Berlin", "London", "Paris", "Wien" };

    public static final String[] getCities()
    {
        return CAPITAL_CITIES;
    }

    public static final Iterator<String> getCityIterator()
    {
        return Arrays.asList(CAPITAL_CITIES).iterator();
    }

    public static void main(final String[] args)
    {
        System.out.println("CITIES " + Arrays.toString(CAPITAL_CITIES));

        final String[] cities = getCities();
        // unerwartete Modifikation (auch im Original-Array!)
        cities[1] = "London has changed!";

        System.out.println("CITIES " + Arrays.toString(cities));
        System.out.println("CITIES " + Arrays.toString(CAPITAL_CITIES));

        // keine Modifikation möglich
        final Iterator<String> cityIterator = getCityIterator();
        while (cityIterator.hasNext())
            System.out.println(cityIterator.next());
    }
}
```

Listing 6.43 *Ausführbar als* '**BESSERITERATORALSARRAY**'

In diesem Beispiel wird die Ausgabe der Städte auf zwei unterschiedliche Arten realisiert: Beim Zugriff über die Methode `getCities()` wird zwar nur eine Referenz auf ein `String[]` zurückgeliefert, allerdings werden unerwartet Modifikationen im `private static final` definierten Array `CAPITAL_CITIES` möglich. Die Begründung dafür ist einfach: Zwar ist die Referenz auf das Array `final` und damit unveränderlich, jedoch gilt das nicht für die dort gespeicherten Referenzen. Das spiegelt sich in folgender gekürzter Konsolenausgabe wider:

```
CITIES [Berlin, London, Paris, Wien]
CITIES [Berlin, London has changed!, Paris, Wien]
[...]
```

Der Eintrag "`London`" wird unerwartet zu "`London has changed!`". Demnach gilt: ***Die Rückgabe von Referenzen auf Arrays birgt die Gefahr von Modifikationen und verschlechtert die Datenkapselung***.

Iterator als Abhilfe Die zweite Variante nutzt einen Iterator zum Zugriff. Das führt zu einer besseren Kapselung, verhindert automatisch Modifikationen sowohl der Zusammensetzung als auch der Daten und löst damit das Problem ungewünschter Änderungen im Array: Allerdings gilt dies nur in diesem Fall für unveränderliche Stringobjekte. Für andere gespeicherte Objektreferenzen erreicht man so jedoch lediglich einen Schutz ähnlich zu dem der `unmodifiable`-Wrapper: Die Zusammensetzung des Arrays lässt sich nicht ändern, aber die Daten der gespeicherten Objekte möglicherweise schon.

6.6.2 Wissenswertes zu `Stack`, `Queue` und `Deque`

Bislang wurden die Klassen `Stack<E>`, `Queue<E>` und `Deque<E>` nur am Rande erwähnt. Nachfolgend gehe ich auf deren Arbeitsweise und auf einige Fallstricke ein.

Die Arbeitsweise der Klasse `Stack<E>` (zu deutsch: Stapel) kann man sich wie einen Stapel Papier in einer Schreibtischablage vorstellen. Neue Aufträge werden oben abgelegt und der oberste Auftrag wird als Nächstes bearbeitet. Diese Bearbeitungsreihenfolge ist in der Informatik unter dem Begriff ***LIFO*** (***Last-In-First-Out***) bekannt.

Die Klasse `Queue<E>` repräsentiert eine Warteschlange. Man kann sich die Verarbeitung wie beim Anstellen an der Kasse im Supermarkt vorstellen. Neue Wartende reihen sich immer hinten in die Schlange der Wartenden ein und jeder kommt der Reihe nach dran. Diese Art der Bearbeitung wird in der Informatik ***FIFO*** (***First-In-First-Out***) oder auch ***FCFS*** (***First-Come-First-Serve***) genannt. Tatsächlich kennt man es aus der Realität, dass sich immer mal wieder Personen vordrängeln oder aber von anderen vorgelassen werden. Diesen Sachverhalt kann man durch sogenannte ***Prioritätswarteschlangen*** abbilden. Sie werden durch das Interface `PriorityQueue<E>` realisiert. Dort übersteuern Prioritäten die Bedienreihenfolge. Die Arbeitsweise der Datenstrukturen wird in Abbildung 6-12 angedeutet.

Abbildung 6-12 *Arbeitsweise von Stacks und Queues*

Fallstricke in der Klasse `Stack`

Die Klasse `Stack<E>` wird durch eine Implementierungsvererbung von der Klasse `Vector<E>` realisiert. Sie bietet daher alle Methoden ihrer Basisklasse `Vector<E>`. Darüber hinaus werden folgende fünf stackspezifische Methoden bereitgestellt:

- `E push(E element)` – Legt ein Element oben auf den Stack und gibt es zurück.
- `E peek()` – Liefert das oberste Element, *ohne* es aus dem Stack zu entfernen. Falls keine Elemente existieren, kommt es zu einer `java.util.EmptyStackException`.
- `E pop()` – Gibt das oberste Element zurück und *entfernt* es aus dem Stack. Wirft eine `EmptyStackException`, falls keine Elemente vorhanden sind.
- `boolean empty()` – Prüft, ob der Stack leer ist.
- `int search(Object object)` – Sucht nach dem übergebenen Element und gibt dessen Position zurück. Diese Funktionalität ist nützlich, um herauszufinden, wie weit ein Element noch von der obersten Position entfernt ist. Die Suche liefert eine Position, die 1-basiert ist: Für das Element ganz oben auf dem Stack gibt die Methode den Wert 1 zurück.

Durch die Implementierungsvererbung von `Vector<E>` sind für die Klasse `Stack<E>` leider auch Methodenaufrufe möglich, die der Arbeitsweise eines Stacks widersprechen. Es können z. B. die Methoden `get(int)`, `indexOf(Object)` oder sogar verändernde Zugriffe aufgerufen werden, etwa `add(int, E)` oder `remove(int)`. Weiterhin existieren mehrere Methoden mit leicht unterschiedlichen Namen, die dasselbe tun, beispielsweise `empty()` und `isEmpty()`. Diese Beispiele sollen Sie für die Problematik der Implementierungsvererbung sensibilisieren.

Deutlich erkennbar werden die resultierenden Fallstricke, wenn man sowohl die Methoden der Basisklasse `Vector<E>` als auch diejenigen der Klasse `Stack<E>` benutzt. Nehmen wir an, es würden einige Elemente statt durch den Aufruf von `push(E)` durch Aufruf der Methode `add(E)` hinzugefügt. Intuitiv erwartet man ein Anfügen an letzter Position. Die in Abbildung 6-12 dargestellten Positionen stimmen jedoch nicht mit den Indexwerten im `Vector<E>` überein. Tatsächlich entspricht das oberste Element eines Stacks dem letzten Element der zugrunde liegenden Basisklasse `Vector<E>`. Diese Details sind beim Einsatz der eigentlichen Schnittstelle der Klasse `Stack<E>` nicht von Interesse. Beim Aufruf von Methoden der Klasse `Vector<E>` muss man diese Details jedoch kennen, ihre Auswirkungen beachten und Vorsicht walten lassen.

Derartige Probleme beim Einsatz eines nicht intuitiven APIs erschweren das Programmieren und lenken von der zu realisierenden Aufgabe ab. Eine weitere wesentliche Erkenntnis aus der obigen Diskussion ist: ***Implementierungsvererbung stellt sehr oft keine geeignete Wahl dar, um funktionale Erweiterungen zu realisieren***. Eine korrekte Realisierung, die zudem das Substitutionsprinzip (»is-a«-Beziehung) nicht verletzt, wäre problemlos möglich gewesen: Eine objektorientierte Umsetzung verwendet eine Referenz auf die Klasse `Vector<E>` (oder einer beliebigen Liste) und nutzt Delegation,

um die zuvor genannten Methoden eines Stacks zu realisieren. Das kann man in etwa wie folgt machen:

```
public class Stack<E>
{
    // Komposition statt Vererbung, hier konkreter Typ LinkedList<E>, weil das
    // Interface List<E> nur add(int, E) und remove(int) bietet, hier aber ...
    private final LinkedList<E> elements = new LinkedList<>();

    public void push(final E item)
    {
        // ... Delegation an besser passende Methode
        element.addFirst(item);
    }

    public E pop()
    {
        if (elements.size())
            throw new EmptyStackException();

        // ... Delegation an besser passende Methode
        return elements.removeFirst()
    }
    // ...
}
```

Merkwürdigkeiten im Interface `Queue`

Zum Datenaustausch zwischen verschiedenen Programmkomponenten können Implementierungen des Interface `Queue<E>` verwendet werden. Dieses erweitert das Interface `Collection<E>` und bietet ergänzend dazu folgende Methoden[26]:

- `boolean add(E element)`,
- `boolean offer(E element)` – Beide Methoden fügen ein Element am Ende der Queue ein. Für größenbeschränkte Queues existiert zwischen beiden Methoden folgender Unterschied: Der Aufruf von `add(E)` löst teilweise – abhängig von der konkreten Realisierung des Interface `Queue<E>` – eine `IllegalStateException` aus, wenn die Queue voll ist. Ein Aufruf von `offer(E)` liefert stattdessen den Wert `false`.
- `E element()`,
- `E peek()` – Beide Methoden geben das erste Element zurück, *ohne* es aus der Queue zu entfernen. `element()` löst eine `NoSuchElementException` aus, wenn die Queue leer ist. `peek()` liefert in diesem Fall den Wert `null`.
- `E poll()`,
- `E remove()` – Beide Methoden geben das erste Element zurück und *entfernen* es aus der Queue. `remove()` löst eine `NoSuchElementException` aus, wenn die Queue leer ist. `poll()` liefert in diesem Fall den Wert `null`.

[26] Die Methode `add(E)` existiert bereits im Interface `Collection<E>`, ist hier aber aufgeführt, um jeweils Methodenpaare vorzustellen und auf eine Besonderheit hinzuweisen.

Da das Interface `Queue<E>` eine Erweiterung des Interface `Collection<E>` ist, fügt es sich nahtlos in das Collections-Framework ein. Unter anderem bedeutet das, dass sich die vordefinierten Algorithmen anwenden lassen. Wie die zuvor kritisierte Realisierung der Klasse `Stack<E>` enthält auch die Definition des Interface `Queue<E>` aufgrund der Vererbungsbeziehung zum Interface `Collection<E>` einige Methoden, die nicht zur Funktionsweise der Datenstruktur Queue passen: Dies ist beispielsweise eine Methode `remove(Object)`, die es erlaubt, beliebige Objekte zu löschen. Außerdem sind dies sogenannte Bulk-Operationen, die auf einer Menge von Elementen arbeiten, etwa `addAll()`, `containsAll()`, `removeAll()` oder `retainAll()`. Die Kritik wird dadurch verstärkt, dass die Klasse `LinkedList<E>` das Interface `Queue<E>` erfüllt. Ein schöneres, objektorientiertes Design hätte darin bestanden, eine eigenständige Klasse zu definieren, die eine Liste nutzt.

Unklarheiten im Interface `Deque`

Eine sogenannte Deque ist eine doppelseitige Warteschlange und wird durch das generische Interface `Deque<E>` beschrieben, das das zuvor vorgestellte Interface `Queue<E>` erweitert. Ergänzend zum Interface `Queue<E>` werden hier Methoden zum Hinzufügen und Löschen am Anfang und am Ende der Datenstruktur bereitgestellt. Dadurch gibt es aber mehrere Methodenpaare, die gleiche Funktionalität anbieten, aber unterschiedliche Namen tragen, etwa `add(E)` und `addLast(E)` sowie `offer(E)` und `offerLast(E)`. Die sich daraus ergebenden Probleme wurden schon in den Beschreibungen der Klasse `Stack<E>` und des Interface `Queue<E>` vorgestellt.

6.7 Weiterführende Literatur

Dieses Kapitel hat eine Einführung in Datenstrukturen und das Collections-Framework gegeben. Weitere Informationen finden Sie in folgenden Büchern:

- »**Java Generics and Collections**« von Maurice Naftalin und Philip Wadler [55]
  Dieses Werk ist die ideale Weiterführung dieses Buchs, da es besonders auf Generics und einige Erweiterungen des Collections-Frameworks eingeht.

Weiterführende Informationen und Grundlagen zu Algorithmen und Datenstrukturen können in folgenden Büchern nachgelesen werden:

- »**Data Structures and Algorithms with Object-Oriented Design Patterns in Java**« von Bruno R. Preiss [61]
  Dieses Buch zeigt, wie man Datenstrukturen in Java realisieren kann. Man erhält hier das notwendige Hintergrundwissen, wenn man Details des Collections-Frameworks verstehen möchte.

- »**Introduction to Algorithms**« von Thomas H. Cormen et al. [14]
  Ähnlich wie das zuvor genannte Buch von Bruno R. Preiss wird hier ein guter, aber eher formaler Überblick über Algorithmen und Datenstrukturen geboten.
- »**Algorithmen**« von Robert Sedgewick [65]
  Robert Sedgewick stellt fundiert verschiedene Algorithmen und Datenstrukturen vor. Das geschieht weniger formal als bei Thomas H. Cormen et al. Etliche anschauliche Abbildungen helfen dabei, die Algorithmen nachzuvollziehen.

7 Das Stream-API

Mit JDK 8 wurden Massenoperationen auf Collections (Bulk Operations on Collections) eingeführt. Neben den schon in den Abschnitten 6.1.3 und 6.1.5 beschriebenen kleineren Erweiterungen in den Interfaces `Collection<E>` und `List<E>` sind insbesondere die sogenannten Streams von Relevanz: Diese liefern die Schnittstelle zur funktionalen Programmierung.

Einführend schauen wir uns Grundlagen zu Streams in Abschnitt 7.1 an. Danach betrachten wir das Filter-Map-Reduce-Framework in Abschnitt 7.2. Zum Abschluss zeige ich in Abschnitt 7.3 einige Einsatzgebiete aus der Praxis.

7.1 Grundlagen zu Streams

Für das in JDK 8 neu eingeführte Konzept der ***Streams*** spielt das Interface `java.util.stream.Stream<T>` eine Schlüsselrolle. Streams sind eine Abstraktion für ***Folgen von Verarbeitungsschritten auf Daten***. Darüber hinaus ähneln Streams sowohl Collections als auch Iteratoren, wobei Streams keine Speicherung der Daten vornehmen und nur einmal traversiert werden können. Als weitere Analogie kann die Abarbeitung als Fließband betrachtet werden. Dabei unterscheidet man zwischen folgenden Typen von Operationen: ***Create*** (Erzeugung), ***Intermediate*** (Berechnung) und ***Terminal*** (Ergebnisermittlung). Nachfolgend ist dies schematisch dargestellt:

$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow ... \Rightarrow OP_{n-1}}_{Intermediate} \Rightarrow \underbrace{OP_n \Rightarrow Ergebnis}_{Terminal}$$

Einführendes Beispiel

Das folgende Listing zeigt die Operationen, ohne auf Details einzugehen. Hier möchte ich Ihnen zunächst einen ersten Eindruck für Streams und die Verarbeitung damit vermitteln. Dazu schauen wir uns eine Liste von Personen an, die auf alle Erwachsenen gefiltert und schließlich wieder als `List<Person>` zurückgegeben wird:

```
List<Person> adults = persons.stream().                    // Create
                              filter(Person::isAdult).      // Intermediate
                              collect(Collectors.toList()); // Terminal
```

Neben all diesen (noch unbekannten) Implementierungsneuerungen erkennt man sehr schön, dass sich Konzepte und das »Was« viel klarer ausdrücken lassen und nicht das »Wie« (die Details der Implementierung der Funktionalität) im Vordergrund steht.

7.1.1 Streams erzeugen – Create Operations

Nach dem ersten Beispiel zu Streams wollen wir unsere Kenntnisse vertiefen. Zunächst stelle ich einige Varianten zur Erzeugung von Streams vor.

Streams für Arrays und Collections

Für Arrays oder Collections gibt es eine Methode `stream()`, die ein `Stream`-Objekt erzeugt:

```
final String[] namesData = { "Karl", "Ralph", "Andi", "Andy", "Mike" };
final List<String> names = Arrays.asList(namesData);

final Stream<String> streamFromArray = Arrays.stream(namesData);
final Stream<String> streamFromList = names.stream();
```

Als Besonderheit können Collections eine sequenzielle sowie eine parallele Variante eines Streams liefern:

```
final Stream<String> sequentialStream = names.stream();
final Stream<String> parallelStream = names.parallelStream();
```

Für Arrays bietet die Utility-Klasse `java.util.Arrays` dagegen nur Zugriff auf eine sequenzielle Variante. Um hier auf Parallelverarbeitung zu wechseln, kann man die Methode `parallel()` auf dem zu parallelisierenden Stream aufrufen. Für das obige Array könnte man somit Folgendes schreiben:

```
final Stream<String> parallelArrayStream = Arrays.stream(namesData).parallel();
```

Streams für vordefinierte Wertebereiche

Teilweise soll über Streams ein fixer, vordefinierter Wertebereich abgebildet und bearbeitet werden. Dazu gibt es spezielle Methoden, etwa `of()`, `range()` und `chars()`:

```
final Stream<String> names = Stream.of("Tim", "Andy", "Mike");   // String
final Stream<Integer> integers = Stream.of(1, 4, 7, 7, 9, 7, 2); // Integer

final IntStream values = IntStream.range(0, 100);                // int
final IntStream chars = "This is a test".chars();                // int
```

Im Listing kommt neben dem generischen Interface `Stream<T>` auch das für den primitiven Datentyp `int` spezifische Interface `java.util.stream.IntStream` zum Einsatz. Die Verarbeitung erfolgt in dieser Art von Streams mit Werten primitiver Typen und nicht wie bei `Stream<Integer>` mit `Integer`-Objekten.

Besonderheit: Streams für primitive Typen

Im vorherigen Beispiel wurde angedeutet, dass es mit der Klasse `IntStream` eine auf primitive Typen ausgerichtete Variante von Streams gibt. Der Grund für deren Existenz ist, dass die Verarbeitung primitiver Werte ein recht gebräuchlicher Anwendungsfall ist und Berechnungen damit schneller als auf den korrespondierenden Wrapper-Klassen ablaufen. Daher wurde das JDK um besondere Streams erweitert, die auf die Verarbeitung der primitiven Typen `int`, `long` und `double` spezialisiert sind. Das sind die Klassen `IntStream`, `LongStream` und `DoubleStream` aus dem Package `java.util.stream`. Neben spezialisierten, etwas performanteren Berechnungen und Konvertierungen untereinander kann man die Streams auch per `boxed()` bzw. `mapToObj()` wieder in einen Stream von Wrapper-Instanzen bzw. beliebigen Objekten umwandeln. Diese Aktionen sind im nachfolgenden Listing gezeigt. Dort entsteht aus einer Liste von Strings ein korrespondierender Stream, der dann in einen Stream gewandelt wird, der die Stringlänge enthält. Dieser `IntStream` lässt sich durch Aufruf von `asLongStream()` in einen auf den Typ `long` spezialisierten `LongStream` konvertieren. Mit `boxed()` wird daraus ein `Stream<Long>` usw.:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Mike", "Stefan", "Nikolaos");
   Stream<String> values = names.stream().                  // -> Stream<String>
                           mapToInt(String::length).        // -> IntStream
                           asLongStream().                  // -> LongStream
                           boxed().                         // -> Stream<Long>
                           mapToDouble(x -> x * .75).       // -> DoubleStream
                           mapToObj(val -> "Val: " + val); // -> Stream<String>

   values.forEach(System.out::println);
}
```

Listing 7.1 *Ausführbar als* **'PRIMITIVESSTREAMEXAMPLE'**

Führen wir das Programm PRIMITIVESSTREAMEXAMPLE aus, so wird die Liste der Namen in eine Liste von Stringlängen vom Typ `int` konvertiert. Nach ein paar Umwandlungen multiplizieren wir diese Werte mit dem Faktor 0.75, um `double`-Werte zu erhalten. Schließlich nutzen wir `mapToObj()`, wodurch ein `Stream<String>` entsteht. Es kommt zu folgenden Ausgaben:

```
Val: 3.0
Val: 4.5
Val: 6.0
```

Frage: Was ist mit dem Support für die anderen primitiven Typen?

Durch die Widening genannte automatische Typkonvertierung (vgl. Abschnitt 5.2.1) sind die drei Spezialisierungen von Streams für `int`, `long` und `double` für alle anderen primitiven Zahlentypen ausreichend, weil es immer eine passende Widening-Konvertierung gibt.

7.1.2 Intermediate und Terminal Operations im Überblick

Gebräuchliche Anwendungsfälle für den Einsatz von Streams sind etwa das Filtern, das Transformieren und das Sortieren von Werten. Dazu nutzt man sogenannte ***Intermediate Operations***. Diese beschreiben ***Verarbeitungsschritte***. Praktischerweise lassen sich diese einfach hintereinanderschalten. Das Besondere daran ist, dass zunächst keine Berechnungen erfolgen, sondern lediglich die Abläufe beschrieben werden. Dabei unterscheidet man zudem zwischen ***zustandslosen*** und ***zustandsbehafteten*** Varianten. Filtern ist eine zustandslose Aktion. Damit ist gemeint, dass für jedes Element des Streams unabhängig von den anderen diese Aktion ausführbar ist. Dadurch lassen sich zustandslose Operationen auch hervorragend parallelisieren. Sortieren ist dagegen eine zustandsbehaftete Aktion, die die Kenntnis der anderen Elemente im Stream (oder zumindest eines Teils davon) erfordert. Da Streams keine (oder für zustandsbehaftete Operationen meistens nur eine Untermenge der) Daten zwischenspeichern, verbrauchen Streams im Gegensatz zu Collections in der Regel deutlich weniger Speicher. Somit hat die Konstruktion von Streams oftmals wenig Einfluss auf Speicherbedarf und Ausführungszeit.

Irgendwann sollen die ***Verarbeitungsergebnisse*** zusammengefasst, auf der Konsole ausgegeben oder anderweitig verarbeitet werden. Dazu dienen ***Terminal Operations***. Erst durch Aufruf einer Terminal Operation werden auch die durch die Intermediate Operations beschriebenen Verarbeitungsschritte tatsächlich ausgeführt.

Abschließend möchte ich ***Short-circuiting Operations*** vorstellen. Short-circuiting Operations zeichnen sich dadurch aus, dass sie ihre Berechnungen nicht immer vollständig für alle Elemente eines Streams ausführen müssen. Beispiele sind die Suche nach einem beliebigen Treffer oder danach, ob es überhaupt ein Element gibt, das einer gewünschten Bedingung genügt. Insbesondere bei Parallelverarbeitung können Short-circuiting Operations die Berechnung deutlich beschleunigen, etwa wenn nach einem Treffer keine weiteren Aktionen ausgeführt werden müssen. Short-circuiting Operations existieren sowohl für Intermediate Operations als auch und vor allem für Terminal Operations. In den nachfolgenden Auflistungen sind die Namen von Short-circuiting Operations jeweils kursiv dargestellt.

Intermediate Operations

Bei den zustandslosen Intermediate Operations sind folgende wichtig:

- `filter()` – Behält diejenigen Elemente, die das übergebene `Predicate<T>` erfüllen. Filtert alle anderen aus dem Stream heraus.
- `map()` – Transformiert Elemente mithilfe einer `Function<T,R>` vom Typ `T` auf solche mit dem Typ `R`. Im Speziellen können die Typen auch gleich sein.
- `flatMap()` – Bildet verschachtelte Streams auf einen flachen Stream ab.
- `peek()` – Führt eine Aktion für jedes Element des Streams aus. Dies kann für Debuggingzwecke sehr nützlich sein.

Darüber hinaus sollte man folgende zustandsbehaftete Intermediate Operations kennen:

- `distinct()` – Entfernt alle gemäß der Methode `equals(Object)` als Duplikate erkannte Elemente aus einem Stream.
- `sorted()` – Sortiert die Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- *`limit()`* – Begrenzt die maximale Anzahl der Elemente eines Streams auf einen als Parameter übergebenen Wert. Dies ist eine Short-circuiting Operation.
- `skip()` – Überspringt die ersten n Elemente eines Streams.

Terminal Operations

Neben den umfangreichen Intermediate Operations wird eine noch imposantere Zahl an Terminal Operations geboten, unter anderem folgende:

- `forEach()` – Führt eine Aktion für jedes Element des Streams aus.
- `toArray()` – Überträgt die Elemente aus dem Stream in ein Array.
- `collect()` – Sammelt die Elemente eines Streams auf und überträgt diese in eine Collection oder berechnet daraus Anzahl, Minimum, Maximum usw.
- `reduce()` – Verbindet die Elemente eines Streams. Ein Beispiel ist die kommaseparierte Konkatenation von Strings. Alternativ kann man auch Summationen, Multiplikationen usw. ausführen, um einen Ergebniswert zu berechnen.
- `min()` – Ermittelt das Minimum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `max()` – Ermittelt das Maximum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `count()` – Zählt die Anzahl an Elementen in einem Stream.
- *`anyMatch()`* – Prüft, ob es mindestens ein Element gibt, das die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.
- *`allMatch()`* – Prüft, ob alle Elemente die Bedingung eines `Predicate<T>` erfüllen. Dies ist eine Short-circuiting Operation, die allerdings abbricht, wenn sie das erste Gegenbeispiel gefunden hat.
- *`noneMatch()`* – Prüft, ob keines der Elemente die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.
- *`findFirst()`* – Liefert das erste Element des Streams, falls es ein solches gibt. Dies ist eine Short-circuiting Operation.
- *`findAny()`* – Liefert ein beliebiges Element, falls es ein solches gibt. Dies ist eine Short-circuiting Operation und kann manchmal günstiger sein als `findFirst()`, wenn es wirklich nur darum geht, einen beliebigen Treffer zu erhalten.

7.1.3 Zustandslose Intermediate Operations

In diesem Abschnitt betrachten wir verschiedene zustandslose Intermediate Operations. Dabei beginnen wir mit der Filterung von Werten und kommen dann zur Extraktion bzw. Abbildung. Im Anschluss lernen wir eine spezielle Mapping-Funktionalität kennen. Danach sehen wir uns das Filtern und die Extraktion im praktischen Einsatz an. Abschließend schauen wir auf die Inspektion von Verarbeitungsschritten. Dies erleichtert es beim Auftreten von Problemen, mögliche Fehlersituationen besser nachvollziehen zu können.

Die Methode `filter()` – Filterung

Das Filtern ist eine gebräuchliche Funktionalität, die bis Java 8 nicht durch das JDK bereitgestellt wurde.

Betrachten wir als Beispiel eine Liste von `Person`-Objekten. Daraus wollen wir mithilfe von `filter(Predicate<Person>)` diejenigen ermitteln, die erwachsen sind, indem wir die Methodenreferenz `Person::isAdult` wie folgt nutzen:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Micha", 43, "Zürich"));
    persons.add(new Person("Barbara", 40, Gender.FEMALE, "Hamburg"));
    persons.add(new Person("Yannis", 5, "Hamburg"));

    // final Predicate<Person> isAdult = person -> person.getAge() >= 18;
    final Stream<Person> adults = persons.stream().filter(Person::isAdult);

    adults.forEach(System.out::println);
}
```

Listing 7.2 *Ausführbar als* **'STREAMFILTEREXAMPLE'**

Die im obigen Listing gezeigte Bedingung `isAdult` kann man – wie im Kommentar angedeutet – als Lambda schreiben oder man verwendet eine hier besser lesbare Methodenreferenz, die auf die Methode `isAdult()` in einer Klasse `Person` verweist. Damit produziert das Programm STREAMFILTEREXAMPLE folgende Ausgaben:

```
Person [name=Micha, age=43, gender=MALE, city=Zürich]
Person [name=Barbara, age=40, gender=FEMALE, city=Hamburg]
```

Mehrstufige Filterung In der Praxis soll oftmals eine mehrstufige Filterung nach verschiedenen Kriterien erfolgen. Mit der Pipeline- oder Fließbandanalogie im Hinterkopf kann man dazu mehrere Filter hintereinanderschalten, wie dies folgendes Listing für drei Filterbedingungen zeigt:

```
final Stream<Person> allAdultMikes = persons.stream().
                            filter(Person::isAdult).
                            filter(person -> person.getName().equals("Mike").
                            filter(mike -> mike.livesIn("Zürich"));
```

Auf diese Weise filtern bzw. ermitteln wir zunächst alle Erwachsenen und dann alle, deren Name »Mike« ist. Aus dieser Ergebnismenge werden wiederum diejenigen behalten, die wohnhaft in Zürich sind.

Meinung: Namensgebung von Lambda-Parametern

Wie Sie vielleicht bemerkt haben, verwende ich für die Parameter in Lambdas bevorzugt sprechende Namen oder aber Standards wie `it`. Meiner Meinung nach gilt auch hier, dass man so lesbar wie möglich programmieren sollte. Nur weil man funktional programmiert, heißt das nicht, dass man wieder auf Namensverkümmerungen wie `a`, `p`, `x` zurückgreifen muss. Natürlich gibt es auch Fälle, in denen Kürzel mit einem Buchstaben ihren Wert haben. Das gilt immer dann, wenn im Lambda eine beliebige Berechnung erfolgt, etwa `x -> x + 1`. Dabei trägt der Parameter keine oder nur wenig semantische Bedeutung – meistens, weil eine »echte« mathematische Funktion beschrieben wird.

Die Methode `map()` – Mapping von Daten, Extraktion von Werten

Neben der Filterung ist die Konvertierung oder Extraktion von Werten eine typische Intermediate Operation. Hierbei soll eine Menge von Eingabedaten in ein anderes Format überführt oder abgebildet werden. So könnte etwa aus einer Liste von Personen jeweils das Attribut Name, Vorname oder Alter extrahiert werden. Dabei findet eine Abbildung oder ein Mapping von einem Typ auf einen anderen statt: Im Beispiel wird aus dem Typ `Person` ein Attribut ausgelesen und auf denjenigen Typ des gewünschten Attributs, z. B. `String`, abgebildet. Dazu kann man Spezialisierungen des Interface `Function<T,R>` nutzen und dort die Methode `apply(T)` entsprechend implementieren. Das haben wir bereits in Abschnitt 4.5.2 im Zusammenhang mit dem Interface `UnaryOperator<T>` kurz besprochen. Hier wird die Definition des Interface `Function<T,R>` zum Einstieg wiederholt:

```
interface Function<T,R>
{
    R apply(T t);
}
```

Nehmen wir an, es wäre der Name aus einem `Person`-Objekt zu extrahieren. Dies implementieren wir mithilfe eines Lambdas oder mit einer Methodenreferenz wie folgt:

```
// Lambda
final Function<Person, String> nameExtractor_V1 = person -> person.getName();

// Alternativ mit Methodenreferenz
final Function<Person, String> nameExtractor_V2 = Person::getName;
```

Extraktion am Beispiel Mit dem gesammelten Wissen machen wir uns daran, die Extraktion des Namens bzw. des Alters für eine Liste von Personen zu programmieren. Dazu setzen wir die Methode `map()` folgendermaßen ein:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Barbara", 40, Gender.FEMALE, "Hamburg"));
    persons.add(new Person("Yannis", 5, "Hamburg"));

    // Mapping auf Name mit Lambda
    final Stream<Person> adults = persons.stream().filter(Person::isAdult);
    final Stream<String> namesStream = adults.map(person -> person.getName());
    // Mapping auf Alter mit Methodenreferenz
    final Stream<Integer> agesStream = persons.stream().map(Person::getAge).
                                            filter(age -> age >= 18);

    namesStream.forEach(System.out::println);
    agesStream.forEach(System.out::println);
}
```

Listing 7.3 *Ausführbar als* '**ATTRIBUTEEXTRACTIONEXAMPLE**'

Das Programm ATTRIBUTEEXTRACTIONEXAMPLE erzeugt folgende Ausgaben:

```
Barbara
40
```

Die Methode `flatMap()` – spezielle Arten von Mappings

Während sowohl die `filter()`- als auch die `map()`-Methode recht intuitiv zu benutzen sind, wirkt die `flatMap()`-Methode zunächst nicht ganz so eingängig. Rekapitulieren wir kurz: `filter()` und `map()` werden auf einem Stream angewendet und liefern einen neuen Stream zurück – `map()` gibt gegebenenfalls einen Stream eines anderen Typs zurück, etwa wenn man von `Person`-Objekten auf deren Attribut `name` abbildet.

Bei Abbildungen gibt es einen Spezialfall zu beachten, nämlich den verschachtelter Streams. Das lässt sich am besten an einem Beispiel verdeutlichen. Gegeben sei eine Menge von Sätzen.[1] Für diese soll die Häufigkeit der Vorkommen einzelner Wörter gezählt werden. Bevor wir uns später an die Implementierung dieser Funktionalität machen, wollen wir uns zunächst dem Stream-im-Stream-Problem widmen, das aus der Beschreibung möglicherweise noch nicht ganz deutlich geworden ist. Schauen wir uns folgende Ausgangslage an:

```
final List<String> sentences = Arrays.asList( "This is the first line.",
                                              "The second line of this text.",
                                              "Third line contains some text.",
                                              "Last line and goodbye:",
                                              "End of text!");

final Stream<String> asStream = sentences.stream();
```

[1] Wir könnten uns vorstellen, diese Zeilen entstammten einer Textdatei.

Um aus diesen Sätzen einzelne Wörter zu extrahieren, könnte man die Methode `String.split(String)` nutzen, die ein `String[]` liefert. Da `map()` einen Stream als Eingabe erwartet, ist hier zusätzlich noch ein Aufruf von `Stream.of()` nötig, um das Array in einen Stream zu konvertieren:

```
Stream<Stream<String>> words = asStream.map(line -> Stream.of(line.split(" ")));
```

Das Ergebnis des Aufrufs ist allerdings ein verschachtelter Stream. Nachfolgend symbolisieren die Zeichen < > die Begrenzer des Streams im Beispiel:

```
<<This, is, the, first, line.>,<The, second, line, ... <End, of, text!>>
```

Diese Form erschwert es aber, über die einzelnen Werte zu iterieren. Zur Auswertung wünscht man sich, dass die geschachtelten Streams »flach geklopft« werden und als Ergebnis ein `Stream<String>` entsteht. Das ist recht einfach möglich, indem man die Methode `flatMap()` folgendermaßen nutzt:

```
Stream<String> words = asStream.flatMap(line -> Stream.of(line.split(" ")));
```

Beispiel: Die Intermediate Operations in Aktion

Wir haben nun einige Intermediate Operations kennengelernt. Um das Wissen darüber zu vertiefen, wollen wir das vorherige Beispiel zur Auswertung einer Menge von Sätzen ausbauen. Ziel ist es, die Häufigkeiten von Wörtern zu zählen. Damit das Ganze etwas reizvoller und realitätsnäher wird, gibt es folgende weitere Anforderungen: Die Sätze sollen aus einer Datei eingelesen werden und alle Wörter mit drei oder weniger Buchstaben sollen nicht gezählt werden. Dazu formulieren wir ein `Predicate<String>` mit dem sprechenden Namen `isShortWord`. Darüber hinaus sollen verschiedene Füllwörter bei der Zählung der Häufigkeiten nicht betrachtet werden. Daher wollen wir die Wörter »this«, »these« und »them« ignorieren. Auch dazu formulieren wir ein korrespondierendes `Predicate<String>` namens `isIgnorableWord`. Durch Negation erhalten wir das Prädikat `isSignificantWord`. Die gewünschte Filterung realisieren wir dann mit den Prädikaten wie folgt:

```
public static void main(final String[] args) throws IOException
{
    final Path exampleFile = Paths.get("src/main/resources/" +
                                       "ch07_stream_api/Example.txt");

    // Datei einlesen neu in JDK 8
    final List<String> contents = Files.readAllLines(exampleFile);

    // Daraus einen Stream von Worten machen
    final Stream<String> words = contents.stream().
                                 flatMap(line -> Stream.of(line.split(" ")));

    // Prädikate für kurze Wörter
    final Predicate<String> isShortWord = word -> word.length() <= 3;
    final Predicate<String> notIsShortWord = isShortWord.negate();
```

```
    // Prädikate für spezielle und zu ignorierende Wörter
    final List<String> ignoreableWords = Arrays.asList("this", "these", "them");
    final Predicate<String> isIgnorableWord = word ->
    {
        return ignoreableWords.contains(word.toLowerCase());
    };
    final Predicate<String> isSignificantWord = isIgnorableWord.negate();

    // Filterung basierend auf den Prädikaten
    final Stream<String> filteredContents = words.map(String::trim).
                                            filter(notIsShortWord).
                                            filter(isSignificantWord);

    filteredContents.forEach(it -> System.out.print(it + ", "));
}
```

Listing 7.4 *Ausführbar als* '**FLATMAPEXAMPLE**'

Das Programm FLATMAPEXAMPLE gibt die gefilterten Ergebnisse wie folgt aus:

```
first, line., second, line, text., Third, line, contains, some, text.,
Last, line, goodbye:, text!,
```

An der Ausgabe sind zwei Schwächen zu erkennen: Zum einen sind noch Satzzeichen wie Punkt, Doppelpunkt und Ausrufezeichen Bestandteil der Wörter. Zum anderen sind die Wörter nicht sortiert. Hier verwenden wir die intuitiv verständliche Methode `sorted()`, um die Ausgabe zu sortieren. Die Methode wird später im Rahmen der zustandsbehafteten Intermediate Operations genauer besprochen. Darüber hinaus nutzen wir ein Mapping, um die Satzendezeichen zu entfernen. Statt des `forEach()`-Aufrufs zur Ausgabe schreiben wir folgende Zeilen:

```
final Function<String, String> removePunctationMarks = str ->
{
    if (str.endsWith(".") || str.endsWith(":") || str.endsWith("!"))
    {
        return str.substring(0, str.length()-1);
    }
    return str;
};

final Stream<String> mapped = filteredContents.map(removePunctationMarks);
final Stream<String> sorted = mapped.sorted(String.CASE_INSENSITIVE_ORDER);

sorted.forEach(it -> System.out.print(it + ", "));
```

Diese Modifikationen sind im Programm FLATMAPEXAMPLE2 umgesetzt. Dessen Start produziert folgende Ausgabe:

```
contains, first, goodbye, Last, line, line, line, line, second, some,
text, text, text, Third,
```

Wir sind schon fast am Ziel der Gruppierung und Häufigkeitsbestimmung. Allerdings könnten wir das abschließende Komma in der Ausgabe bemängeln. Diese Thematik werden wir bei der Besprechung der Terminal Operations wieder aufgreifen. Vorher

betrachten wir eine Besonderheit bei Intermediate Operations, nämlich die Inspektion von Verarbeitungsschritten.

Die Methode `peek()` – Inspektion von Verarbeitungsschritten

Für den Fall, dass Intermediate Operations komplexer werden, möchte man sich eventuell auch einmal Zwischenergebnisse ausgeben lassen und danach die Verarbeitung fortsetzen. Auf ein Problem stößt man, wenn man einen bereits mit `forEach(Consumer<? super T>)` ausgegebenen oder mit einer Terminal Operation verarbeiteten Stream weiter bearbeiten möchte. Das folgende Beispiel zeigt die Problematik:

```
final List<Person> persons = Arrays.asList(new Person("Michael", 43),
                                           new Person("Mike", 42));
final Stream<Person> adults = persons.stream().filter(Person::isAdult);

// Ausgabe, um die Filterung zu überprüfen
adults.forEach(System.out::println);

// Weitere Filterung auf dem Stream vornehmen
final Stream<Person> mikes = adults.filter(person ->
                                           person.getName().equals("Mike"));
```

Listing 7.5 *Ausführbar als* '**WRONGSTREAMINSPECTIONEXAMPLE**'

Führt man diese Schritte aus, so kommt es anstelle einer Weiterverarbeitung zu einer `IllegalStateException` mit dem Hinweis »`stream has already been operated upon or closed`«. Daran erkennt man die bereits erwähnte Eigenschaft von Streams, die Daten nur einmal bereitstellen zu können.[2]

Weil die Inspektion von Zwischenzuständen aber eine sehr wünschenswerte Funktionalität ist, stellt das Stream-API hierfür eine Möglichkeit bereit. Mithilfe der Intermediate Operation in Form der Methode `peek(Consumer<? super T>)` kann man beliebige Aktionen, etwa Konsolenausgaben, durchführen. Dies gleicht der Verarbeitung mit `forEach(Consumer<? super T>)`. Im Gegensatz dazu erhält man aber beim Aufruf von `peek(Consumer<? super T>)` wiederum einen Stream, mit dem man weiter arbeiten kann. Obiges Beispiel lässt sich zur Realisierung einer Inspektion folgendermaßen abändern:

```
final List<Person> persons = Arrays.asList(new Person("Michael", 43),
                                           new Person("Mike", 42));
final Stream<Person> adults = persons.stream().filter(Person::isAdult);

// Ausgabe mit peek(), um die Filterung zu überprüfen
final Stream<Person> adultsPeek = adults.peek(System.out::println);

// Weitere Filterung auf dem Stream vornehmen
final Stream<Person> mikes = adultsPeek.filter(person ->
                                               person.getName().equals("Mike"));
```

Listing 7.6 *Ausführbar als* '**STREAMINSPECTIONCORRECTEDEXAMPLE**'

[2] Deswegen spricht man auch von »Terminal« Operations.

Nachfolgend zeige ich eine Variante, die nach jedem Verarbeitungsschritt der Pipeline eine Ausgabe vornimmt – im Listing jeweils fett markiert:

```
public static void main(final String[] args)
{
   final List<Person> persons = createDemoData();

   final Stream<Person> stream = persons.stream();
   final Stream<String> allMikes = stream.peek(System.out::println).
                                          filter(Person::isAdult).
                                          peek(System.out::println).
                                          map(Person::getName).
                                          peek(System.out::println).
                                          filter(name -> name.startsWith("Mi")).
                                          peek(System.out::println).
                                          map(String::toUpperCase);

    // Löst die Verarbeitung aus
    System.out.println("Protokollierung jedes Schritts -- Filter 'Mi':");
    allMikes.forEach(System.out::println);
}

private static List<Person> createDemoData()
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Michael", 43));
    persons.add(new Person("Max", 5));
    persons.add(new Person("Moritz", 7));
    persons.add(new Person("Merten", 38));
    persons.add(new Person("Micha", 42));
    return persons;
}
```

Listing 7.7 *Ausführbar als* **'STREAMPEEKEXAMPLE'**

Nach dem Start des Programms STREAMPEEKEXAMPLE werden die Filtervorgänge auf der Konsole protokolliert:

```
Protokollierung jedes Schritts -- Filter 'Mi':
Person [name=Michael, age=43, gender=MALE]
Person [name=Michael, age=43, gender=MALE]
Michael
Michael
MICHAEL
Person [name=Max, age=5, gender=MALE]
Person [name=Moritz, age=7, gender=MALE]
Person [name=Merten, age=38, gender=MALE]
Person [name=Merten, age=38, gender=MALE]
Merten
Person [name=Micha, age=42, gender=MALE]
Person [name=Micha, age=42, gender=MALE]
Micha
Micha
MICHA
```

Anhand der Ausgabe erkennt man, dass die Verarbeitung nicht je Verarbeitungsschritt komplett für den Stream auf einmal, sondern elementweise geschieht: Die Daten werden Element für Element bearbeitet und durchlaufen jeweils einzeln die Stufen der Verarbeitungskette. Diese Art der Ausführung bildet die Grundlage für die Parallelisierbar-

keit von Aktionen. Darüber hinaus wird der bereits erwähnte fundamentale Unterschied von Streams zu sonstigen Verarbeitungen klar, nämlich, dass Intermediate Operations wirklich erst dann abgearbeitet werden, wenn eine Terminal Operation eine Berechnung auslöst. Das können Sie prüfen, wenn Sie die letzte Zeile in der `main()`-Methode mit dem Aufruf von `forEach(Consumer<? super T>)` auskommentieren. Dann werden Sie keine Ausgaben sehen.

7.1.4 Zustandsbehaftete Intermediate Operations

Kommen wir nun zum Thema zustandsbehafteter Intermediate Operations. Wir lernen verschiedene Varianten davon kennen: Das Sortieren und Herausfiltern doppelter Einträge zeige ich zuerst. Danach gehe ich auf das Beschränken der Ausgabe auf eine gewisse Anzahl von Elementen ein: Dabei kann sowohl der Startwert als auch die Anzahl der gewünschten, im Stream zu verbleibenden Elemente festgelegt werden.

Die Methoden `distinct()` und `sorted()` – Duplikation entfernen und Ausgaben sortieren

Das Herausfiltern doppelter Einträge und das Sortieren von Einträgen ist im folgenden Listing durch Aufrufe der Methoden `distinct()` und `sorted()` realisiert. Dabei betrachten wir zunächst die Ausführung jeder Methode für sich und danach in Kombination:

```
public static void main(final String[] args)
{
    final Stream<Integer> distinct = createIntStream().distinct();
    final Stream<Integer> sorted= createIntStream().sorted();
    final Stream<Integer> sortedAndDistinct = createIntStream().sorted().
                                                              distinct();

    printResult("distinct:          ", distinct);
    printResult("sorted:            ", sorted);
    printResult("sortedAndDistinct: ", sortedAndDistinct);
}

private static Stream<Integer> createIntStream()
{
    return Stream.of(7, 1, 4, 3, 7, 2, 6, 5, 7, 9, 8);
}

private static void printResult(final String hint,
                                final Stream<Integer> stream)
{
    final List<Integer> result = stream.collect(Collectors.toList());
    System.out.println(hint +  result);
}
```

Listing 7.8 *Ausführbar als* '**SORTEDANDDISTINCTEXAMPLE**'

Führt man das Programm SORTEDANDDISTINCTEXAMPLE aus, so werden doppelte Elemente entfernt und die Zahlen sortiert. Man erhält die erwartete Ausgabe:

```
distinct:          [7, 1, 4, 3, 2, 6, 5, 9, 8]
sorted:            [1, 2, 3, 4, 5, 6, 7, 7, 7, 8, 9]
sortedAndDistinct: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Wenn Sie im Listing ganz genau hingeschaut haben, dann ist Ihnen vielleicht der Aufruf von `collect(Collectors.toList())` aufgefallen. Dabei handelt es sich wie bei `forEach(Consumer<? super T>)` um eine Terminal Operation. Ein Aufruf von `collect(Collectors.toList())` überträgt die Daten aus einem Stream in eine Liste. Zum Thema Terminal Operations erfahren Sie gleich mehr.

Die Methoden `limit()` und `skip()` – Ausgaben beschränken

Die bereits gezeigte Filterung mit `filter(Predicate<? super T>)` erlaubt es, den Datenbestand einzuschränken. Eine Variante davon ist es, die Ergebnismenge auf n Elemente zu limitieren. Dazu kann man einen Aufruf von `limit(long)` nutzen. In Kombination mit `skip(long)` zum Überspringen von n Datensätzen lässt sich sogenanntes ***Paging*** realisieren – eine Aufbereitung von n Ergebnissen auf einer Seite, wie man dies etwa von der Präsentation von Suchergebnissen im Internet kennt. Man kann die Ausgabe folgendermaßen auf 25 Sucheinträge ab dem 75. Eintrag beschränken:

```
searchResults.skip(75).limit(25);
```

Es gibt aber noch weitere Anwendungsfälle für `limit(long)` und `skip(long)`, nämlich im Zusammenhang mit unendlichen Streams, etwa für eine Folge von `int`-Werten:

```
final IntStream iteratingValues = IntStream.iterate(0, x -> x + 1);

final IntStream values = iteratingValues.skip(50).limit(12);
values.forEach(System.out::println); // => 50,51,52,53,54,55,56,57,58,59,60,61
```

Listing 7.9 *Ausführbar als* '**SKIPANDLIMITEXAMPLE**'

Im Beispiel sehen wir die Begrenzung des unendlichen Streams auf bestimmte Datensätze – hier die 12 Einträge, die auf die ersten 50 Einträge folgen.

Hinweis: Reihenfolge der Aufrufe

Die beiden Methoden scheinen relativ selbsterklärend zu sein, doch man kann sich fragen, ob die Reihenfolge der Aufrufe eine Rolle spielt und wie sich diese auswirkt: Mit `limit()` begrenzt man in jedem Fall den Stream und mit `skip()` überspringt man Elemente. Macht es also einen Unterschied, ob man zunächst `skip()` und danach `limit()` oder andersherum aufruft? Schauen wir uns ein Beispiel an:

```
final Stream<Integer> values1 = Stream.of(1, 2, 3, 4, 5, 6, 7);
final Stream<Integer> values2 = Stream.of(1, 2, 3, 4, 5, 6, 7);

System.out.println("skip(3).limit(4)");
values1.skip(3).limit(4).forEach(num -> System.out.print(num + " "));

System.out.println("\nlimit(4).skip(3)");
values2.limit(4).skip(3).forEach(num -> System.out.print(num + " "));
```

Listing 7.10 *Ausführbar als* **'SKIPLIMITORDEREXAMPLE'**

Mit etwas Nachdenken ist ganz offensichtlich, dass sich die Ergebnisse unterscheiden. Ansonsten erhalten Sie die Antwort spätestens durch den Start des Programms SKIPLIMITORDEREXAMPLE:

```
skip(3).limit(4)
4 5 6 7
limit(4).skip(3)
4
```

7.1.5 Terminal Operations

Bisher haben wir mithilfe von Streams verschiedene Berechnungen ausgeführt und dabei häufig die Terminal Operation `forEach(Consumer<? super T>)` genutzt, um Konsolenausgaben zu produzieren. Betrachten wir Terminal Operations nun ein wenig allgemeiner. Erinnern wir uns zunächst nochmals daran, dass diese zur Abarbeitung der Pipeline führen und dadurch ein Ergebnis produzieren.

Die Methode `forEach()` – Verarbeitung oder Konsolenausgaben

Nachfolgend ist zunächst der Vollständigkeit halber die Aufbereitung von Konsolenausgaben mit `forEach(Consumer<? super T>)` gezeigt:

```
streamFromArray.forEach(System.out::println);
streamFromValues.sorted().distinct().forEach(System.out::println);
```

Die Methode `toArray()` – Streams in Arrays übertragen

Statt Berechnungsergebnisse auf der Konsole auszugeben, ist es meistens sinnvoller, diese etwa in ein Array oder eine Collection zu überführen. Hier werden wir zunächst die Methode `toArray()` betrachten. Im nachfolgenden Abschnitt zur Methode `collect()` schauen wir uns dann eine Umwandlung der Daten eines Streams in eine Collection an.

Betrachten wir als Beispiel die Aktion »Erzeuge sieben Zufallszahlen von 0 bis 100 und stelle die Berechnungsergebnisse in einem Array bereit«. Dabei ist die Methode `toArray()` nützlich, die normalerweise ein `Object[]` liefert – für Streams primitiver Typen werden aber Arrays von diesen Typen erzeugt, hier ein `int[]`:

```
public static void main(final String[] args)
{
    // Zufallszahlen von 0 bis 100
    final Random random = new Random();
    final Supplier<Float> randomSupplier = () -> random.nextFloat() * 100;

    final Object[] randomNumbers = Stream.generate(randomSupplier).
                                          limit(7).toArray();
    System.out.println(Arrays.toString(randomNumbers));
    System.out.println("Element type: " + randomNumbers[0].getClass());

    final int[] intRandoms = Stream.generate(randomSupplier).
                             limit(7).mapToInt(val -> val.intValue()).toArray();
    System.out.println(Arrays.toString(intRandoms));
}
```

Listing 7.11 *Ausführbar als* **'STREAMTOARRAYEXAMPLE'**

Das Programm STREAMTOARRAYEXAMPLE produziert in etwa folgende Ausgaben:

```
[82.59304, 92.31408, 33.92508, 44.195183, 59.675797, 16.191816, 99.11495]
Element type: class java.lang.Float
[91, 12, 44, 48, 44, 69, 58]
```

Die Methode `collect()` – Streams in Collections übertragen

Für diverse Anwendungsfälle ist es erforderlich, die Daten aus einem `Stream<T>` in einer `Collection<E>` zu speichern. Mithilfe von `java.util.stream.Collector`-Instanzen kann man die Daten auslesen und z. B. in eine Liste übertragen. Praktischerweise bietet die Utility-Klasse `java.util.stream.Collectors` schon verschiedene vordefinierte Methoden, die passende `Collector`-Instanzen zurückliefern. Dadurch reduziert sich die Komplexität der `Collector`-Bereitstellung stark, wie dies nachfolgend gezeigt ist:

```
final List<Integer> ages = agesStream.collect(Collectors.toList());
final List<String> names = namesStream.collect(Collectors.
                                          toCollection(ArrayList::new));
```

Im Listing sehen wir den für viele Anwendungsfälle praktischen Aufruf von `toList()`. Benötigt man mehr Kontrolle über den Typ der Ergebnisdatenstruktur, so kann man die Methode `toCollection()` aufrufen, der man die Referenz auf den Konstruktor der gewünschten Collection übergibt, wie dies im zweiten Aufruf dargestellt ist.

Die Methoden `count()`, `sum()`, `min()`, `max()` und `average()`

Neben Konsolenausgaben oder der Übertragung in eine Collection stellen Berechnungen auf den Daten gebräuchliche Terminal Operations dar. Für Zahlenwerte sind etwa die Ermittlung von Minimum, Maximum, Summe oder Durchschnitt typisch. Aber auch die Anzahl an Elementen im Stream gehört dazu.

Nachfolgend bestimmen wir für eine Liste von Personen mit `count()`, wie viele davon mit dem Buchstaben »T« im Namen beginnen. Dann summieren wir durch Aufruf der Methode `sum()` die Altersangaben dieser Personen. Das Ergebnis ist vom Typ `int`. Ebenso können wir durch Aufruf von `min()` bzw. `max()` das Minimum bzw. Maximum, also hier das Alter der jüngsten bzw. ältesten Person, ermitteln (im Listing nicht gezeigt). Wenn wir das Durchschnittsalter berechnen wollen, können wir dazu die Methode `average()` nutzen. Dabei lassen sich zwei Dinge lernen. Zum einen ist das Ergebnis keine Ganzzahl mehr und zum anderen existiert möglicherweise kein Durchschnitt, nämlich dann, wenn die Quelle für den Stream leer ist. Um diesen Sachverhalt ausdrücken zu können, gibt `average()` den Typ `OptionalDouble` zurück, der optionale Werte mit dem Typ `double` modelliert. Details zur Repräsentation optionaler Werte beschreibt Abschnitt 6.5. Das folgende Programm demonstriert die Anwendung:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Ten", 10));
    persons.add(new Person("Twenty", 20));
    persons.add(new Person("Thirty", 30));
    persons.add(new Person("Forty", 40));

    final Predicate<Person> startsWithT = person ->
                                          person.getName().startsWith("T");

    // Anzahl an Personen, deren Name mit 'T' startet
    final int count = persons.stream().filter(startsWithT).count();
    System.out.println("count: " + count);

    // Summe berechnen
    final int sum = persons.stream().filter(startsWithT).
                                    mapToInt(Person::getAge).sum();
    System.out.println("sum: " + sum);

    // Durchschnitt berechnen
    final OptionalDouble avg = persons.stream().
                                       filter(person -> person.getName().
                                       contains("X")).
                                       mapToInt(Person::getAge).average();
    System.out.println("avg: " + avg);
}
```

Listing 7.12 *Ausführbar als* '**CALCULATIONEXAMPLE**'

Starten wir das Programm CALCULATIONEXAMPLE, so werden mit `count()` nur drei Personen gezählt, weil die Person namens Forty durch das Prädikat `startWithT` herausgefiltert wird. Dadurch errechnet die Methode `sum()` für die Altersangaben die Summe 60. Die als letztes ausgeführte Berechnung des Durchschnitts der Altersangaben (`average()`) für Personen mit einem »X« im Namen kann jedoch keinen Wert liefern, da keine entsprechenden Einträge dafür existieren. Dies wird durch den Wert `OptionalDouble.empty` ausgedrückt. Somit kommt es zu folgenden Ausgaben:

```
count: 3
sum: 60
avg: OptionalDouble.empty
```

Die Auswertung verschiedener statistischer Daten erfordert leider wiederholt die Erzeugung neuer Streams, da `min()`, `max()` usw. jeweils eine Terminal Operation sind. Praktischerweise gibt es im JDK die Klasse `java.util.IntSummaryStatistics`, für die es jeweils auch eine Ausprägung für den Typ `long` und `double` gibt. Diese Klassen ermöglichen es, die Werte von Statistikberechnungen zentral zu verarbeiten. Für das obige Beispiel kann man aggregierte Informationen über das Alter für Personen, deren Name mit einem »T« beginnt, mit nur einem Aufruf wie folgt ermitteln:

```
final IntStream ages = persons.stream().filter(startsWithT).
                                        mapToInt(Person::getAge);

final IntSummaryStatistics stats = ages.summaryStatistics();
System.out.println("stats: " + stats);
```

Ergänzt man diese Zeilen im Programm, so werden folgende Statistikdaten ausgegeben:

```
stats: IntSummaryStatistics{count=3, sum=60, min=10, average=20,000000, max=30}
```

Hintergrundwissen: Verarbeitungsmethoden in Wrapper-Klassen

Damit sich bestimmte Stream-Operationen leichter formulieren lassen, wurden die Wrapper-Klassen `Integer`, `Long` und `Double` um verschiedene Methoden erweitert, besonders hervorzuheben sind hier `min()`, `max()` und `sum()`.

Die Methoden `allMatch()`, `anyMatch()`, `noneMatch()`

Gebräuchliche Anwendungsfälle beim Verarbeiten von Daten sind Prüfungen, ob eine gewisse Bedingung durch alle, einige Elemente oder aber kein Element erfüllt wird, z. B., ob alle Personen älter als 18 Jahre sind oder ob Namen mit einem gewissen Buchstaben beginnen. Zur Umsetzung bietet das `Stream<T>`-Interface die drei Methoden

- `allMatch(Predicate<? super T>)`,
- `anyMatch(Predicate<? super T>)` und
- `noneMatch(Predicate<? super T>)`,

die anhand des übergebenen `Predicate<T>` ein Ergebnis vom Typ `boolean` berechnen. Das Listing zeigt drei Varianten von Tests, die auf den Startbuchstaben »`T`« prüfen:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Tim", "Tom", "Micha");

   final Predicate<String> startsWithT = str -> str.startsWith("T");

   final boolean allStartWithT = names.stream().allMatch(startsWithT);
   final boolean anyStartWithT = names.stream().anyMatch(startsWithT);
   final boolean noneStartWithT = names.stream().noneMatch(startsWithT);

   System.out.println("allStartWithT:  " + allStartWithT);
   System.out.println("anyStartWithT:  " + anyStartWithT);
   System.out.println("noneStartWithT: " + noneStartWithT);
}
```

Listing 7.13 *Ausführbar als* '**MATCHEXAMPLE**'

Die Ausführung des Programms MATCHEXAMPLE produziert folgende Ausgaben, die die obigen Aussagen zur Arbeitsweise verdeutlichen:

```
allStartWithT:  false
anyStartWithT:  true
noneStartWithT: false
```

Die Methoden `findAny()` und `findFirst()`

Auch das Suchen ist ein so verbreiteter Anwendungsfall, dass er im Interface `Stream<T>` durch die Methoden `findAny()` bzw. `findFirst()` bereitgestellt wird. Anders als man zunächst aufgrund der Kenntnis der zuvor vorgestellten Methoden `allMatch()`, `anyMatch()` und `noneMatch()` meinen könnte, wird an `findAny()` bzw. `findFirst()` kein `Predicate<T>` übergeben, sondern der Aufruf geschieht folgendermaßen:

```
final Optional<Person> optionalAny = filteredPersons.findAny();
final Optional<Person> optionalFirst = filteredPersons.findFirst();
```

Ein Aufruf von `findAny()` liefert ein beliebiges Element aus dem Stream, falls es ein solches gibt. `findFirst()` gibt dagegen das erste Element des Streams zurück, sofern es ein solches gibt. In beiden Fällen erfolgt die Rückgabe des Werts `Optional.empty`, wenn kein passendes Element bereitgestellt werden kann.

Für Anwendungsfälle in der Praxis ist es wahrscheinlich, dass vor der Ermittlung eines Elements eine Filterung ausgeführt wird:

```
final Predicate<String> startsWithT = str -> str.startsWith("T");

final Optional<Person> optionalAny = names.stream().filter(startsWithT).
                                                    findAny();
```

Beispiel zu `anyMatch()` und `findFirst()`

Nehmen wir an, wir wollten feststellen, ob in einer Liste von Personen eine Person mit dem gewünschten Namen enthalten ist, bzw. wir möchten nach einer Person anhand des Namens suchen und den ersten Treffer zurückliefern. Vor JDK 8 würde man entsprechende Methoden `containsPersonWithName()` und `findPersonByName()` wohl wie folgt implementieren:

```
boolean containsPersonWithName(final List<Person> persons, final String desired)
{
    return findPersonByName(person, desired) != null;
}

Person findPersonByName(final List<Person> persons, final String desired)
{
    for (final Person person : persons)
    {
        if (person.getName().equals(desired))
        {
            return person;
        }
    }

    return null;
}
```

Mit Lambdas in Kombination mit Streams schreiben wir kürzer Folgendes:

```
// Namensfilter definieren
final Predicate<Person> nameFilter = person -> person.getName().equals(desired);

// containsPersonWithName()
final boolean personFound = persons.stream().anyMatch(nameFilter);

// findPersonByName()
final Optional<Person> searchedPerson = persons.stream().filter(nameFilter).
                                                         findFirst();
```

Die Filterbedingung vom Typ `Predicate<T>` realisieren wir als Lambda. Diesen `nameFilter` nutzen wir als Eingabe für einen Aufruf von `anyMatch(Predicate<T>)` zum Ermitteln, ob mindestens ein Element im Stream die übergebene Bedingung erfüllt. Außerdem wird der `nameFilter` an eine Kombination aus `filter(Predicate<T>)` und `findFirst()` übergeben, wodurch eine Filterung erfolgt und der erste Eintrag der Treffermenge ermittelt wird. Diese kann durchaus leer sein. Gewöhnlich wird so etwas durch die Rückgabe von `null` oder von Null-Objekten gemäß dem gleichnamigen Entwurfsmuster implementiert. Wie bereits für optionale Rückgaben in Abschnitt 6.5 gesehen, bietet sich dafür die generische Klasse `java.util.Optional<T>` an, um optionale Werte in Form eines Objekts zu modellieren. Existiert kein gültiger Wert, so wird dies durch den Wert `Optional.empty` ausgedrückt.

Die Methode `reduce()` – Elemente zusammenfassen

Wir haben mittlerweile verschiedene Terminal Operations kennengelernt, etwa solche, die aus einer Menge von Elementen einen akkumulierten Wert wie die Summe, den Durchschnitt oder einen booleschen Wert berechnet haben.

Nun wollen wir die Methode `reduce(BinaryOperator<T>)` betrachten. Durch deren Einsatz kann man Elemente des gleichen Typs `T` miteinander kombinieren. Dafür kann eine beliebige Operation genutzt werden: Zunächst schauen wir uns dies für Strings und deren kommaseparierte Konkatenation an, danach für Zahlen und deren Multiplikation, wodurch sich die Fakultät berechnen lässt:

```
public static void main(final String[] args)
{
   final Stream<String> names = Stream.of("Mike", "Tom", "Peter", "Chris");
   final Stream<Integer> integers = Stream.of(1, 2, 3, 4, 5, 6);

   final BinaryOperator<String> csvCombine = (str1, str2) -> str1 + ", " + str2;
   final BinaryOperator<Integer> multiply = (x, y) -> x * y;

   final Optional<String> stringConcat = names.reduce(csvCombine);
   final Optional<Integer> factorial = integers.reduce(multiply);

   System.out.println("stringConcat: " + stringConcat);
   System.out.println("factorial:    " + factorial);
}
```

Listing 7.14 *Ausführbar als* '**REDUCEEXAMPLE**'

Das Programm REDUCEEXAMPLE erzeugt die nachfolgend gezeigten Ausgaben. Diese verdeutlichen die Konkatenation bzw. Berechnung, wobei die von einem `Optional<T>` gespeicherten Werte in dessen Stringausgabe in eckigen Klammern angegeben sind:

```
stringConcat: Optional[Mike, Tom, Peter, Chris]
factorial:    Optional[720]
```

Die Methoden `joining()`, `groupingBy()` und `partitioningBy()`

Die Utility-Klasse `Collectors` bietet verschiedene Hilfsmethoden, die man gewinnbringend in Kombination mit der Methode `collect()` für weitere Transformationen, etwa das Verknüpfen von Strings sowie die Gruppierung oder Partitionierung von Daten, nutzen kann. Wir wollen einen Blick auf folgende Methoden werfen:

- `joining()` – Fasst Einträge vom Typ `String` zusammen. Das ist nützlich, um etwa eine kommaseparierte Auflistung zu realisieren.
- `groupingBy()` – Gruppiert Elemente anhand eines Kriteriums.
- `counting()` – Zählt die Vorkommen in Kombination mit `groupingBy()`.
- `partitioningBy()` – Unterteilt die Eingabedaten basierend auf einer Realisierung eines `Predicate<T>` in zwei Partitionen.

Den Einsatz der obigen Methoden zeigt das folgende Listing, wobei hier zur besseren Lesbarkeit der Berechnungen statische Imports genutzt werden. Als Beispieldaten verwenden wir verschiedene Namen, die wir unter anderem nach Länge gruppieren und die Vorkommen wie folgt zählen:

```
import static java.util.stream.Collectors.counting;
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.joining;
import static java.util.stream.Collectors.partitioningBy;

// ...

final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",
                                         "Florian", "Michael", "Sebastian");

final String joined = names.stream().sorted().collect(joining(", "));

Map<Integer, List<String>> grouped =
                           names.stream().collect(groupingBy(String::length));

Map<Integer, Long> counting =
                   names.stream().collect(groupingBy(String::length,
                                          counting()));

Map<Boolean, List<String>> partitions =
                           names.stream().filter(str -> str.contains("i")).
                           collect(partitioningBy(str -> str.length() > 4));

// ...
```

Listing 7.15 *Ausführbar als* **'COLLECTORSSPECIALEXAMPLE'**

Das Programm COLLECTORSSPECIALEXAMPLE produziert folgende Ausgaben, anhand derer sich die zuvor kurz beschriebene Arbeitsweise der Methoden erschließt:

```
joined:     Andi, Florian, Michael, Mike, Ralph, Sebastian, Stefan
grouped:    {4=[Andi, Mike], 5=[Ralph], 6=[Stefan], 7=[Florian, Michael],
             9=[Sebastian]}
counting:   {4=2, 5=1, 6=1, 7=2, 9=1}
partitions: {false=[Andi, Mike], true=[Florian, Michael, Sebastian]}
```

Beispiel: Worthäufigkeitshistogramm

Bei der Besprechung der zustandslosen Intermediate Operations hatte ich im Rahmen der Ausführungen zur Methode `flatMap()` gezeigt, wie sich aus einer Eingabe, bestehend aus einer Menge von Sätzen, die Häufigkeit bestimmter Wörter ermitteln lässt. Dort waren wir so weit gekommen, alle Wörter sortiert hintereinander ausgeben zu können. Mit den gerade vorgestellten Methoden zur Gruppierung wollen wir nun ein Worthäufigkeitshistogramm aufbereiten. Im nachfolgenden Listing sind zur Erinnerung einige der relevanten Zeilen nochmals abgebildet, die um die Gruppierung und die case-insensitive Sortierung der Schlüssel ergänzt wurden:

```
public static void main(final String[] args) throws IOException
{
    final Path exampleFile = Paths.get("src/main/resources/" +
                                       "ch07_stream_api/Example.txt");

   // Datei einlesen neu in JDK 8
   final List<String> contents = Files.readAllLines(exampleFile);

   // ...

   // Filterung basierend auf den Prädikaten
   final Stream<String> filteredContents = words.map(String::trim).
                                                 filter(notIsShortWord).
                                                 filter(isSignificantWord);

   // Mappings zum Satzzeichen entfernen
   final Function<String, String> removePunctationMarks = str ->
   {
      if (str.endsWith(".") || str.endsWith(":") || str.endsWith("!"))
      {
         return str.substring(0, str.length()-1);
      }
      return str;
   };

   final Stream<String> mapped = filteredContents.map(removePunctationMarks);

   // Gruppierung
   Function<String, String> identity = Function.identity(); // str -> str;
   Map<String, Long> grouped = mapped.collect(groupingBy(identity, counting()));

   // Sortierung der Schlüssel mithilfe einer TreeMap<K,V>
   Map<String, Long> sorted = new TreeMap<>(String.CASE_INSENSITIVE_ORDER);
   sorted.putAll(grouped);

   System.out.println(sorted);
}
```

Listing 7.16 *Ausführbar als* **'FLATMAPANDGROUPINGEXAMPLE'**

Das Programm FLATMAPANDGROUPINGEXAMPLE erzeugt folgende Ausgabe:

```
{contains=1, first=1, goodbye=1, Last=1, line=4, second=1, some=1,
text=3, Third=1}
```

7.1.6 Wissenswertes zur Parallelverarbeitung

Eingangs erwähnte ich, dass Streams als sequenzielle oder parallele Variante erzeugt werden können. Besonders interessant ist, dass man sich zu Beginn der Verarbeitung nicht darauf festlegen muss, wie alle Schritte abgearbeitet werden sollen. Es scheint vielmehr möglich, beliebig zwischen paralleler und sequenzieller Abarbeitung hin und her zu schalten, wie es das nachfolgende Beispiel zeigt:

```
Stream<String> adults = persons.parallelStream().filter(Person::isAdult).
                                sequential().map(Person::getName).
                                parallel().filter(str -> str.startsWith("Mi")).
                                sequential().map(String::toUpperCase);
```

Doch stimmt das wirklich? Nein, tatsächlich ist das Ganze doch etwas komplizierter. Trotzdem erahnt man an diesem Beispiel die Möglichkeiten zur Parallelisierung von Berechnungen, die sich durch die neuen Sprachfeatures ergeben. Besonders praktisch ist, dass die dahinter stehende Komplexität für den Entwickler verborgen bleibt. Man muss sich somit nicht um Details der Multithreading-Verarbeitung kümmern, sondern beschreibt die Abläufe auf einer höheren Abstraktionsebene.

Schauen wir uns das Ganze mal genauer an. Tatsächlich werden nicht alle Elemente für jeden Verarbeitungsschritt, sondern pro Element jeweils die Verarbeitungsschritte des Fließbands hintereinander abgearbeitet – auf Wunsch kann das sogar parallel erfolgen. Der Sachverhalt und die Vorgänge bei der Stream-Verarbeitung lassen sich am besten visuell verdeutlichen (vgl. Abbildung 7-1).

Abbildung 7-1 *Pipeline von Intermediate Operations*

Bei der Abarbeitung kann man sich vorstellen, dass die Elemente eines Streams hintereinander aufgereiht sind und dann die Aktionen ausgeführt werden, wie dies in Abbildung 7-1 dargestellt ist. Die Elemente werden jeweils für sich durch die einzelnen

Schritte der Pipeline geschickt. Diese Verarbeitung bildet die Grundlage für eine mögliche Parallelisierbarkeit.

Abläufe bei der Verarbeitung mit zustandslosen Operationen

Im Beispiel zur Methode `peek()` habe ich auf Seite 463 die elementweise Verarbeitung angedeutet. Anhand des dort begonnenen und hier erweiterten Beispiels möchte ich die Abläufe bei der Verarbeitung einzelner Elemente in Streams verdeutlichen. Dazu fügen wir in jeden Verarbeitungsschritt eine Konsolenausgabe wie folgt ein:

```
public static void main(final String[] args)
{
  final List<Person> persons = createDemoData();
  final Stream<String> allMikes = persons.stream().
    filter(person ->
    {
        final boolean ok = person.isAdult();
        System.out.println("Step 1 filter: " + person + " / isAdult(): " + ok);
        return ok;
    }).
    map(person ->
    {
        final String name = person.getName();
        System.out.println("Step 2 map: " + person + " -> " + name);
        return name;
    }).
    filter(name ->
    {
        final boolean ok = name.startsWith("Mi");
        System.out.println("Step 3 filter: " + name + " / startsWith('Mi'): " +
              ok);
        return ok;
    }).
    map(name ->
    {
        final String upperName = name.toUpperCase();
        System.out.println("Step 4 map: " + name + " -> " + upperName);
        return upperName;
    });

  System.out.println("Protokollierung jedes Schritts beim Filtern nach 'Mi':");
  allMikes.forEach(System.out::println);     // Löst die Verarbeitung aus
}

private static List<Person> createDemoData()
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Michael", 43));
    persons.add(new Person("Max", 5));
    persons.add(new Person("Moritz", 7));
    persons.add(new Person("Merten", 38));
    persons.add(new Person("Micha", 42));
    return persons;
}
```

Listing 7.17 *Ausführbar als* '**STREAMPROCESSINGEXAMPLE**'

Startet man das Programm STREAMPROCESSINGEXAMPLE, so kommt es zu folgenden Ausgaben, die die elementweise Abarbeitung nachvollziehbar machen:

```
Protokollierung jedes Schritts beim Filtern:
Step 1 filter: Person [name=Michael, age=43, gender=MALE] / isAdult(): true
Step 2 map:    Person [name=Michael, age=43, gender=MALE] -> Michael
Step 3 filter: Michael / startsWith('Mi'): true
Step 4 map:    Michael -> MICHAEL
MICHAEL
Step 1 filter: Person [name=Max, age=5, gender=MALE] / isAdult(): false
Step 1 filter: Person [name=Moritz, age=7, gender=MALE] / isAdult(): false
Step 1 filter: Person [name=Merten, age=38, gender=MALE] / isAdult(): true
Step 2 map:    Person [name=Merten, age=38, gender=MALE] -> Merten
Step 3 filter: Merten / startsWith('Mi'): false
Step 1 filter: Person [name=Micha, age=42, gender=MALE] / isAdult(): true
Step 2 map:    Person [name=Micha, age=42, gender=MALE] -> Micha
Step 3 filter: Micha / startsWith('Mi'): true
Step 4 map:    Micha -> MICHA
MICHA
```

Anhand der vorherigen Grafik und der Programmausgabe erkennt man sehr schön die elementweise Verarbeitung. Welchen Vorteil bringt das?

Vorteil der elementweisen Verarbeitung Diese Art der Abarbeitung ist einfach parallelisierbar, insbesondere für die bislang genutzten voneinander unabhängigen, zustandslosen Operationen: Für diese lassen sich beliebige Elemente zu Gruppen zusammenfassen, die jeweils parallel durch eigene Threads verarbeitet werden.

Wenn Sie das nachvollziehen wollen, dann ergänzen Sie doch einen Aufruf von `parallel()`, wie es nachfolgend angedeutet ist – der Rest des Programms ist analog wie zuvor und die weiteren Schritte sind hier verkürzt dargestellt:

```
public static void main(final String[] args)
{
   final List<Person> persons = createDemoData();

    final Stream<String> allMikes = persons.stream().parallel();

    allMikes.filter(person -> ...).
             map(person -> ... ).
             filter(name -> ... ).
             map(name -> ... );

    System.out.println("Protokollierung jedes Schritts beim Filtern nach 'Mi':")
          ;
    allMikes.forEach(System.out::println);
}
```

Listing 7.18 *Ausführbar als* '**STREAMPARALLELPROCESSINGEXAMPLE**'

Wenn Sie das Programm STREAMPARALLELPROCESSINGEXAMPLE starten, so erhalten Sie eine Ausgabe ähnlich zu folgender:

```
Protokollierung jedes Schritts beim Filtern nach 'Mi':
Step 1 filter: Person [name=Moritz, age=7, gender=MALE] / isAdult(): false
Step 1 filter: Person [name=Micha, age=42, gender=MALE] / isAdult(): true
Step 1 filter: Person [name=Merten, age=38, gender=MALE] / isAdult(): true
Step 2 map:    Person [name=Merten, age=38, gender=MALE] -> Merten
Step 3 filter: Merten / startsWith('Mi'): false
Step 1 filter: Person [name=Michael, age=43, gender=MALE] / isAdult(): true
Step 2 map:    Person [name=Michael, age=43, gender=MALE] -> Michael
Step 3 filter: Michael / startsWith('Mi'): true
Step 4 map:    Michael -> MICHAEL
Step 1 filter: Person [name=Max, age=5, gender=MALE] / isAdult(): false
Step 2 map:    Person [name=Micha, age=42, gender=MALE] -> Micha
Step 3 filter: Micha / startsWith('Mi'): true
Step 4 map:    Micha -> MICHA
MICHAEL
MICHA
```

An den Ausgaben erkennt man, dass die einzelnen Abarbeitungen durch die Parallelisierung nahezu beliebig durcheinandergewürfelt sind. Selbstverständlich wird für jedes Element einzeln sichergestellt, dass die Reihenfolge der Verarbeitungsschritte derjenigen der Stream-Methoden im Programm entspricht.

Abläufe bei der Verarbeitung mit zustandsbehafteten Operationen

Auch für die zustandsbehafteten Operationen kann man eine recht effiziente Parallelisierung erreichen. Dazu werden die Elemente zunächst in eine Vielzahl von Teilbereichen aufgeteilt und wenn diese klein genug sind, erfolgt die konkrete Verarbeitung mithilfe mehrerer Threads. Schließlich werden die einzelnen Zwischenergebnisse zusammengefasst. In Abbildung 7-2 ist das exemplarisch gezeigt.

Abbildung 7-2 *Schematische Darstellung der parallelen Summierung bzw. Sortierung*

Besonderheit bei `parallelStream()` und `forEach()`

Wenn Sie eine Verarbeitung parallel ausgeführt haben, dann entsprechen die Ausgaben über `forEach(Consumer<? super T>)` teilweise nicht dem, was Sie erwarten: Die Reihenfolge der Ausgabe ist oftmals ungeordnet.

Für den Aufruf des Sortierens kann man den geschilderten Sachverhalt gut nachvollziehen. Betrachten wir zur Verdeutlichung eine Liste von Namen:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike");
    names.parallelStream().sorted().forEach(System.out::println);
}
```

Listing 7.19 *Ausführbar als* **'WRONGPARALLELFOREACHEXAMPLE'**

Das Programm WRONGPARALLELFOREACHEXAMPLE produziert – obwohl explizit `sorted()` zur Sortierung aufgerufen wurde – manchmal ungeordnete Ausgaben:

```
Ralph
Andi
Stefan
Mike
```

Führt man das Programm mehrmals aus, so erkennt man die zufällige Reihenfolge. Um die korrekte Reihenfolge bei Parallelverarbeitung für die abschließende Terminal Operation, hier die Konsolenausgabe, sicherzustellen, muss man Aufrufe der Methode `forEachOrdered(Consumer<? super T>)` wie folgt nutzen:

```
final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike");
names.parallelStream().sorted().forEachOrdered(System.out::println);
```

Besonderheit bei `parallel()` und `findAny()`

Wir sind durch das vorangegangene Beispiel schon ein wenig sensibilisiert, was die Parallelisierung von Aktionen und mögliche (unerwartete) Resultate angeht. Um die Unvorhersagbarkeit nochmals zu verdeutlichen, zeige ich hier eine weitere Variante einer möglicherweise überraschenden Parallelverarbeitung:

```
public static void main(final String[] args)
{
    IntStream.range(0, 10_000).filter(i -> i % 2 == 0).
                               parallel().findAny().
                               ifPresent(System.out::println);
}
```

Listing 7.20 *Ausführbar als* **'PARALLELFINDANYEXAMPLE'**

Startet man das Programm PARALLELFINDANYEXAMPLE einmal mit einem Aufruf von `parallel()` und einmal ohne, so erhält man unterschiedliche Ausgaben, die sogar

mitunter von Ausführung zu Ausführung sowie von Rechner zu Rechner variieren – nur die sequenzielle Variante ist deterministisch und produziert immer die gleiche Ausgabe:

```
sequenzielle Ausführung: 0, 0, 0, ...
parallele Ausführung:    6562, 8126, 1562 usw.
```

Bei `findAny()` ist die nicht deterministische Ausrichtung gewollt, weil es hier um die Prüfung geht, ob es überhaupt ein Element mit den gewünschten Eigenschaften gibt. Für ein stabiles Ergebnis muss man `findFirst()` aufrufen.

Anmerkung: Sinnhaftigkeit der »Umschalterei«

Eine Umschaltung zwischen sequenzieller und paralleler Verarbeitung zwischen einzelnen Schritten des Fließbands ist aufgrund der zuvor gezeigten mit Streams realisierten Abarbeitung nicht möglich – variierende Aufrufe an `sequential()` und `parallel()` aber schon. Allerdings entscheidet der letzte Aufruf vor der Terminal Operation, in welchem Modus (sequenziell oder parallel) die Abarbeitung des Streams erfolgen wird.

7.1.7 Neuerungen im Stream-API in JDK 9

Das Stream-API mit dem Interface `Stream<T>` stellt eine der wesentlichen Neuerungen in Java 8 dar. Streams besaßen bereits von Beginn an ein recht umfangreiches API. Dieses wurde mit Java 9 nochmals leicht erweitert. Zunächst schauen wir uns folgende zwei neuen Methoden an:

- `takeWhile(Predicate<T>)` – Verarbeitet Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.
- `dropWhile(Predicate<T>)` – Überspringt Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.

Diese Ergänzungen findet man analog auch in den für die primitiven Typen `int`, `long` und `double` spezialisierten Stream-Klassen `IntStream`, `LongStream` sowie `DoubleStream`. Dort ist dann jeweils das Prädikat auf den korrespondierenden Typ angepasst, etwa `takeWhile(IntPredicate)`.

Beispiel für die Methoden `takeWhile()` und `dropWhile()`

Zur Demonstration der Arbeitsweise der Methode `takeWhile()` wird ein unendlicher Stream von Ganzzahlen durch Aufruf von `iterate()` auf einem `IntStream` erzeugt, der mit der Zahl 1 beginnt. Für `dropWhile()` nutzen wir einen Stream mit dem vordefinierten Wertebereich von 7 bis 14, den wir durch Aufruf von `rangeClosed()` konstruieren. Zur Darstellung einer Besonderheit bei der Verarbeitung mit `dropWhile()` verwenden wir schließlich einen Stream mit vordefinierten Werten, der durch einen Aufruf von `of()` erzeugt wird:

```
public static void main(final String[] args)
{
    System.out.println("takeWhile");
    final IntStream stream1 = IntStream.iterate(1, n -> n + 1);
    System.out.println(stream1.takeWhile(n -> n < 10).
                               mapToObj(Integer::toString).
                               collect(joining(", ")));

    System.out.println("\ndropWhile 1");
    final IntStream stream2 = IntStream.rangeClosed(7, 14);
    System.out.println(stream2.dropWhile(n -> n < 10).
                               mapToObj(Integer::toString).
                               collect(joining(", ")));

    System.out.println("\ndropWhile 2");
    final IntStream stream3 = IntStream.of(7,9,11,13,15,5,3,1);
    System.out.println(stream3.dropWhile(n -> n < 10).
                               mapToObj(Integer::toString).
                               collect(joining(", ")));
}
```

Listing 7.21 *Ausführbar als* '**STREAMSEXAMPLE**'

Für alle Streams konvertieren wir durch den Aufruf von `mapToObj(Integer::toString)`[3] die Zahlen in einen String und bereiten mit `collect(joining(", "))` eine kommaseparierte Darstellung auf – die Methode `joining()` stammt aus der Klasse `Collectors` (vgl. Abschnitt 7.1.5) und wurde zur besseren Lesbarkeit statisch importiert. Mit diesem Wissen ist leicht nachvollziehbar, dass es zu den folgenden Ausgaben kommt, wenn man das Programm STREAMSEXAMPLE startet:

```
takeWhile
1, 2, 3, 4, 5, 6, 7, 8, 9

dropWhile 1
10, 11, 12, 13, 14

dropWhile 2
11, 13, 15, 5, 3, 1
```

Das Beispiel `dropWhile 2` verdeutlicht, dass bei Aufrufen von `dropWhile()` nur zu Beginn die Einhaltung der Bedingung überprüft wird. Gilt diese einmal, so erfolgt danach keine weitere Prüfung und es werden im Anschluss möglicherweise Elemente konsumiert, die gegen die angegebene Bedingung verstoßen. Dieser Fall kann für `takeWhile()` so nicht auftreten, da dort die Verarbeitung sofort abgebrochen würde.

[3] Alternativ wäre natürlich auch der Einsatz des Lambda-Ausdrucks `num -> "" + num` möglich gewesen, der jedoch die Intention der Konvertierung in einen String nicht so deutlich macht wie die Methodenreferenz `Integer::toString`.

Beide Methoden in Kombination Auch in Kombination lassen sich die beiden Methoden sinnvoll einsetzen. Das gilt etwa immer dann, wenn zunächst Informationen so lange aussortiert werden sollen, bis diese einem gewissen Gütekriterium oder Wert entsprechen, und dann im Anschluss so lange gelesen werden sollen, bis eine Abbruchbedingung erfüllt ist.

Als Beispiel werden die Informationen aus einen `Stream<String>` extrahiert, die zwischen den Markierungen `<START>` und `<END>` liegen:

```
public static void main(final String[] args)
{
    Stream<String> words = Stream.of("ab", "bla", "<START>",
                                     "Hier", "steht", "der", "Text",
                                     "<END>", "saas", "bla");

    Stream<String> content = words.dropWhile(word -> !word.equals("<START>"))
                                  .skip(1)
                                  .takeWhile(word -> !word.equals("<END>"));

    content.forEach(System.out::println);
}
```

Listing 7.22 *Ausführbar als* '**DROPANDTAKEWHILEEXAMPLE**'

Das `skip(1)` ist nötig, um den Begrenzer `<START>` nicht mit in die Ergebnisliste aufzunehmen. Auf ähnliche Weise könnte man übrigens auch die Header- oder Body-Informationen eines HTML-Dokuments extrahieren.

Startet man das Programm DROPANDTAKEWHILEEXAMPLE, so sieht man sehr schön die Extraktion:

```
Hier
steht
der
Text
```

Weitere Methoden

Neben den beiden Methoden `takeWhile()` und `dropWhile()` findet man für Streams folgende Neuerungen:

- `ofNullable(T)` – Liefert einen `Stream<T>` mit einem Element, sofern das übergebene Element ungleich `null` ist. Ansonsten wird ein leerer Stream erzeugt.
- `iterate(T, Predicate<? super T>, UnaryOperator<T>)` – Es wird ein `Stream<T>` mit dem als ersten Parameter übergebenen Startwert erzeugt. Die folgenden Werte werden durch den `UnaryOperator` berechnet. Im Gegensatz zu der bereits mit JDK 8 existierenden Methode `iterate(T, UnaryOperator<T>)` wird hierbei auch noch das übergebene `Predicate<T>` geprüft und die Erzeugung gestoppt, sobald dieses nicht mehr erfüllt ist.

Für die zweite Methode möchte ich ein Beispiel präsentieren. Zuvor haben wir zur Ausgabe der Zahlen von 1 bis 9 ein `IntPredicate` und die Methode `takeWhile()` mit einer Prüfung von `n -> n < 10` vorgenommen. Stattdessen können wir diese Prüfung auch direkt im Aufruf von `iterate()` durchführen. Ausgehend vom Wert 1 erzeugen wir mit dem Lambda `n -> n + 1` als `IntUnaryOperator` eine aufsteigende Zahlenfolge, deren Ende der Berechnung über die Bedingung `n -> n < 10` gesteuert wird.

```
public static void main(final String[] args)
{
    // iterate() unterstützt seit JDK 9 eine Bedingung
    System.out.println("iterate with predicate");
    final IntStream stream = IntStream.iterate(1, n -> n < 10, n -> n + 1);
    System.out.println(stream.mapToObj(Integer::toString).
                              collect(joining(", ")));
}
```

Listing 7.23 *Ausführbar als* '**STREAMSITERATEEXAMPLE**'

Das Programm STREAMSITERATEEXAMPLE gibt erwartungsgemäß Folgendes aus:

```
iterate with predicate
1, 2, 3, 4, 5, 6, 7, 8, 9
```

Die Methode `iterate()` bietet damit eine ähnliche Funktionalität wie eine klassische `for`-Schleife, allerdings mit dem Unterschied, dass die Aktionen im Stream lazy, also erst durch eine Terminal Operation wie etwa das obige `collect()`, ausgeführt werden.

7.1.8 Neuerungen im Stream-API in JDK 10

Das Stream-API war schon bei seiner Einführung mit Java 8 recht umfangreich und wurde mit Java 9 nochmals erweitert. Allerdings ließen sich bislang keine unveränderlichen Datenstrukturen erzeugen. Dies wird seit Java 10 durch die Methoden `toUnmodifiableList()`, `toUnmodifiableSet()` und `toUnmodifiableMap()` der Klasse `java.util.stream.Collectors` möglich. Exemplarisch schauen wir uns ein Beispiel für Listen und Sets an, für Maps funktioniert das Ganze analog.

Beispiel: Unveränderliche Collections aus Streams erzeugen

Basierend auf einer Liste von Strings, deren Inhalt teils aus doppelten Namen besteht, wollen wir eine unveränderliche Kopie als Liste bzw. als Set erzeugen. Die oben genannten Methoden haben wir der besseren Lesbarkeit halber statisch importiert:

```
var names = List.of("Tim", "Tom", "Mike", "Peter", "Tom", "Tim");

var immutableNames = names.stream().collect(toUnmodifiableList());
System.out.println("immutableNames type: " + immutableNames.getClass());

var uniqueImmutableNames = names.stream().collect(toUnmodifiableSet());
System.out.println("uniqueImmutableNames content: " +
                   uniqueImmutableNames);
System.out.println("uniqueImmutableNames type: " +
                   uniqueImmutableNames.getClass());
```

Diese Zeilen geben Folgendes aus:

```
immutableNames type: class java.util.ImmutableCollections$ListN
uniqueImmutableNames content: [Peter, Mike, Tim, Tom]
uniqueImmutableNames type: class java.util.ImmutableCollections$SetN
```

Beachten Sie bitte, dass die Reihenfolge der Elemente bei Ihnen durchaus abweichen kann – diese ist durch `toUnmodifiableSet()` und die dadurch entstehende Instanz der inneren Klasse `SetN` aus `ImmutableCollections` nicht festgelegt.

Hinweis Interessanterweise verhält sich hier der Kollektor `toUnmodifiableSet()` anders als die Methode `of()` aus dem Interface `Set<E>`: Diese würde nämlich bei Duplikaten eine Exception auslösen. Das geschieht hier für die Duplikation von »Tim« und »Tom« nicht.

7.2 Filter-Map-Reduce

Wir haben uns mittlerweile so viel Grundlagenwissen zu Streams erarbeitet, dass uns nun das Verständnis der mächtigen neuen Filter-Map-Reduce-Funktionalität – einer speziellen Untermenge des Stream-APIs – recht leicht fallen sollte.

Aufgabenstellung: Filtere eine Liste und extrahiere Daten

Nehmen wir an, unsere Aufgabe bestünde darin, eine Liste von Personen zu filtern, dabei alle im Juli Geborenen zu ermitteln und deren Namen kommasepariert auszugeben. Gegeben sei dazu folgende `List<Person>` als Eingabe:[4]

```
private static final List<Person> persons = Arrays.asList(
                        new Person("Stefan", LocalDate.of(1971, MAY, 12)),
                        new Person("Micha", LocalDate.of(1971, FEBRUARY, 7)),
                        new Person("Andi Bubolz", LocalDate.of(1968, JULY, 17)),
                        new Person("Andi Steffen", LocalDate.of(1970, JULY, 17)),
                        new Person("Merten", LocalDate.of(1975, JUNE, 16)));
```

Die Aufgabe lässt sich in folgende drei Schritte untergliedern:

1. Filtere auf alle im Juli Geborenen
2. Extrahiere ein Attribut, im Beispiel den Namen
3. Bereite eine kommaseparierte Liste auf

[4] Aufmerksamen Lesern fallen die hier genutzten Typen `LocalDate` und `Month` und deren Konstanten, etwa `MAY` und `JULY`, auf. Diese werden später in Kapitel 8 genauer erläutert.

7.2.1 Herkömmliche Realisierung

Bevor wir uns die mit JDK 8 bereitgestellte Filter-Map-Reduce-Funktionalität anschauen, werfen wir einen Blick darauf, wie man so etwas mit JDK 7 realisiert hätte. Dort müsste man die Funktionalität der Filterung und Extraktion selbst programmieren. Der Übersichtlichkeit halber werden die einzelnen Schritte in Form kurzer Methoden realisiert. Zum besseren Verständnis beginnen wir mit der Implementierung einer `main()`-Methode, die drei Schritte ausführt:

```
public static void main(final String[] args)
{
   final List<Person> persons = createDemoPersons();

    // Schritt 1: Filtere
    final List<Person> bornInJuly = filterByMonth(persons, Month.JULY);

    // Schritt 2: Extrahiere
    final List<String> names = extractNameAttribute(bornInJuly);

    // Schritt 3: Bereite Ergebnis auf
    System.out.println(joinStrings(names, ", "));
}
```

Listing 7.24 *Ausführbar als* '**FILTERMAPREDUCEOLDSTYLEEXAMPLE**'

Filter: Filtere auf alle im Juli Geborenen

Wir konstruieren eine Ergebnisliste `filteredPersons` und fügen dort diejenigen `Person`-Objekte hinzu, die das Kriterium »Geboren im Juli« erfüllen:

```
static List<Person> filterByMonth(final List<Person> persons, final Month month)
{
    final List<Person> filteredPersons = new ArrayList<>();
    for (final Person person : persons)
    {
        if (person.getBirthday().getMonth() == month))
        {
            filteredPersons.add(person);
        }
    }
    return filteredPersons;
}
```

Bitte beachten Sie, dass bei der Abfrage der Attribute keine `null`-Prüfungen notwendig sind, weil die Klasse `Person` im Konstruktor die Gültigkeit der Attribute sicherstellt.

Map: Extrahiere ein Attribut

Danach nehmen wir eine ***Extraktion*** der Daten vor. Die Namen der Personen werden zunächst ausgelesen und als `List<String>` bereitgestellt:

```
static List<String> extractNameAttribute(final List<Person> persons)
{
    final List<String> names = new ArrayList<>();
    for (final Person person : persons)
    {
        names.add(person.getName());
    }
    return names;
}
```

Reduce: Bereite eine kommaseparierte Liste auf

Zur Aufbereitung der Ausgabe durchlaufen wir die als Parameter übergebene Liste und fügen jedes Element gefolgt von einem Komma (mit Ausnahme des letzten) in ein `StringBuilder`-Objekt per `append()` ein:

```
static String joinStrings(final List<String> names, final String delimiter)
{
    final StringBuilder sb = new StringBuilder();

    final Iterator<String> it = names.iterator();
    while (it.hasNext())
    {
        sb.append(it.next());
        if (it.hasNext())
            sb.append(delimiter);
    }
    return sb.toString();
}
```

Führen wir die realisierte Funktionalität – wie zuvor in `main()` gezeigt – aus, so werden die beiden im Juli geborenen Personen ausgegeben:

```
Andi Bubolz, Andi Steffen
```

Wenn man die Lösung betrachtet, so fällt negativ auf, dass diese recht lang ist. Erst bei etwas genauerem Überlegen bemerkt man einen weiteren Nachteil: Die Abarbeitung erfolgt sequenziell und zudem erhöht sich die Laufzeit linear zur Anzahl gespeicherter Personen. Als Alternative kann man die gesamte Funktionalität innerhalb nur einer Methode und ineinander verschränkt realisieren. Das wird zwar minimal schneller, jedoch deutlich unübersichtlicher – insbesondere, wenn die Abfragen komplexer werden. Auch kann man Erweiterungen kaum realisieren und die fehlende Kombinierbarkeit widerspricht dem Gedanken der Orthogonalität. Denken wir beispielsweise an die Filterung nach einem speziellen Kriterium, an die Beschränkung auf eine gewünschte Ergebnisgröße oder gar an die Kombination von beidem. Schauen wir uns jetzt eine mit Java 8 mögliche Variante an, die derartige Erweiterungen ohne viel Mühe erlaubt.

7.2.2 Filter-Map-Reduce mit JDK 8

Mit JDK 8 wird eine Filter-Map-Reduce-Funktionalität in Java bereitgestellt, die stark vom Einsatz von Lambdas profitiert. Dabei stehen die drei Begriffe Filter, Map und Reduce für die bereits im vorangegangenen Beispiel kennengelernten Aktionen:

- **Filter** – Aus einer Ausgangsmenge von Objekten werden diejenigen herausgefiltert bzw. genauer gesagt behalten, die festgelegten Anforderungen entsprechen.
- **Map** – Mit Mapping oder Projektion ist der Vorgang gemeint, der aus einem Objekt gewisse Informationen ableitet und diese in einer gewünschten Form aufbereitet. Map beschreibt eine Projektion, d. h. eine Transformation eines Elements in eine andere Repräsentation (wobei die Anzahl Elemente im Stream gleich bleibt).
- **Reduce** – Schlussendlich sollen die Berechnungsergebnisse verarbeitet werden, etwa auf der Konsole ausgegeben oder als Ergebnismenge in einer Collection aufbereitet werden. Reduce beschreibt somit das Zusammenfassen zu einem Resultat.

Diese Schritte sind in der nachfolgenden Abbildung visualisiert:

Abbildung 7-3 *Symbolische Darstellung von Filter-Map-Reduce*

Filter-Map-Reduce im Einsatz

Wir machen uns jetzt daran, die drei Verarbeitungsschritte zur Filterung mithilfe des Filter-Map-Reduce-Frameworks und von Lambdas zu implementieren. Die benötigten Grundlagen haben wir uns zuvor bei der Besprechung der Intermediate und Terminal Operations erarbeitet. Dieses Wissen müssen wir nur noch geeignet kombinieren:

```
final Predicate<Person> isBornInJuly =
                        person -> person.birthday.getMonth() == Month.JULY;

final String bornInJuly = persons.stream().
                                  filter(isBornInJuly)).
                                  map(Person::getName).
                                  reduce("", stringCombiner);
```

Die beiden Schritte Filter und Map sind intuitiv verständlich, beim Reduce-Schritt bedienen wir uns eines Tricks. Hierbei wird die nachfolgend gezeigte Realisierung des funktionalen Interface `BinaryOperator<T>` namens `stringCombiner` genutzt. Dort wird aus zwei Eingaben vom Typ `T` ein Ergebnis vom Typ `T` berechnet. Die Kombination zweier Strings können wir wie folgt – schon leicht unleserlich – implementieren:

```
final BinaryOperator<String> stringCombiner = (str1, str2) ->
{
    return str1.isEmpty() ? str2 : str1 + ", " + str2;
};
```

Reduce: Zusammenfassen von Werten mit `collect()`

Das Zusammenfassen von Einträgen ist ein gebräuchlicher Anwendungsfall. Daher bietet das JDK 8 eine spezielle Form einer Reduce-Methode namens `collect()`, die wir bereits kurz im Rahmen der Terminal Operations in Abschnitt 7.1.5 besprochen haben. Vorgefertigte Implementierungen der dort benötigten `Collector<T,A,R>`-Instanzen finden sich in der Utility-Klasse `Collectors`. Wir wählen hier `joining(String)` zur Kombination von Strings wie folgt:

```
public static void main(final String[] args)
{
    final List<Person> persons = createDemoPersons();

    final Predicate<Person> isBornInJuly =
                            person -> person.birthday.getMonth() == Month.JULY;

    final String bornInJuly = persons.stream().
                                      filter(isBornInJuly).
                                      map(person -> person.getName()).
                                      collect(Collectors.joining(", "));

    System.out.println(bornInJuly);
}
```

Listing 7.25 *Ausführbar als* **'FILTERMAPREDUCEJDK8EXAMPLE'**

Diese Form ist viel stärker am zu lösenden Problem ausgerichtet als die vorherige, unter anderem weil der Algorithmus zum Filtern nicht selbst programmiert wird. Abweichend von der Realisierung zuvor können wir für die `enum`-Aufzählung `Month` den Vergleich hier sogar noch etwas besser lesbar mit `==` anstelle eines Aufrufs von `equals(Object)` schreiben. Auch wird Innerhalb von `map()` nun ein Lambda statt einer Methodenreferenz genutzt. Außerdem versteckt `joining()` die Spezialbehandlung der korrekten Aufbereitung einer kommaseparierten Ausgabe.

Führen wir das Programm FILTERMAPREDUCEJDK8EXAMPLE aus, so werden – wie zu erwarten – die beiden im Juli geborenen Personen ausgegeben:

```
Andi Bubolz, Andi Steffen
```

Aufbereitung und Verknüpfung nicht textueller Werte Gerade haben wir gesehen, wie einfach die kommaseparierte Aufbereitung von textuellen Werten mithilfe von Lambdas und den Neuerungen im Stream-API implementiert werden kann.

Etwas mehr Aufwand ist erforderlich, wenn man Zahlen oder Objekte auf diese Weise miteinander verknüpfen möchte und z. B. die Altersangaben von Personen kommasepariert aufbereitet. Dazu müssen wir die Objekte, hier vom Typ `Integer`, in Strings wandeln, bevor wir sie mit `joining()` verknüpfen können. Die Umwandlung kann explizit durch einen Aufruf von `toString()` erfolgen oder aber implizit durch die Notation `"" + value`. Im folgenden Listing sind beide Varianten gezeigt:

```
// Explizite Umwandlung / Mapping mit toString
final String joined1 = persons.stream().mapToInt(Person::getAge)
                                       .mapToObj(Integer::toString)
                                       .collect(joining(", "));

// Implizite Umwandlung / Mapping durch "" + value
final String joined2 = persons.stream().map(person -> "" + person.getAge())
                                       .collect(joining(", "));
```

Hinweis: Die Methode `String.join()`

Wenn lediglich textuelle Werte miteinander verknüpft werden sollen und dabei nur eine Verknüpfungszeichenfolge, aber keine Start- und Endkennungen benötigt werden, dann kann man die in der Klasse `String` mit JDK 8 eingeführte Methode `join(CharSequence delimiter, CharSequence... elements)` nutzen:[a]

```
final String stringConcat = String.join(", ", names);
```

Etwas unglücklich empfinde ich die Signatur, in der der Delimiter vor den zu verknüpfenden Elementen angegeben werden muss. Das liegt aber daran, dass Varargs in Java nur für den letzten Parameter in einer Signatur auftreten dürfen.

[a] Wir haben in Abschnitt 7.2.1 mit `joinStrings()` eine ähnliche Methode entworfen.

7.3 Praxisbeispiele

In diesem Unterkapitel wollen wir das bisher Gelernte vertiefen. Dazu schauen wir uns zwei Praxisbeispiele an. Zum einen wollen wir Elemente gruppieren und dann daraus ein Histogramm erstellen, also eine Häufigkeitsverteilung aufbereiten. Zum anderen zeige ich, wie man die Daten in einer Map auch nach Wert sortieren kann.

Auf Folgendes sei noch hingewiesen: Diverse nützliche Methoden aus der Klasse `Collectors` wie `groupingBy()` oder `counting()` importieren wir der Lesbarkeit halber statisch.

7.3.1 Aufbereiten von Gruppierungen und Histogrammen

Zur Erinnerung und als Vorbereitung wollen wir uns noch mal kurz dem Gruppieren von Elementen widmen. Dazu definieren wir ein `String[]` mit einigen Namen. Zur Gruppierung nach Anfangsbuchstaben bietet sich die Methode `groupingBy()` sowie eine Gruppierungsfunktion `toFirstChar` an:

```
public class GroupingExample
{
    public static void main(final String[] args)
    {
       final String[] names = { "Tom", "Tim", "Mike", "Kay", "Micha", "Kai" };

       final Function<String, String> toFirstChar =
                                      str -> str.substring(0, 1).toUpperCase();

       var result = Stream.of(names).
                          collect(Collectors.groupingBy(toFirstChar));
       System.out.println(result);
    }
}
```

Listing 7.26 *Ausführbar als* **'GROUPINGEXAMPLE'**

Starten wir das Programm GROUPINGEXAMPLE, so kommt es zu folgender Ausgabe:

```
{T=[Tom, Tim], K=[Kay, Kai], M=[Mike, Micha]}
```

Histogramm erzeugen

Zur Aufbereitung eines Histogramms unabhängig von der Groß- und Kleinschreibung nutzen wir neben `toLowerCase()` insbesondere die Möglichkeit, bei `groupingBy()` einen Aufruf von `counting()` auszuführen:

```
public static void main(final String[] args)
{
   final Stream<String> names =
         Stream.of("tom", "Tim", "Mike", "Kai", "mike", "kai", "Kai", "TOM");

   final Map<String, Long> histogram = histogram(names, String::toLowerCase);
   System.out.println(histogram);
}

public static <T> Map<T, Long> histogram(final Stream<T> stream,
                                         final Function<T, T> groupFunction)
{
    return stream.collect(groupingBy(groupFunction, counting()));
}
```

Listing 7.27 *Ausführbar als* **'HISTOGRAMEXAMPLE'**

Das Programm HISTOGRAMEXAMPLE erzeugt folgende Ausgabe:

```
{kai=3, tom=2, mike=2, tim=1}
```

7.3.2 Maps nach Wert sortieren

Wie eingangs schon angedeutet, möchte man manchmal Daten in Maps sortieren. Bei Sortierungen und Maps kommt einem eine `TreeMap<K,V>` in den Sinn. So verlockend deren Einsatz erscheint, ist dies jedoch nicht ausreichend, wenn man nach Wert und nicht nach Schlüssel sortieren möchte.

Sortieren mit speziellem Komparator nach Wert

Praktischerweise kann einer `TreeMap<K,V>` im Konstruktor ein `Comparator<T>` übergeben werden. Diesen implementieren wir zur Sortierung der Werte trickreich durch Aufruf von `get()` für die jeweiligen Schlüssel wie folgt:

```
// Achtung: Dieser Komparator ist bewusst nicht konsistent mit equals()
public static class DescendingValueComparator implements Comparator<String>
{
    final Map<String, Long> map;

    public DescendingValueComparator(final Map<String, Long> map)
    {
        this.map = map;
    }

    public int compare(final String first, final String second)
    {
        final long value1 = map.get(first);
        final long value2 = map.get(second);

        final int result = Long.compare(value2, value1);
        if (result == 0) // Der Wert 0 würde ungewünschte Ergebnisse liefern
        {
            return -1;
        }
        return result;
    }
}
```

Diese Realisierung sortiert dann die Map absteigend basierend auf den enthaltenen Werten und nicht nach Schlüsseln, benötigt dazu aber die ursprüngliche `Map<K,V>` und setzt zudem voraus, dass die Werte den Basistyp `Comparable<T>` besitzen, was hier für `Long` gegeben ist. In der Implementierung von `compare(String, String)` erfolgt zunächst ein Zugriff auf die Werte, die danach verglichen werden. Als Ergebnis wird der Wert des Aufrufs von `Long.compare()` zurückgeliefert. Dabei muss man beachten, dass bei Übereinstimmung der Werte explizit nicht der Wert 0 zurückgegeben werden darf, weil man sonst gleiche Werte fälschlicherweise zusammenfassen würde. Übrigens ist es unbedeutend, ob der Wert 0 auf den Wert 1 oder -1 abgebildet wird. Der Einsatz wird allerdings ein wenig unhandlich:

```
public static void main(final String[] args)
{
   final Map<String, Long> highscores = new TreeMap<>();
   highscores.put("Tim", 27371L);
   highscores.put("Andy", 20483L);
   highscores.put("Micha", 7271L);

   final Comparator<String> byValue = new DescendingValueComparator(highscores);
   final TreeMap<String, Long> sortedHighscores = new TreeMap<>(byValue);
   sortedHighscores.putAll(highscores);

   System.out.println(sortedHighscores);
}
```

Listing 7.28 *Ausführbar als* **'SORTBYVALUEEXAMPLE'**

Das Programm SORTBYVALUEEXAMPLE erzeugt folgende Ausgabe:

```
{Tim=27371, Andy=20483, Micha=7271}
```

Designverbesserung: Sortieren mit speziellem Komparator nach Wert

Mein Freund Andreas Schöneck wies mich auf eine vom Design her schönere Variante hin: Statt die Map im Komparator zu speichern, bietet es sich an, die Komparator-Klasse lokal innerhalb einer Methode `sortMapByValue(Map<K, V>)` zu definieren.[5] Zudem sollten wir noch eine generische Realisierung vornehmen. Das setzt man wie folgt um:

```
public static <K,V extends Comparable<V>>
              Map<K, V> sortMapByValue(final Map<K, V> map)
{
   class DescendingValueComparator implements Comparator<K>
   {
      public int compare(final K first, final K second)
      {
         final long value1 = map.get(first);
         final long value2 = map.get(second);

         final int result = Long.compare(value2, value1);
         if (result == 0) // Der Wert 0 würde ungewünschte Ergebnisse liefern
            return -1;

         return result;
      }
   }

   final Comparator<K> byValue = new DescendingValueComparator();
   final TreeMap<K, V> valueSortedMap = new TreeMap<>(byValue);
   valueSortedMap.putAll(map);
   return valueSortedMap;
}
```

[5] Die Idee entstammt der Seite `http://stackoverflow.com/questions/109383/how-to-sort-a-mapkey-value-on-the-values-in-java/22132422#22132422`.

Damit lässt sich die sortierte Map statt mit drei Zeilen mit nur einer erzeugen:

```
final Map<String, Long> sortedHighscores = sortMapByValue(highscores);
```

Diese Änderung ist im Programm SORTBYVALUEEXAMPLE2 enthalten. Dessen Start erzeugt (erwartungsgemäß) wiederum folgende Ausgabe:

```
{Tim=27371, Andy=20483, Micha=7271}
```

Zwischenbetrachtung Obwohl wir nun auch komplexere Sortierungen vornehmen können, so ist das Ganze doch ein wenig unschön in der Realisierung, da wir in einem `Comparator<String>` Vergleiche auf `Long` ausführen. Sinnvoller wäre es, auf den Einträgen der Map zu arbeiten, was wir uns nun anschauen wollen.

Alternative mit Java 8 und Streams

Tatsächlich lässt sich eine Sortierung nach Wert mit dem Stream-API und einer `LinkedHashMap<K,V>` recht einfach realisieren, allerdings wird das Ganze doch ein wenig kryptisch in der Lesbarkeit.

Für die Sortierung müssen wir entweder über einen Komparator, basierend auf `getValue()`, oder auf `Map.Entry.comparingByValue()` zurückgreifen:

```
public static <K,V extends Comparable<V>>
              Map<K, V> sortMapByValue(final Map<K, V> inputMap)
{
    final Comparator<Entry<K, V>> byValue = Map.Entry.comparingByValue();
    // Comparator.comparing(Entry::getValue)

    return inputMap.entrySet().stream()
                              .sorted(byValue.reversed())
                              .collect(LinkedHashMap::new,
                                       (map,entry) -> map.put(entry.getKey(),
                                                              entry.getValue()),
                                       Map::putAll);
}
```

Diese Änderung ist im Programm SORTBYVALUEEXAMPLE3 enthalten. Dessen Start erzeugt (erwartungsgemäß) folgende Ausgabe:

```
{Tim=27371, Andy=20483, Micha=7271}
```

Abschließende Verbesserungen des Komparators

Wenn man möchte, kann man nach den numerisch sortierten Werten zusätzlich auch die Schlüssel alphabetisch sortieren. Das ist immer dann praktisch, wenn es zu einem Wert mehrere Schlüssel gibt, etwa bei einer Statistik über die Anzahl an Methodenaufrufen die korrespondierenden Methodennamen.

Im nachfolgenden Listing realisieren wir eine kombinierte Sortierung erst absteigend nach Wert, dann alphabetisch nach Schlüssel:

```
public static <K extends Comparable<K>, V extends Comparable<V>>
              Map<K, V> sortMapByValue(final Map<K, V> inputMap)
{
    class DescendingValueComparator implements Comparator<Entry<K, V>>
    {
        public int compare(final Entry<K, V> first, final Entry<K, V> second)
        {
            // Achtung: Größte Werte zuerst, daher hier andere
            // Parameterreihenfolge
            final int result = second.getValue().compareTo(first.getValue());
            if (result == 0)
            {
                return first.getKey().compareTo(second.getKey());
            }
            return result;
        }
    }

    return inputMap.entrySet().stream()
                              .sorted(new DescendingValueComparator())
                              .collect(LinkedHashMap::new,
                                       (map,entry) -> map.put(entry.getKey(),
                                                              entry.getValue()),
                                       Map::putAll);
}
```

Damit wir die Auswirkungen feststellen können, müssen wir einige Einträge in unserer Map ergänzen, die gleiche Punktestände besitzen.

```
public static void main(final String[] args)
{
    final Map<String, Long> highscores = new TreeMap<>();
    highscores.put("Tim", 1111L);
    highscores.put("Andy", 2222L);
    highscores.put("Sven", 2222L);
    highscores.put("Otto", 2222L);
    highscores.put("Tom",  3333L);
    highscores.put("Bodo", 3333L);
    highscores.put("Mike", 3333L);

    final Map<String, Long> sortedHighscores = sortMapByValue(highscores);
    System.out.println(sortedHighscores);
}
```

Listing 7.29 *Ausführbar als* **'SORTBYVALUEEXAMPLE4'**

Starten wir das Programm SORTBYVALUEEXAMPLE4, so werden die drei Einträge mit den Punkteständen 3333 und 2222 erst absteigend nach Wert und zusätzlich nach Name sortiert:

```
{Bodo=3333, Mike=3333, Tom=3333, Andy=2222, Otto=2222, Sven=2222, Tim=1111}
```

8 Datumsverarbeitung seit JDK 8

In diesem Kapitel stelle ich das in JDK 8 integrierte Date and Time API vor. Einführend gebe ich in Abschnitt 8.1 einen Überblick über wichtige Aufzählungen, Klassen und Interfaces. Nachdem die Grundlagen für das Verständnis und den praktischen Einsatz gelegt wurden, beschäftigen wir uns in Abschnitt 8.2 mit dem Thema Datumsarithmetik. Schließlich stelle ich in Abschnitt 8.3 ein paar Informationen zur Migration bestehender Datumsverarbeitungsfunktionalitäten bereit.

Die im Rahmen von JDK 8 entwickelten Klassen adressieren die Probleme mit den bisherigen Datums-APIs des JDKs und nutzen vor allem Ideen aus der Bibliothek Joda-Time, deren Schöpfer Stephen Colebourne eine führende Rolle bei der Entwicklung des neuen Datums-APIs innehatte. Die Zielsetzung war, alles robuster und einfacher nutzbar zu machen und ein gelungenes, hilfreiches API zur Verwaltung und zur Manipulation von Datums- und Zeitwerten bereitzustellen. Diese Ergänzungen finden mit Java 8 endlich Einzug ins JDK. Schauen wir uns das Ganze einmal an.

8.1 Überblick über die neu eingeführten Typen

Das mit JDK 8 realisierte Date and Time API fügt dem JDK einige Funktionalität in verschiedenen Packages unter `java.time` hinzu. Dabei unterscheidet man im Wesentlichen zwei Konzepte: Zum einen gibt es die kontinuierliche oder Maschinenzeit, die linear voranschreitet und für die durch die Klasse `java.time.Instant` ein spezieller Zeitpunkt repräsentiert wird. Das ist mit der Klasse `Date` vergleichbar, jedoch wird eine Auflösung im Bereich von Nanosekunden geboten. Zum anderen existieren Datumsklassen, die eher an menschlichen Denkweisen ausgerichtet sind: Die Klassen `LocalDate` und `LocalTime` aus dem Package `java.time` repräsentieren Datumswerte ohne Zeitzonen in Form eines Datums bzw. einer Zeit. Beide modellieren jeweils nur die durch den Klassennamen beschriebene Zeitkomponente, also Datum oder Zeit.

Nach dieser Einführung möchte ich nun einige Aufzählungen, Klassen und Interfaces kurz vorstellen, bevor ich dann in jeweils separaten Abschnitten auf diese Typen anhand von Beispielen ein wenig genauer eingehe.

8.1.1 Neue Aufzählungen, Klassen und Interfaces

Das Date and Time API definiert im Package `java.time.temporal` verschiedene Basisinterfaces, unter anderem folgende in Abbildung 8-1 gezeigte:

- `TemporalAccessor` – Dieses Interface bildet die Basis für viele Zeit- und Datumsrepräsentationen und definiert Lesezugriffe auf den jeweils modellierten Wert. Ein solcher besitzt eine Einheit, die durch das Interface `TemporalUnit` bestimmt wird. Diverse Aufzählungen implementieren das Interface `TemporalAccessor`, etwa `DayOfWeek` und `Month` zur Modellierung von Wochentagen bzw. Monaten.
- `TemporalAdjuster` – Dieses Interface definiert eine Basis für verschiedene Varianten der Anpassung von Datums- und Zeitwerten.

Abbildung 8-1 *Zentrale Interfaces im Überblick*

Ergänzend gibt es das Basisinterface `Temporal` sowie drei weitere Interfaces, die uns in den folgenden Abschnitten immer wieder einmal begegnen werden:

- `Temporal` – Dieses Interface ist eine Erweiterung von `TemporalAccessor` und ist die Basis für verschiedene Klassen aus dem neuen Date and Time API, die Zeitpunkte modellieren, wie `Instant` oder `LocalTime`. Das Interface `Temporal` bietet neben Lesezugriffen auch modifizierende Zugriffe, wobei hiermit gemeint ist, dass neue Instanzen mit veränderter Wertebelegung entstehen.
- `TemporalUnit` – Dieses Interface beschreibt eine Zeiteinheit. Konkrete Werte findet man in der Aufzählung `java.time.temporal.ChronoUnit`, etwa `MILLIS`.
- `TemporalAmount` – Dieses Interface modelliert eine Abstraktion für eine Zeitspanne und nutzt eine `TemporalUnit`.
- `TemporalField` – Dieses Interface bietet Zugriff auf Attribute von Datums- bzw. Zeitwerten, wie etwa die Attribute `DAY_OF_WEEK` oder `HOUR_OF_DAY`.

Diese drei sind zum Verständnis des Zusammenspiels in Abbildung 8-2 dargestellt.

Abbildung 8-2 *Weitere Interface im Date and Time API*

Auch die in Abbildung 8-3 gezeigten und hier aufgelisteten Klassen machen die Arbeit mit dem neuen Date and Time API angenehm:

- `Instant` – Ein `Instant` repräsentiert ein eher technisches Konstrukt.
- `LocalDate`, `LocalTime` und `LocalDateTime` – Diese Klassen dienen der Verarbeitung von Datums- bzw. Zeitinformationen, aber auch zu deren Kombination.
- `ZoneId`, `ZoneOffset` und `ZonedDateTime` – Diese Klassen helfen bei der Verarbeitung von Daten, die sich auf Zeitzonen beziehen.

Abbildung 8-3 *Wichtige Klassen aus dem Date and Time API im Überblick*

8.1.2 Die Aufzählungen DayOfWeek und Month

Die Aufzählungen `java.time.DayOfWeek` und `java.time.Month` implementieren die Interfaces `TemporalAccessor` und `TemporalAdjuster`. Der Einsatz dieser Aufzählungen macht einerseits den Sourcecode lesbar und vermeidet andererseits auch einfache Fehler, weil man typsichere Konstanten anstelle von Magic Numbers[1] verwenden kann. Bei Nutzung der alten APIs entstehen schnell Probleme durch Inkonsistenzen bei 0- und 1-basierten Angaben. Betrachten wir exemplarisch den Monat Februar. Im `Calendar`-API wusste man – ohne Blick in das Javadoc des APIs – nie so genau, ob Februar nun dem Wert 1 oder 2 entsprach.

Neben der Typsicherheit bieten die neuen Aufzählungstypen den Vorteil, dass man Berechnungen mit ihnen durchführen kann. Nachfolgend demonstriere ich dies, indem ich zu einem Sonntag 5 Tage hinzu addiere und zum Februar 10 Monate:

```
public static void main(final String[] args)
{
    final DayOfWeek sunday = DayOfWeek.SUNDAY;
    final Month february = Month.FEBRUARY;

    final DayOfWeek friday = sunday.plus(5);
    final Month december = february.plus(10);

    System.out.println(friday);
    System.out.println(december);
}
```

Listing 8.1 *Ausführbar als* '**MONTHANDDAYOFWEEKEXAMPLE**'

Wie erwartet, landet man an einem Freitag bzw. im Dezember, wenn man das Progamm MONTHANDDAYOFWEEKEXAMPLE ausführt:

```
FRIDAY
DECEMBER
```

8.1.3 Die Klassen MonthDay, YearMonth und Year

Für einige Anwendungsfälle benötigt man statt einer vollständigen Angabe aus Jahr, Monat und Tag lediglich eine Teilmenge der Informationen. Man kann sich Kombinationen aus Jahr und Monat, Monat und Tag sowie einfach nur Jahr vorstellen, um gewisse Datumsangaben zu modellieren. Für diese Zwecke wurden im Package `java.time` die Klassen `MonthDay`, `YearMonth` sowie `Year` eingeführt. Während `MonthDay` keine Veränderungen erlaubt, weil es die Interfaces `TemporalAccessor` und `TemporalAdjuster` implementiert, ermöglichen `YearMonth` und `Year` auch Modifikationen. Dazu realisieren diese neben `TemporalAdjuster` zusätzlich das Interface `Temporal`, das bei modifizierendem Zugriff jeweils entsprechend veränderte Instanzen zurückliefert. Im folgenden Listing nutzen wir die drei zuvor genannten Klassen:

[1] Darunter versteht man Zahlenliterale, die im Sourcecode auftreten.

```
public static void main(final String[] args)
{
    // MonthDay: Achtung, ISO-Format mit der Reihenfolge: Monat, Tag
    final MonthDay february7th = MonthDay.of(Month.FEBRUARY, 7);

    // YearMonth: Zur Lesbarkeit besser Month statisch importieren
    final YearMonth february2014 = YearMonth.of(2014, Month.FEBRUARY);

    // Year
    final Year year = Year.of(2012);
    final boolean isLeapYear = year.isLeap();

    System.out.println("MonthDay:  " + february7th);
    System.out.println("YearMonth: " + february2014);
    System.out.println("Year:      " + year + " / isLeap? " + isLeapYear);
}
```

Listing 8.2 *Ausführbar als* '**YEARANDMOREEXAMPLE**'

Das Programm YEARANDMOREEXAMPLE produziert folgende Konsolenausgaben:

```
MonthDay:  --02-07
YearMonth: 2014-02
Year:      2012 / isLeap? true
```

Anhand der Ausgaben sieht man verschiedene Notationsformen, wobei die Darstellung von `MonthDay` durch das Doppelminus etwas komisch anmutet. Man erkennt aber auch, dass man von der objektorientierten Umsetzung profitiert: Beispielsweise lässt sich per Aufruf von `isLeap()` prüfen, ob ein Jahr ein Schaltjahr ist.

8.1.4 Die Klasse `Instant`

Die Klasse `java.time.Instant` basiert auf dem Interface `Temporal`. Eine Instanz vom Typ `Instant` repräsentiert einen Zeitpunkt in Nanosekunden bezogen auf den Referenzzeitpunkt 1.1.1970 00:00:00 Uhr UTC. Die Zeit schreitet bei dieser Modellierung linear voran, wodurch sich die Verarbeitung durch Computer vereinfacht, da keine Spezialfälle zu betrachten sind.[2]

Im folgenden Beispiel modellieren wir Abfahrts- und Ankunftszeiten einer Reise mit der Dauer von 5 Stunden, die um 12:30 Uhr beginnt. Diesen Startzeitpunkt ermitteln wir aus textuellen Informationen durch den Aufruf von `parse(String)`. Zuvor zeige ich den Aufruf von `now()`, um den jetzigen Zeitpunkt zu bestimmen. Für die Berechnungen nutzen wir die Methode `plus(long, TemporalUnit)` bzw. die überladene Variante `plus(TemporalAmount)`, um auf einen Zeitpunkt vom Typ `Instant` eine Zeitspanne zu addieren. So erhalten wir die erwartete Ankunftszeit als `expectedArrivalTime`. Zudem nehmen wir eine Verspätung von 7 Minuten an, was der realen Ankunftszeit im `Instant`-Objekt `arrival` entspricht:

[2] Allerdings werden Schaltsekunden auf die letzten 1000 Sekunden des Tages verteilt, wodurch es möglicherweise zu Abweichungen von Zeitsystemen außerhalb von Java kommen kann.

```
public static void main(final String[] args)
{
    // Instant mit now() erzeugen
    final Instant now = Instant.now();

    // Abfahrt 12:30 und Reisedauer 5 Stunden sowie 7 Minuten Verspätung
    final Instant departureTime = Instant.parse("2015-02-07T12:30:00Z");
    final Instant expectedArrivalTime = departureTime.plus(5, ChronoUnit.HOURS);
    final Instant arrival = expectedArrivalTime.plus(Duration.ofMinutes(7));

    System.out.println("now:       " + now);
    System.out.println("departure: " + departureTime);
    System.out.println("expected:  " + expectedArrivalTime);
    System.out.println("arrival:   " + arrival);
}
```

Listing 8.3 *Ausführbar als* **'INSTANTEXAMPLE'**

Das Programm INSTANTEXAMPLE gibt in etwa Folgendes aus:

```
now:       2015-05-17T17:28:10.654Z
departure: 2015-02-07T12:30:00Z
expected:  2015-02-07T17:30:00Z
arrival:   2015-02-07T17:37:00Z
```

8.1.5 Die Klasse `Duration`

Die Klasse `java.time.Duration` erlaubt es, eine Zeitdauer in Nanosekunden festzulegen. Instanzen der Klasse `Duration` können durch Aufruf verschiedener Methoden konstruiert werden, z. B. aus Werten unterschiedlicher Zeiteinheiten[3] wie folgt:

```
public static void main(final String[] args)
{
    // Erzeugung mit ofXYZ()-Methoden
    final Duration durationFromNanos = Duration.ofNanos(3);
    final Duration durationFromMillis = Duration.ofMillis(7);
    final Duration durationFromSeconds = Duration.ofSeconds(15);
    final Duration durationFromMinutes = Duration.ofMinutes(30);
    final Duration durationFromHours = Duration.ofHours(45);
    final Duration durationFromDays = Duration.ofDays(60);

    System.out.println("From Nanos:   " + durationFromNanos);
    System.out.println("From Millis:  " + durationFromMillis);;
    System.out.println("From Seconds: " + durationFromSeconds);
    System.out.println("From Minutes: " + durationFromMinutes);
    System.out.println("From Hours:   " + durationFromHours);
    System.out.println("From Days:    " + durationFromDays);
}
```

Listing 8.4 *Ausführbar als* **'DURATIONEXAMPLE'**

[3] Zeiteinheiten mit variabler Länge, wie Monate, werden nicht unterstützt.

Führen wir das Programm DURATIONEXAMPLE aus, so kommt es zu den gezeigten Ausgaben, wobei im Speziellen folgende Dinge von Interesse sind: Zeitdifferenzen werden offenbar minimal in der Zeiteinheit Sekunden und maximal in der Zeiteinheit von Stunden abgebildet, wodurch für 60 Tage der Wert 1440 Stunden zustande kommt:

```
From Nanos:   PT0.000000003S
From Millis:  PT0.007S
From Secs:    PT15S
From Minutes: PT30M
From Hours:   PT45H
From Days:    PT1440H
```

Beim Betrachten dieser Ausgaben könnten wir durch die Stringrepräsentation von `Duration` irritiert sein. Diese mag zunächst etwas ungewöhnlich erscheinen. Sie folgt aber der Norm ISO 8601 und die Ausgabe startet immer mit dem Kürzel `PT`.[4] Danach gibt es Sektionen für Stunden (`H`), Minuten (`M`) und Sekunden (`S`). Sofern nötig, werden Millisekunden bzw. gar Nanosekunden als Dezimalzahl dargestellt.

Betrachten wir einfache Berechnungen mit der Methode `between()`, die eine `Duration` aus der Differenz zweier `Instant`-Objekte folgendermaßen errechnen kann:

```
public static void main(final String[] args)
{
    final Instant now = Instant.now();
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant myBirthday2015 = Instant.parse("2015-02-07T00:00:00Z");

    // Erzeugung mit between()
    final Duration duration1 = Duration.between(now, silvester2013);
    final Duration duration2 = Duration.between(silvester2013,
                                                myBirthday2015);

    System.out.println(now + " -- " + silvester2013 + ": " + duration1);
    System.out.println(silvester2013 + " -- " + myBirthday2015 + ": " +
                       duration2);
}
```

Listing 8.5 *Ausführbar als* **'DURATIONCREATIONEXAMPLE'**

Führen wir das Programm DURATIONCREATIONEXAMPLE aus, so kommt es in etwa zu folgenden Ausgaben:

```
2017-05-23T10:03:42.816Z -- 2013-12-31T00:00:00Z: PT-29746H-3M-42.816S
2013-12-31T00:00:00Z -- 2015-02-07T00:00:00Z: PT9672H
```

Anhand der Ausgaben sehen wir, dass nicht immer alle Einzelangaben vorhanden sein müssen und auch negative Zeitdauern möglich sind. Das entsteht im Beispiel dadurch, dass die Differenz eines vergangenen Zeitpunkts mit einem danach liegenden berechnet wird.

[4] Laut `http://en.wikipedia.org/wiki/ISO_8601#Durations` ergibt sich dies aus der historischen Benennung Period, also `P`, und das `T` steht für Time.

Wissenswertes zu Berechnungen

Zusätzlich zu der zuvor gezeigten Differenzberechnung mit `between(Temporal, Temporal)` ist auch eine Addition einer durch eine `Duration` definierten Zeitspanne zu einem `Instant`-Objekt möglich. Man erhält als Ergebnis wieder ein `Instant`-Objekt. Als Beispiel soll unter anderem ausgehend vom Heiligabend, dem 24.12.2013, eine Woche in die Zukunft zum 31.12.2013, also Silvester, gesprungen werden. Auch für das Releasedatum von JDK 8, den 18.3.2014 ermitteln wir die Zeitdifferenz zum Silverster-Tag. Schließlich zeige ich zwei Additionen einer Zeitspanne von einer Woche. Wir schreiben dazu folgendes Programm:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Instant christmas2013 = Instant.parse("2013-12-24T00:00:00Z");
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant jdk8Release = Instant.parse("2014-03-18T00:00:00Z");

    // Vergleichswerte errechnen
    System.out.println("Christmas -> Silvester:     " +
                       Duration.between(christmas2013, silvester2013));
    System.out.println("Silvester -> JDK 8 Release: " +
                       Duration.between(silvester2013, jdk8Release));

    // Berechnungen
    final Instant calcSilvester_1 = christmas2013.plus(Duration.ofDays(7));
    final Instant calcSilvester_2 = christmas2013.plus(7, ChronoUnit.DAYS);

    System.out.println(calcSilvester_1);
    System.out.println(calcSilvester_2);
}
```

Listing 8.6 *Ausführbar als* **'DURATIONCALCULATIONEXAMPLE'**

Das Programm DURATIONCALCULATIONEXAMPLE erzeugt folgende Ausgaben:

```
Christmas -> Silvester:     PT168H
Silvester -> JDK 8 Release: PT1848H
2013-12-31T00:00:00Z
2013-12-31T00:00:00Z
```

Wenn wir einige Wochen oder Monate in die Vergangenheit oder Zukunft springen wollen, dann wären dazu Methoden wie `ofWeeks(long)` bzw. `ofMonths(long)` praktisch. Diese existieren jedoch für `Instant`s nicht. Während die fehlende Bereitstellung einer Methode von `ofWeeks(long)` sich noch recht gut durch eigene Berechnungen unter Zuhilfenahme von `ofDays(long)` realisieren lässt, wird dies für Monate ohne die Existenz der Methode `ofMonths(long)` schwieriger. Ein Nachbau per `ofDays(long)` wirkt unter anderem wegen unterschiedlicher Monatslängen umständlich und man entdeckt möglicherweise die Methode `plus(long, TemporalUnit)`. Diese scheint für unsere Berechnungen prädestiniert zu sein, um Wochen oder Monate in die Zukunft zu springen. Setzen wir diese Methode einfach einmal ein:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Instant christmas2013 = Instant.parse("2013-12-24T00:00:00Z");
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");

    // Achtung: Duration bietet nicht ofWeeks(long) oder ofMonths(long)
    final Instant silvester_OneWeek = christmas2013.plus(1, ChronoUnit.WEEKS);
    System.out.println(silvester_OneWeek);
}
```

Listing 8.7 *Ausführbar als* **'DURATIONSPECIALEXAMPLE'**

Wenn Sie das Programm DURATIONSPECIALEXAMPLE ausführen, wird jedoch statt der gewünschten Berechnungen eine Exception folgender Form ausgelöst:

```
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
      Unsupported unit: Weeks
    at java.time.Instant.plus(Instant.java:861)
```

Die Ursache liegt darin, dass zur Definition einer `Duration` keine Zeiteinheiten genutzt werden können, die sich nicht präzise durch Stunden, Minuten usw. ausdrücken lassen. Allerdings könnte man sich Folgendes fragen: Wir haben doch eine `Duration` für die gewünschten Zeiträume basierend auf `Instant`s berechnen können. Wieso war das möglich? Die Antwort ist ganz einfach: Weil wir hier fixe Werte vorliegen haben und somit die Differenz dazwischen eindeutig zu bestimmen war. Andersherum gilt das jedoch nicht: Die abstrakte Angabe von einer Woche oder einem Monat besitzt nämlich kein exaktes Äquivalent in Form einer fixen Zeitspanne in (Milli-)Sekunden, weil ein Tag in der Regel 24 Stunden lang ist – manchmal jedoch auch 23 bzw. 25 Stunden. Selten, aber ab und an gibt es auch Schaltsekunden, die einen Tag minimal verlängern. Hier steht die (vereinfachte) Modellierung in Maschinenzeit in Konflikt mit der komplexeren Wirklichkeit, wo Tage, Wochen und Monate unterschiedlich lang sein können. Später werden wir als Abhilfe die Klasse `Period` kennenlernen.
Neben den `ofXyz()`-Methoden gibt es `with()`-Methoden, die eine neue Instanz einer `Duration` liefern, mit passend veränderten Zeitangaben. Dagegen sind zwar auch mehrere kaskadierende Aufrufe von `ofXyz()` möglich. Diese führen allerdings dazu, dass der letzte »gewinnt«, also nur dieser ausschlaggebend ist – im Abschnitt zu `Period` erfahren wir mehr dazu.

8.1.6 Die Aufzählung `ChronoUnit`

Teilweise haben wir in den bisher gezeigten Beispielen vorgreifend die Aufzählung `java.time.temporal.ChronoUnit` genutzt, um Einheiten von Zeitdauern zu spezifizieren. In der Aufzählung `ChronoUnit` sind alle Zeiteinheiten definiert, mit denen im Date and Time API gerechnet werden kann, unter anderem Minuten, Stunden, Wochen usw. Schauen wir uns folgenden gekürzten Auszug aus dem JDK an:

```
public enum ChronoUnit implements TemporalUnit
{
    NANOS("Nanos", Duration.ofNanos(1)),
    MICROS("Micros", Duration.ofNanos(1000)),
    MILLIS("Millis", Duration.ofNanos(1000_000)),
    SECONDS("Seconds", Duration.ofSeconds(1)),
    MINUTES("Minutes", Duration.ofSeconds(60)),
    ...
```

Tatsächlich ist die Liste der Konstanten umfangreich und reicht von Nanosekunden bis hin zu Jahrtausenden sowie Äras – es gibt sogar eine `FOREVER`-Konstante.

Greifen wir die Idee aus dem für `Instant`s vorgestellten Beispiel der Berechnung von Ankunftszeiten auf: Wir nutzen hier Instanzen von `ChronoUnit`, um die Zeitdauer in verschiedenen Varianten (Stunden und Minuten) darzustellen. Ebenso wie für `Duration` gibt es auch für `ChronoUnit` eine Methode `between(Temporal, Temporal)`, mit der sich die Differenz zwischen zwei Zeitpunkten bestimmen lässt:

```
public static void main(final String[] args)
{
    // Abfahrt jetzt und Reisedauer 5 Stunden
    final Instant departureTime = Instant.now();
    final Instant arrivalTime = departureTime.plus(5, ChronoUnit.HOURS);

    System.out.println("departure now:    " + departureTime);
    System.out.println("arrival now + 5h: " + arrivalTime);

    // Berechnungen durchführen: Differenz bilden
    final long inBetweenHours = ChronoUnit.HOURS.between(departureTime,
                                                         arrivalTime);
    final long inBetweenMinutes = ChronoUnit.MINUTES.between(departureTime,
                                                             arrivalTime);

    System.out.println("inBetweenHours:   " + inBetweenHours);
    System.out.println("inBetweenMinutes: " + inBetweenMinutes);
}
```

Listing 8.8 *Ausführbar als* '**ChronoUnitExample**'

Führt man das Programm ChronoUnitExample aus, so erhält man in etwa folgende Ausgaben auf der Konsole, die sehr schön die Berechnungen von 5 Stunden in die Zukunft sowie die Differenzbildung zwischen zwei Zeitpunkten in verschiedenen Zeiteinheiten (Stunden und Minuten) zeigen:

```
departure now:    2014-02-19T22:13:50.691Z
arrival now + 5h: 2014-02-20T03:13:50.691Z
inBetweenHours:   5
inBetweenMinutes: 300
```

8.1.7 Die Klassen `LocalDate`, `LocalTime` und `LocalDateTime`

Wie eingangs erwähnt, hat die Darstellung von Zeitangaben in Millisekunden, die zwar hilfreich für die Verarbeitung mit Computern ist, nur sehr wenig mit der menschlichen Denkweise und der Orientierung im Zeitsystem zu tun. Menschen denken bevorzugt in Zeitabschnitten oder wiederkehrenden Datumsangaben, etwa 24.12. für Heiligabend, 31.12. für Silvester usw., also Datumsangaben ohne Uhrzeit und Jahr. Manchmal benötigt man »unvollständige Zeitangaben«, wie Uhrzeiten ohne Bezug zu einem Datum, etwa 18:00 Uhr Feierabend, oder als Kombination: dienstags und donnerstags 19 Uhr Karate-Training.[5] Ähnliches mit dem vor Java 8 existierenden API auszudrücken, war recht schwierig. Schauen wir uns nun die Möglichkeiten seit JDK 8 an.

Die Klasse `LocalDate` repräsentiert eine Datumangabe, bestehend rein aus Jahr, Monat und Tag ohne Zeitinformationen. Mit der Klasse `LocalTime` wird eine Zeitangabe ohne Datumsangabe modelliert, z. B. 18:00 Uhr. Die Klasse `LocalDateTime` ist eine Kombination aus beiden. Folgendes Programm zeigt die Klassen im Einsatz und wie einfach und sprechend sich damit Datumsarithmetik realisieren lässt. Abschließend sehen wir noch die Abfrage eines Wochentags sowie die des Tags im Monat und im Jahr, die jeweils sprechende Methodennamen besitzen:

```
public static void main(final String[] args)
{
    final LocalDate michasBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate barbarasBirthday = michasBirthday.plusYears(2).
                                                      plusMonths(1).
                                                      plusDays(17);
    final LocalDate lastDayInFebruary = michasBirthday.with(TemporalAdjusters.
                                                            lastDayOfMonth());

    System.out.println("michasBirthday:    " + michasBirthday);
    System.out.println("barbarasBirthday:  " + barbarasBirthday);
    System.out.println("lastDayInFebruary: " + lastDayInFebruary);

    final LocalTime atTen = LocalTime.of(10,00,00);
    final LocalTime tenFifteen = atTen.plusMinutes(15);
    final LocalTime breakfastTime = tenFifteen.minusHours(2);

    System.out.println("\natTen:             " + atTen);
    System.out.println("tenFifteen:        " + tenFifteen);
    System.out.println("breakfastTime:     " + breakfastTime);

    System.out.println("\nDay Of Week:  " + michasBirthday.getDayOfWeek());
    System.out.println("Day Of Month: " + michasBirthday.getDayOfMonth());
    System.out.println("Day Of Year:  " + michasBirthday.getDayOfYear());
}
```

Listing 8.9 *Ausführbar als* '**LocalDateAndTimeExample**'

Im Listing sehen wir verschiedene Berechnungen mithilfe von `plusXyz()`- sowie `minusXyz()`-Methoden. Darüber hinaus haben wir die Utility-Klasse `Temporal-`

[5] Insbesondere interessiert uns dabei in der Regel nicht die Zeitzone, in der die Termine stattfinden – mit Ausnahme von Telefonterminen etwa mit Geschäftspartnern in Übersee.

`Adjusters` genutzt, in der unterschiedlichste Hilfsmethoden definiert sind. Dazu gehört auch die Methode `lastDayOfMonth()` zur Ermittlung des letzten Tags im Monat, hier um den letzten Tag des Monats Februar im Jahr 1971 zu bestimmen.

Führt man das Programm LOCALDATEANDTIMEEXAMPLE aus, so erhält man folgende Ausgaben:

```
michasBirthday:    1971-02-07
barbarasBirthday:  1973-03-24
lastDayInFebruary: 1971-02-28

atTen:             10:00
tenFifteen:        10:15
breakfastTime:     08:15

Day Of Week:  SUNDAY
Day Of Month: 7
Day Of Year:  38
```

Erweiterungen in der Klasse `LocalDate` mit Java 9

Die Klasse `LocalDate` besitzt seit JDK 9 eine überladene Methode `datesUntil()`. Diese erzeugt einen `Stream<LocalDate>` zwischen zwei `LocalDate`-Instanzen und erlaubt es, optional eine Schrittweite vorzugeben.

Im nachfolgenden Beispiel verwenden wir den Geburtstag des Autors und den Heiligabend desselben Jahres zur Demonstration von Berechnungen mit der Methode `datesUntil()`:

```
public static void main(final String[] args)
{
   final LocalDate myBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
   final LocalDate christmas = LocalDate.of(1971, Month.DECEMBER, 24);

   System.out.println("Day-Stream");
   final Stream<LocalDate> daysUntil = myBirthday.datesUntil(christmas);
   daysUntil.skip(150).limit(4).forEach(System.out::println);

   System.out.println("\n3-Month-Stream");
   final Stream<LocalDate> monthsUntil =
                           myBirthday.datesUntil(christmas, Period.ofMonths(3));
   monthsUntil.limit(3).forEach(System.out::println);
}
```

Listing 8.10 *Ausführbar als* '**DATESUNTILEXAMPLE**'

Führen wir das Programm DATESUNTILEXAMPLE aus, so wird ausgehend vom 7. Februar zunächst 150 Tage in die Zukunft gesprungen, wodurch man am 7. Juli landet. Dann werden vier Werte eines Streams von Tagen ausgegeben:

```
Day-Stream
1971-07-07
1971-07-08
1971-07-09
1971-07-10
```

Zudem zeigen die zweiten Ausgaben die Vorgabe einer Schrittweite, hier drei Monate. Ausgehend vom 7. Februar werden zusätzlich zu diesem Datum noch zwei Daten in die Zukunft aufgelistet:

```
3-Month-Stream
1971-02-07
1971-05-07
1971-08-07
```

Weitere Detailverbesserungen mit Java 9

Neben dieser Neuerung findet man in der Klasse `LocalDate` folgende zwei erwähnenswerte Methoden:

- `toEpochSecond(LocalTime, ZoneOffset)` – Konvertiert eine `LocalDate`-Instanz in einen `long`, der den seit dem Referenzwert 1970-01-01T00:00:00Z vergangenen Sekunden entspricht. Weil ein `LocalDate` keine Zeitinformation besitzt, muss hier ein `LocalTime`- sowie ein `java.time.ZoneOffset`-Objekt übergeben werden.
- `ofInstant(Instant, ZoneId)` – Wandelt ein `java.time.Instant`-Objekt in ein `LocalDate` um, wobei eine Zeitzone in Form einer `java.time.ZoneId` benötigt wird.

Auch für diese beiden Methoden schreiben wir ein kleines Beispiel:

```
public static void main(final String[] args)
{
    final LocalDate someday = LocalDate.of(1971, 2, 7);
    final LocalTime time = LocalTime.of(2, 15);
    final Instant instant = Instant.ofEpochMilli(0);

    System.out.println("toEpochSecond: " + someday.toEpochSecond(time,
                                                ZoneOffset.ofHours(-10)));

    System.out.println("ofInstant: " + LocalDate.ofInstant(instant,
                                              ZoneId.of("Europe/Zurich")));
}
```

Listing 8.11 *Ausführbar als* **'LOCALDATECONVERSIONEXAMPLE'**

Das Programm LOCALDATECONVERSIONEXAMPLE gibt Folgendes aus:

```
toEpochSecond: 34758900
ofInstant: 1970-01-01
```

8.1.8 Die Klasse `Period`

Ähnlich wie die Klasse `Duration` implementiert auch die Klasse `java.time.Period` das Interface `TemporalAmount` und modelliert ebenfalls einen Zeitabschnitt, allerdings mit größerer Dauer. Beispiele sind etwa »2 Monate« oder »3 Tage«. Diese Art der Darstellung ist oftmals einfacher zu handhaben als eine korrespondierende Repräsentation in Nano- oder Millisekunden. Konstruieren wir ein paar Instanzen von `Period`:

```
public static void main(final String[] args)
{
    // Erzeuge ein Period-Objekt mit 1 Jahr, 6 Monaten und 3 Tagen
    final Period oneYear_sixMonths_ThreeDays = Period.ofYears(1).withMonths(6).
                                                                  withDays(3);

    // Chaining von of() arbeitet anders, als man es eventuell erwartet!
    // Ergibt ein Period-Objekt mit 3 Tagen statt 2 Monate, 1 Woche und 3 Tagen
    final Period twoMonths_OneWeek_ThreeDays = Period.ofMonths(2).ofWeeks(1).
                                                                   ofDays(3);

    final Period twoMonths_TenDays = Period.ofMonths(2).withDays(10);
    final Period sevenWeeks = Period.ofWeeks(7);
    final Period threeDays = Period.ofDays(3);

    System.out.println("1 year 6 months ...:  " + oneYear_sixMonths_ThreeDays);
    System.out.println("Surprise just 3 days: " + twoMonths_OneWeek_ThreeDays);
    System.out.println("2 months 10 days:     " + twoMonths_TenDays);
    System.out.println("sevenWeeks:           " + sevenWeeks);
    System.out.println("threeDays:            " + threeDays);
}
```

Listing 8.12 *Ausführbar als* **'PERIODEXAMPLE'**

Startet man das Programm PERIODEXAMPLE, so wird Folgendes ausgegeben:

```
1 year 6 months ...:  P1Y6M3D
Surprise just 3 days: P3D
2 months 10 days:     P2M10D
sevenWeeks:           P49D
threeDays:            P3D
```

Anhand des Beispiels und dessen Ausgaben lernen wir einiges über die Klasse `Period`: Zunächst fällt uns wieder die etwas kryptische Stringrepräsentation auf, die der ISO 8601 folgt. Dabei ist `P` das Startkürzel (für Period) und dann stehen `Y` für Jahre, `M` für Monate und `D` für Tage. Als Besonderheit gibt es zum einen noch die Umrechnung für Wochen: `P14D` steht für 2 Wochen und könnte durch `Period.ofWeeks(2)` erzeugt werden. Zum anderen sind auch negative Offsets erlaubt, etwa `P-2M4D`.

Neben diesen Details der Ausgabe sehen wir, dass sich Aufrufe von `ofXyz()` hintereinander ausführen lassen – es gewinnt aber der zuletzt aufgerufene, was logisch ist, wenn man weiß, dass `ofXyz()` statische Methoden sind. Deshalb kann man auf diese Weise keine Zeiträume kombinieren, sondern legt einen initialen Zeitraum fest. Sollen weitere Zeitabschnitte hinzugefügt werden, so muss man dafür verschiedene `withXyz()`-Methoden nutzen. Dabei wird ein Implementierungsdetail sichtbar: Die Klasse `Period` verwaltet drei Einzelwerte, nämlich für Jahre, Monate und Tage, aber

eben nicht für Wochen. Somit gibt es auch keine Methode `withWeeks()`, sondern nur eine `ofWeeks()`, die intern eine Umrechnung in Tage vornimmt.

Wir schauen uns nun an, wie einfach und lesbar Berechnungen gestaltet werden können: Ausgehend vom 7.2.1971 10:11 springen wir 31 Tage und zum Vergleich vier Wochen sowie einen Monat in die Zukunft:

```
public static void main(final String[] args)
{
    final LocalDateTime start = LocalDateTime.of(1971, 2, 7, 10, 11);

    final Period thirtyOneDays = Period.ofDays(31);
    final Period fourWeeks = Period.ofWeeks(4);
    final Period oneMonth = Period.ofMonths(1);

    System.out.println("7.2.1971 + 31 Tage:  " + start.plus(thirtyOneDays));
    System.out.println("7.2.1971 + 4 Wochen: " + start.plus(fourWeeks));
    System.out.println("7.2.1971 + 1 Monat:  " + start.plus(oneMonth));
}
```

Listing 8.13 *Ausführbar als* '**PERIODCALCULATIONEXAMPLE**'

Das Programm PERIODCALCULATIONEXAMPLE erzeugt folgende Ausgaben:

```
7.2.1971 + 31 Tage:  1971-03-10T10:11
7.2.1971 + 4 Wochen: 1971-03-07T10:11
7.2.1971 + 1 Monat:  1971-03-07T10:11
```

8.1.9 Die Klasse `ZonedDateTime`

Neben der Klasse `LocalDateTime` zur Repräsentation von Datum und Uhrzeit ohne Zeitzonenbezug existiert eine Klasse `java.time.ZonedDateTime`. Dieser ist eine Zeitzone zugeordnet und bei Berechnungen werden nicht nur die Zeitzone, sondern auch die Auswirkungen von Sommer- und Winterzeit berücksichtigt. Um die aktuelle Zeit als `ZonedDateTime` zu ermitteln, kann man die Methode `now()` nutzen. Es existieren weitere Methoden, etwa um die Zeitzone und andere Werte abzufragen bzw. Instanzen von `ZonedDateTime` mit geänderter Wertebelegung durch Aufruf von `withXyz()`-Methoden zu erzeugen. Interessant und etwas schade ist, dass man beim Aufruf von `withMonth(int)` einen `int`-Wert und keine Monatskonstante übergeben muss. Zur besseren Lesbarkeit empfiehlt sich, die Konstanten trotzdem zu verwenden und durch Aufruf der Methode `getValue()` auf deren `int`-Wert zuzugreifen. Das habe ich im Listing fett markiert.

Im Listing sind verschiedene Beispiele für Berechnungen mit der Klasse `ZonedDateTime` gezeigt, im Speziellen auch ein Wechsel von Jahr, Monat und Tag in zwei Varianten sowie mit einer anderen Zeitzone:

```
public static void main(final String[] args)
{
   // Aktuelle Zeit als ZonedDateTime-Objekt ermitteln
   final ZonedDateTime someDay = ZonedDateTime.of(LocalDate.parse("2020-02-07"),
                                                  LocalTime.parse("17:30:15"),
                                                  ZoneId.of("Europe/Zurich"));

   // Die Uhrzeit ändern und in neuem Objekt speichern
   final ZonedDateTime someDayChangedTime = someDay.withHour(11).withMinute(44);

   // Neues Objekt mit komplett verändertem Datum erzeugen
   final ZonedDateTime dateAndTime = someDayChangedTime.withYear(2008).
                                                        withMonth(9).
                                                        withDayOfMonth(29);

   // Einsatz einer Monatskonstanten und wechseln der Zeitzone
   final ZonedDateTime dateAndTime2 = someDayChangedTime.withYear(2008).
                                      withMonth(Month.SEPTEMBER.getValue()).
                                      withDayOfMonth(29).
                                      withZoneSameInstant(ZoneId.of("GMT"));

   System.out.println("someDay:      " + someDay);
   System.out.println("-> 11:44:     " + someDayChangedTime);
   System.out.println("-> 29.9.2008: " + dateAndTime);
   System.out.println("-> 29.9.2008: " + dateAndTime2);
}
```

Listing 8.14 *Ausführbar als* '**ZONEDDATETIMEEXAMPLE**'

Führt man das Programm ZONEDDATETIMEEXAMPLE aus, so startet man zunächst im Jahr 2020, wechselt dann aber ins Jahr 2008. Dabei kommt es zu den nachfolgenden Ausgaben. Diese zeigen insbesondere den Einfluss von Sommer- und Winterzeit, wodurch im September 2008 die Abweichung von +02:00 angegeben wird. Worauf sich dies bezieht, erkennt man dann durch den Wechsel der Zeitzone auf GMT:

```
someDay:      2020-02-07T17:30:15+01:00[Europe/Zurich]
-> 11:44:     2020-02-07T11:44:15+01:00[Europe/Zurich]
-> 29.9.2008: 2008-09-29T11:44:15+02:00[Europe/Zurich]
-> 29.9.2008: 2008-09-29T09:44:15Z[GMT]
```

8.1.10 Zeitzonen und die Klassen `ZoneId` und `ZoneOffset`

Wir haben bereits einiges an Wissen über das Date and Time API erworben. Zeitzonen wurden bislang eher am Rande betrachtet. Wenn Sie allerdings bei der Datumsarithmetik Zeitzonen beachten müssen, dann helfen dabei seit JDK 8 die Klassen `ZoneId`, `ZoneOffset` sowie die gerade vorgestellte Klasse `ZonedDateTime` aus dem Package `java.time`.

Die Klasse `ZoneId`

Nun wollen wir die Verarbeitung von Zeitzonen anhand eines Beispiels kennenlernen. Zunächst ermitteln wir basierend auf einigen textuellen Zeitzonenkennungen durch Aufruf von `ZoneId.of(String)` die zugehörige `ZoneId`-Instanz und konstruieren

daraus jeweils ein `LocalTime`-Objekt. Dann rufen wir `ZoneId.getAvailableZoneIds()` auf und erhalten so alle verfügbaren Zeitzonen. Mithilfe von Streams und den beiden Methoden `filter()` und `limit()` finden wir drei Kandidaten aus Europa. Das Ganze realisieren wir wie folgt:

```
public static void main(final String[] args)
{
    final Stream<String> zoneIdNames = Stream.of("Asia/Chungking",
                                                 "Africa/Nairobi",
                                                 "America/Los_Angeles");

    zoneIdNames.forEach(zoneIdName ->
    {
        final ZoneId zoneId = ZoneId.of(zoneIdName);
        final LocalTime now = LocalTime.now(zoneId);

        System.out.println(zoneIdName + ": " + now);
    });

    final Set<String> allZones = ZoneId.getAvailableZoneIds();
    final Predicate<String> inEurope = name -> name.startsWith("Europe/");
    final List<String> threeFromEurope = allZones.stream().
                                                 filter(inEurope).limit(3).
                                                 collect(Collectors.toList());

    System.out.println("\nSome timezones in europe:");
    threeFromEurope.forEach(System.out::println);
}
```

Listing 8.15 *Ausführbar als* '**ZONEIDEXAMPLE**'

Das Programm ZONEIDEXAMPLE produziert folgende Ausgaben:

```
Asia/Chungking: 20:10:22.219
Africa/Nairobi: 15:10:22.220
America/Los_Angeles: 05:10:22.222

Some timezones in europe:
Europe/London
Europe/Brussels
Europe/Warsaw
```

Im obigen Listing sehen wir die Verarbeitung von Zeitzonen basierend auf der Klasse `ZoneId`. Etwas unschön ist, dass im JDK statt Konstanten lediglich einfache Strings als IDs genutzt werden. Praktischerweise erhält man die Menge der gültigen Zeitzonen-IDs durch Aufruf von `ZoneId.getAvailableZoneIds()`.

Die Klasse `ZoneOffset`

Auf die Klasse `ZoneOffset` zur Modellierung des Offsets zur GMT (Greenwich Mean Time) (die auch – wenn auch formal nicht ganz korrekt – als UTC (Universal Coordinated Time) bezeichnet wird) möchte ich ganz kurz anhand eines kleinen Beispiels eingehen. Startend beim jetzigen Zeitpunkt `LocalDateTime.now()` berechnen wir für unterschiedliche Zeitzonen ein zugehöriges `ZonedDateTime`-Objekt und geben dessen Offset zur GMT aus:

```
public static void main(final String[] args)
{
    final Stream<String> zoneIdNames = Stream.of("Europe/Berlin",
                                                 "America/Los_Angeles",
                                                 "Australia/Adelaide");

    zoneIdNames.forEach(zoneIdName ->
    {
        final ZoneId zoneId = ZoneId.of(zoneIdName);
        final LocalDateTime ldt = LocalDateTime.now();
        final ZonedDateTime zdt = ldt.atZone(zoneId);
        final ZoneOffset offset = zdt.getOffset();

        System.out.format("offset for '%s' is %s\n", zoneId, offset);
    });
}
```

Listing 8.16 *Ausführbar als* '**ZONEOFFSETEXAMPLE**'

Führen wir das Programm ZONEOFFSETEXAMPLE aus, erhalten wir folgende Ausgaben, mit deren Hilfe wir die zu den Zeitzonen korrespondierenden Offsets erfahren:

```
offset for 'Europe/Berlin' is +02:00
offset for 'America/Los_Angeles' is -07:00
offset for 'Australia/Adelaide' is +10:30
```

8.1.11 Die Klasse `Clock`

In einigen technischen Anwendungsfällen benötigt man Zugriff auf Millisekundenangaben. Früher hat man dann Aufrufe von `System.currentTimeMillis()` genutzt, um die aktuelle Zeit in Millisekunden seit dem 1.1.1970 zu ermitteln. Nun kann man dazu die Klasse `java.time.Clock` verwenden und schreibt etwa Folgendes:

```
public static void main(final String[] args)
{
    printClockAndMillis(Clock.systemUTC()); // Basis UTC
    printClockAndMillis(Clock.systemDefaultZone()); // Basis Default-Zeitzone
}

private static void printClockAndMillis(final Clock clock)
{
    System.out.println(clock + " / ms: " + clock.millis());
}
```

Listing 8.17 *Ausführbar als* '**FIRSTCLOCKEXAMPLE**'

Das Programm FIRSTCLOCKEXAMPLE gibt etwa Folgendes aus:

```
SystemClock[Z] / ms: 1430766415687
SystemClock[Europe/Berlin] / ms: 1430766415745
```

Wenn man genau hinschaut, sieht man eine minimale Differenz in den Millisekunden. Das liegt vor allem daran, dass die Methode `millis()` immer bezogen auf die Zeitzone GMT bzw. UTC arbeitet.

Es ist nicht direkt ein wirklicher Mehrwert dieser Klasse sichtbar. Dieser liegt darin, dass die Klasse `Clock` als Taktgeber für viele Zeitklassen genutzt werden kann: Dazu besitzen diverse Klassen neben der Methode `now()` auch eine Methode `now(Clock)`, der man eine alternative `Clock`-Instanz übergeben kann, etwa wie folgt:

```
final LocalDateTime now = LocalDateTime.now(mySpecialClock);
```

Wozu könnte das sinnvoll einsetzbar sein? Beispielsweise lassen sich für Unit Tests fixe Zeitangaben realisieren. Schauen wir zum Verständnis auf folgendes Beispiel, das durch Aufruf von `fixed()` eine sich nicht verändernde `Clock` erzeugt:

```
public static void main(final String[] args) throws InterruptedException
{
   final Clock clock1 = Clock.systemUTC();
   final Clock clock2 = Clock.systemDefaultZone();
   final Clock clock3 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Hong_Kong"));

   printClocks(clock1, clock2, clock3);

   Thread.sleep(10_000);
   System.out.println("\nAfter 10 s\n");

   printClocks(clock1, clock2, clock3);
}

private static void printClocks(final Clock... clocks)
{
   for (final Clock clock : clocks)
   {
      System.out.println("LocalTime: " + LocalTime.now(clock));
   }
}
```

Listing 8.18 *Ausführbar als* **'SECONDCLOCKEXAMPLE'**

Führen wir das Programm SECONDCLOCKEXAMPLE aus, so zeigt sich im Gegensatz zum Aufruf von `millis()`, dass die mithilfe einer `Clock` erzeugten Instanzen die Zeitzonen berücksichtigen. Es wird auch deutlich, dass die Wartezeit von 10 Sekunden für die ersten beiden `Clock`s jeweils eine Änderung von 10 Sekunden bewirkt. Die FixedClock verändert sich dagegen nicht:

```
LocalTime: 20:48:10.936
LocalTime: 22:48:10.937
LocalTime: 04:48:10.936

After 10 s

LocalTime: 20:48:20.948
LocalTime: 22:48:20.948
LocalTime: 04:48:10.936
```

Wie dieses Beispiel zeigt, wird es mit einer speziellen `Clock` möglich, die Datumsarithmetik unter anderen Gegebenheiten, etwa veränderter Zeitzone, zu testen, ohne dass man dazu die Systemeinstellungen verändern müsste. Gerade für Unit Tests kann dies nützlich sein, um definierte Ausgangssituationen zu erhalten.

8.1.12 Formatierung und Parsing

Die Klasse `java.time.format.DateTimeFormatter` ist bei der formatierten Ausgabe und beim Parsing von Datumswerten hilfreich. Neben diversen vordefinierten Formaten kann man vielfältige eigene Varianten bereitstellen. Das wird unter anderem sowohl durch die Methoden `ofPattern()` sowie `ofLocalizedDate()` als auch `parse()` mit Angabe eines Formats möglich. Dabei ist es praktischerweise möglich, ein komplexeres Muster zur Formatierung anzugeben und sogar auch zum Parsen zu verwenden. Das wird im folgenden Listing demonstriert – es gibt eine Vielzahl weiterer Möglichkeiten, die hier nicht alle vorgestellt werden können. Deshalb ist ein Blick in die ausführliche Dokumentation des JDKs lohnenswert.

```
public static void main(final String[] args)
{
   // Definition einiger spezieller Formatter
   final DateTimeFormatter ddMMyyyyFormat = ofPattern("dd.MM.yyyy");
   final DateTimeFormatter italiandMMMMy = ofPattern("d.MMMM y",
                                                     Locale.ITALIAN);
   final DateTimeFormatter shortGerman =
                           DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT).
                           withLocale(Locale.GERMAN);

   // Achtung: Die textuellen Teile sind in Hochkomma einzuschließen
   final String customPattern = "'Der 'dd'. Tag im 'MMMM' im Jahr 'yy'.'";
   final DateTimeFormatter customFormat = ofPattern(customPattern);

   System.out.println("Formatting:\n");
   final LocalDate february7th = LocalDate.of(1971, 2, 7);

   System.out.println("ddMMyyyyFormat: " + ddMMyyyyFormat.format(february7th));
   System.out.println("italiandMMMMy:  " + italiandMMMMy.format(february7th));
   System.out.println("shortGerman:    " + shortGerman.format(february7th));
   System.out.println("customFormat:   " + customFormat.format(february7th));


   // Parsen von Datumswerten
   System.out.println("\nParsing:\n");

   final LocalDate fromIsoDate = LocalDate.parse("1971-02-07");
   final LocalDate fromddMMyyyyFormat = LocalDate.parse("18.03.2014",
                                                        ddMMyyyyFormat);
   final LocalDate fromShortGerman = LocalDate.parse("18.03.14",
                                                     shortGerman);
   final LocalDate fromCustomFormat =
                   LocalDate.parse("Der 31. Tag im Dezember im Jahr 19.",
                                   customFormat);

   System.out.println("From ISO Date:       " + fromIsoDate);
   System.out.println("From ddMMyyyyFormat: " + fromddMMyyyyFormat);
   System.out.println("From short german:   " + fromShortGerman);
   System.out.println("From custom format:   " + fromCustomFormat);
}
```

Listing 8.19 *Ausführbar als* **'FORMATTINGANDPARSINGEXAMPLE'**

Das Programm FORMATTINGANDPARSINGEXAMPLE produziert folgende Ausgaben, die ein erstes Gefühl für die Formatierung und das Parsing vermitteln:

```
Formatting:

ddMMyyyyFormat:  07.02.1971
italian_dMMMMyy: 7.febbraio 1971
shortGerman:     07.02.71
customFormat:    Der 07. Tag im Februar im Jahr 71.

Parsing:

From ISO Date:       1971-02-07
From ddMMyyyyFormat: 2014-03-18
From short german:   2014-03-18
From custom format:   2019-12-31
```

8.2 Datumsarithmetik

Zuvor bin ich zwar auf Erleichterungen bei der Datumsverarbeitung mithilfe der vorgestellten Klassen und Interfaces eingegangen – jedoch habe ich bislang (zumindest komplexere) Berechnungen noch weitestgehend außen vor gelassen. Allerdings findet man gerade in der Praxis bei der Verarbeitung von Datumswerten oftmals auch die Notwendigkeit für Berechnungen, beispielsweise um einige Tage, Wochen oder gar Monate in die Zukunft oder die Vergangenheit zu springen. Praktischerweise sind diverse gebräuchliche Operationen der Datumsarithmetik in der Utility-Klasse `TemporalAdjusters` gebündelt und basieren auf dem Functional Interface `TemporalAdjuster` – beide aus dem Package `java.time.temporal`. Im Beispiel für die Klassen `LocalDate`, `LocalTime` und `LocalDateTime` haben wir bereits einen ersten Eindruck von den Möglichkeiten gewinnen können. Dieses Wissen wollen wir nun ausbauen.

8.2.1 Einstieg in die Datumsarithmetik

Das Interface `TemporalAdjuster`

Ein `TemporalAdjuster` definiert die Methode `adjustInto(Temporal)`. In deren Implementierungen kann man eine flexible Anpassung sowohl für Datums- als auch Zeit-Objekte, also genauer für alle Objekte mit dem Basistyp `Temporal`, vornehmen:

```
@FunctionalInterface
public interface TemporalAdjuster
{
    Temporal adjustInto(Temporal temporal);
}
```

Realisierungen dieses Interface helfen dabei, Berechnungen auszuführen, wie etwa den ersten oder letzten Tag im Monat zu ermitteln. Relativ klar wird der Sinn, wenn man sich die vordefinierten Varianten anschaut.

Vordefinierte `TemporalAdjusters`

Wie eingangs erwähnt, findet man in der Utility-Klasse `TemporalAdjusters` eine Menge gebräuchlicher Operationen der Datumsarithmetik. Das sind unter anderem:

- `firstDayOfMonth()`, `firstDayOfNextMonth()` und `lastDayOfMonth()` – Ermittelt den ersten oder letzten Tag im (nächsten) Monat.
- `firstDayOfYear()`, `firstDayOfNextYear()` und `lastDayOfYear()` – Berechnet den ersten oder letzten Tag im (nächsten) Jahr.
- `firstInMonth(DayOfWeek)` und `lastInMonth(DayOfWeek)` – Springt zum ersten oder letzten als Parameter übergebenen Wochentag im Monat.
- `next(DayOfWeek)`, `nextOrSame(DayOfWeek)`, `previous(DayOfWeek)` und `previousOrSame(DayOfWeek)` – Mit diesen Methoden kann man den nächsten oder vorherigen Wochentag berechnen, etwa den nächsten Freitag. Dabei wird gegebenenfalls auch berücksichtigt, ob man sich bereits an diesem Wochentag befindet. In diesem Fall findet selbstverständlich keine Anpassung statt.

Zum besseren Verständnis der aufgelisteten `TemporalAdjusters` schauen wir, wie einfach und gut lesbar sich damit Datumsarithmetik programmieren lässt – vor allem wenn man dabei statische Imports nutzt. Im folgenden Listing habe ich bewusst einige Dinge nicht statisch importiert, um einen Vergleich zu ermöglichen, der den Gewinn an Lesbarkeit zeigt.[6] Im Beispiel werden ausgehend vom 7. Februar 2015 einige »Zeitsprünge« vorgenommen und das Ergebnis ausgegeben:

```
import static java.time.temporal.TemporalAdjusters.firstDayOfNextMonth;
import static java.time.temporal.TemporalAdjusters.firstInMonth;
import static java.time.temporal.TemporalAdjusters.lastInMonth;
import static java.time.DayOfWeek.SATURDAY;

// ...

public static void main(final String[] args)
{
    final LocalDate date = LocalDate.of(2020, Month.FEBRUARY, 7);

    LocalDate firstOfMonth = date.with(TemporalAdjusters.firstDayOfMonth());
    LocalDate lastofMonth = date.with(TemporalAdjusters.lastDayOfMonth());
    LocalDate firstOfNextMonth = date.with(firstDayOfNextMonth());
    LocalDate firstMondayInMonth = date.with(firstInMonth(DayOfWeek.MONDAY));
    LocalDate lastSaturdayInMonth = date.with(lastInMonth(SATURDAY));

    System.out.println("date:                 " + date);
    System.out.println("lastofMonth:          " + lastofMonth);
    System.out.println("firstOfMonth:         " + firstOfMonth);
    System.out.println("firstOfNextMonth:     " + firstOfNextMonth);
    System.out.println("firstMondayInMonth:   " + firstMondayInMonth);
    System.out.println("lastSaturdayInMonth:  " + lastSaturdayInMonth);
}
```

Listing 8.20 *Ausführbar als* **'PREDEFINEDTEMPORALADJUSTERSEXAMPLE'**

[6] Unüberlegt eingesetzt leidet oftmals die Verständlichkeit, weil Abhängigkeiten schwieriger ersichtlich sind.

Starten wir das Programm PREDEFINEDTEMPORALADJUSTERSEXAMPLE, so erhalten wir folgende Resultate der Datumsberechnungen:

```
date:                2020-02-07
lastofMonth:         2020-02-29
firstOfMonth:        2020-02-01
firstOfNextMonth:    2020-03-01
firstMondayInMonth:  2020-02-03
lastSaturdayInMonth: 2020-02-29
```

In der Definition des Interface `TemporalAdjuster` haben wir die Methode `adjustInto(Temporal)` gesehen. Wenn wir uns aber das obige Beispiel anschauen, sehen wir nirgendwo den Aufruf der `adjustInto(Temporal)`-Methode. Stattdessen wird die Methode `with(TemporalAdjuster)` aufgerufen. Das liegt daran, dass diese eine besser lesbare und zudem semantisch äquivalente Alternative ist:

```
temporal = thisAdjuster.adjustInto(temporal);
temporal = temporal.with(thisAdjuster);
```

Speziellere vordefinierte `TemporalAdjusters`

Die zuvor genannten `TemporalAdjusters` sind für viele Anwendungsfälle schon ausreichend. Für mehr Flexibilität gibt es noch folgende zwei Methoden:

- `ofDateAdjuster(UnaryOperator<LocalDate>)` – Mithilfe dieser Methode lassen sich `TemporalAdjusters` erstellen. Die auszuführenden Berechnungen werden mithilfe eines `UnaryOperator<LocalDate>` beschrieben. Das ermöglicht viel Flexibilität: Mit dem Lambda `date -> date.plusDays(5)` springt man fünf Tage in die Zukunft. Andere Berechnungen sind ähnlich leicht.
- `dayOfWeekInMonth(int, DayOfWeek)` – Errechnet den n-ten Wochentag im Monat. Dabei wird auch über Monatsgrenzen hinweg gesprungen, etwa wenn man den (nicht existierenden) 7. Dienstag eines Monats ermitteln wollte. Zudem sind negative Werte erlaubt, wobei die Werte 0 und -1 eine spezielle Bedeutung tragen: 0 ermittelt den letzten gewünschten Wochentag im Vormonat, -1 den letzten Wochentag in diesem Monat. Die negativen Werte rechnen ausgehend vom letzten Wochentag des Monats die übergebene Anzahl an Wochen zurück. Der -3. Sonntag wäre also 3 Wochen vor dem letzten Sonntag des Monats und man müsste die Werte -4 sowie `SUNDAY` an `dayOfWeekInMonth(int, DayOfWeek)` übergeben.

Im nachfolgenden Listing definieren wir jeweils eine Hilfsmethode, die die Arbeitsweise der beiden Methoden aus der Aufzählung verdeutlicht:

```
public static void main(final String[] args)
{
    ofDateAdjusterCalculations();
    dayOfWeekInMonthCalculations();
}
```

```
private static void ofDateAdjusterCalculations()
{
    System.out.println("Adjusting with ofDateAdjuster()");
    System.out.println("------------------------------");

    final Map<String, TemporalAdjuster> adjusters = new LinkedHashMap<>();

    adjusters.put("ONE_WEEK_LATER", ofDateAdjuster(localDate ->
                                                   localDate.plusDays(7)));
    adjusters.put("FOUR_WEEKS_LATER", ofDateAdjuster(localDate ->
                                                   localDate.plusDays(28)));
    adjusters.put("ONE_MONTH_LATER", ofDateAdjuster(localDate ->
                                                   localDate.plusMonths(1)));

    final LocalDate base = LocalDate.of(2020, FEBRUARY, 27);
    applyAdjusters(base, adjusters);
}

private static void dayOfWeekInMonthCalculations()
{
    System.out.println("\nAdjusting with dayOfWeekInMonth()");
    System.out.println("---------------------------------");

    final Map<String, TemporalAdjuster> adjusters = new LinkedHashMap<>();

    adjusters.put("THIRD_FRIDAY", dayOfWeekInMonth(3, FRIDAY));
    adjusters.put("LAST_FRIDAY", dayOfWeekInMonth(-1, FRIDAY));
    adjusters.put("LAST_FRIDAY_LAST_MONTH", dayOfWeekInMonth(0, FRIDAY));
    adjusters.put("FRIDAY_3_WEEKS_BEFORE_LAST", dayOfWeekInMonth(-4, FRIDAY));

    final LocalDate base = LocalDate.of(2020, FEBRUARY, 7);
    applyAdjusters(base, adjusters);
}

private static void applyAdjusters(final LocalDate base,
                           final Map<String, TemporalAdjuster> adjusters)
{
    System.out.println("Starting on: " + base);
    adjusters.forEach((name, adjuster) ->
       System.out.println("adjusting to " + name + ": " + base.with(adjuster)));
}
```

Listing 8.21 *Ausführbar als* **'PREDEFINEDTEMPORALADJUSTERSEXAMPLE2'**

Das Programm PREDEFINEDTEMPORALADJUSTERSEXAMPLE2 produziert folgende Ausgaben:

```
Adjusting with ofDateAdjuster()
------------------------------
Starting on: 2020-02-27
adjusting to ONE_WEEK_LATER: 2020-03-05
adjusting to FOUR_WEEKS_LATER: 2020-03-26
adjusting to ONE_MONTH_LATER: 2020-03-27

Adjusting with dayOfWeekInMonth()
---------------------------------
Starting on: 2020-02-07
adjusting to THIRD_FRIDAY: 2020-02-21
adjusting to LAST_FRIDAY: 2020-02-28
adjusting to LAST_FRIDAY_LAST_MONTH: 2020-01-31
adjusting to FRIDAY_3_WEEKS_BEFORE_LAST: 2020-02-07
adjusting to FRIDAY_4_WEEKS_BEFORE_LAST: 2020-01-31
```

Beispiel: `TemporalAdjuster` selbst definieren

Zwar sind die Möglichkeiten der vordefinierten `TemporalAdjusters` schon recht umfangreich und dürften für diverse Anwendungsfälle ausreichen. Mitunter möchte man aber auch eigene Varianten erstellen. Das wollen wir uns anhand des Beispiels `FridayAfterMidOfMonth` anschauen. Hierbei soll ausgehend von einem `LocalDate` auf den Freitag nach Mitte des Monats gesprungen werden. Dazu ist die Methode `Temporal adjustInto(Temporal)` geeignet zu implementieren. Die einzige Hürde ist, aus dem übergebenen `Temporal` ein `LocalDate` zu ermitteln, wozu man `LocalDate.from()` nutzt. Danach muss man auf den 15. Tag im Monat springen (bzw. den 14. für Februar), was einfach durch den Aufruf von `withDayOfMonth()` zu erledigen ist. Schließlich wendet man einen der vordefinierten Adjuster mit `nextOrSame(DayOfWeek)` an:

```
public class FridayAfterMidOfMonth implements TemporalAdjuster
{
    @Override
    public Temporal adjustInto(final Temporal temporal)
    {
        final LocalDate startday = LocalDate.from(temporal);
        final int dayOfMonth = startday.getMonth() == Month.FEBRUARY ? 14 : 15;

        return startday.withDayOfMonth(dayOfMonth).
                        with(TemporalAdjusters.nextOrSame(DayOfWeek.FRIDAY));
    }
}
```

Es ist schon erstaunlich, wie einfach man Funktionalitäten umsetzen kann, wenn man die geeigneten APIs nutzt.

Zum Ausprobieren nutzen wir diesmal das in das JDK integrierte Tool `jshell`, dass mit Java 14 und 15 für kleinere Experimente wirklich brauchbar geworden ist. Nach dem Start der JShell benötigen wir folgende Imports:

```
jshell> import java.time.*
jshell> import java.time.temporal.*
```

Nun können wir die obige Klasse dort definieren und beispielsweise wie folgt aufrufen:

```
jshell> LocalDate feb7 = LocalDate.of(2020, 2, 7)
feb7 ==> 2020-02-07

jshell> var adjustedDay = feb7.with(new FridayAfterMidOfMonth())
adjustedDay ==> 2020-02-14

jshell> adjustedDay.getDayOfWeek()
$17 ==> FRIDAY

jshell> LocalDate mar24 = LocalDate.of(2020, 3, 24)
mar24 ==> 2020-03-24

jshell> var adjustedDay2 = mar24.with(new FridayAfterMidOfMonth())
adjustedDay2 ==> 2020-03-20

jshell> adjustedDay2.getDayOfWeek()
$22 ==> FRIDAY
```

8.2.2 Real-World-Example: Gehaltszahltag

In diesem Abschnitt wollen wir schrittweise eine Klasse `NextPaydayAdjuster` entwickeln, die den typischen Gehaltszahltag in der Schweiz errechnet. Dies ist oftmals der 25. des Monats – wie für die Praxis üblich mit einigen Sonderfällen.

Spezifikation

Fällt dieser Tag jedoch auf ein Wochenende, so gilt der Freitag davor als Auszahlungstag. Befinden wir uns nach dem Auszahlungstag, so ist dieser für den darauffolgenden Monat zu ermitteln. Als Kür gilt es noch, eine Sonderregel für die Gehaltszahlung im Dezember umzusetzen: Hierbei soll die Auszahlung dann Mitte des Monats erfolgen. Gegebenenfalls wird die Zahlung auf den darauffolgenden Montag verlegt, falls der Zahltag auf ein Wochenende fällt. Zum Überprüfen der Implementierung können wir folgende Datumswerte nutzen:

Eingabe	Resultat	Regel
2020-06-21	2020-06-25	normale Anpassung
2020-05-27	2020-06-25	normale Anpassung, nächster Monat
2020-07-21	2020-07-24	Freitag, falls 25. am Wochenende
2019-12-06	2019-12-16	Mitte Monat, Montag nach Wochenende
2019-12-23	2020-01-24	nächster Monat und Freitag, falls 25. am Wochenende

Algorithmus

Versuchen wir einmal, die Anforderungen schrittweise zu entwickeln. Wir beginnen mit der Anpassung eines beliebigen Zeitpunkts auf den 25. Tag des Monats.

Ermittlung des 25. eines Monats Um die gewünschte Anpassung des Wochentags zu realisieren, implementieren wir wiederum das Interface `TemporalAdjuster` und im Speziellen die Methode `adjustInto(Temporal)`. Zunächst ermitteln wir durch Aufruf der Methode `from(TemporalAccessor)` aus der Klasse `LocalDate` eine zum übergebenen `Temporal`-Objekt korrespondierende `LocalDate`-Instanz, die wir mit deren Methode `withDayOfMonth(int)` auf den 25. verschieben:

```
class NextPaydayAdjuster implements TemporalAdjuster
{
    @Override
    public Temporal adjustInto(final Temporal temporal)
    {
        LocalDate date = LocalDate.from(temporal);

        date = date.withDayOfMonth(25);

        return temporal.with(date);
    }
}
```

Probieren wir das doch einmal in der JShell aus:

```
jshell> import java.time.*
jshell> import java.time.temporal.*

jshell> var jan31 = LocalDate.of(2020, Month.JANUARY, 31)

jshell> jan31.with(new NextPaydayAdjuster())
$24 ==> 2020-01-25

jshell> var feb7 = LocalDate.of(2020, Month.FEBRUARY, 7);

jshell> feb7.with(new NextPaydayAdjuster())
$26 ==> 2020-02-25
```

Wie hier gezeigt, funktioniert die Anpassung schon ziemlich gut. Es gibt aber noch eine Kleinigkeit zu bedenken: An Tagen nach dem 25. ist der nächste Zahltag nicht im selben Monat, sondern die Tage vom 26. bis zum Monatsende werden zum nächsten Monat gezählt. Somit passen wir die Berechnung wie folgt an:

```
@Override
public Temporal adjustInto(final Temporal temporal)
{
    LocalDate date = LocalDate.from(temporal);

    if (date.getDayOfMonth() > 25)
    {
         date = date.plusMonths(1);
    }
    date = date.withDayOfMonth(25);

    return temporal.with(date);
}
```

Das ist schon ein guter Schritt. Es fehlt noch die Betrachtung von Wochenenden.

Spezialkorrektur für Wochenenden Fällt ein Zahltag auf ein Wochenende, so ist der Freitag davor auszuwählen. Daraus ergibt sich die fett markierte Ergänzung:

```
@Override
public Temporal adjustInto(final Temporal temporal)
{
    LocalDate date = LocalDate.from(temporal);

    if (date.getDayOfMonth() > 25)
    {
        date = date.plusMonths(1);
    }
    date = date.withDayOfMonth(25);

    if (date.getDayOfWeek() == SATURDAY ||
        date.getDayOfWeek() == SUNDAY)
    {
        date = date.with(TemporalAdjusters.previous(FRIDAY));
    }

    return temporal.with(date);
}
```

Man sieht, wie die ursprünglich sehr einfache Methode durch die Integration der realitätsnahen Anforderungen schon ein wenig kompliziert wird – trotzdem bleibt sie aber dank dem neuen Date and Time API recht lesbar!

Spezialbehandlung für Dezember Wir integrieren nun die Spezialbehandlung für den Dezember. Dort erfolgt die Gehaltszahlung Mitte des Monat oder, falls der 15. auf ein Wochenende fällt, am darauffolgenden Montag.

Wenn wir etwas über die Problematik und die Lösung nachdenken, kommen uns möglicherweise die vordefinierten `TemporalAdjuster`-Klassen in den Sinn: Man kann hier die Methoden `nextOrSame(DayOfWeek)` sowie `previousOrSame(DayOfWeek)` und die passenden `TemporalAdjuster`-Instanzen wie folgt nutzen:

```
if (isDecember)
{
    date = date.with(TemporalAdjusters.nextOrSame(MONDAY));
}
else
{
    date = date.with(TemporalAdjusters.previousOrSame(FRIDAY));
}
```

Vollständige Realisierung Nach diesen einzelnen Schritten schauen wir zum besseren Verständnis noch mal auf die sich daraus ergebende vollständige Realisierung:

```
public class NextPaydayAdjuster implements TemporalAdjuster
{
  @Override
  public Temporal adjustInto(final Temporal temporal)
  {
     LocalDate date = LocalDate.from(temporal);

     boolean isDecember = date.getMonth() == Month.DECEMBER;
     int paymentDay = isDecember ? 15 : 25;

     if (date.getDayOfMonth() > paymentDay)
     {
       date = date.plusMonths(1);

       // Abfragen nochmals nötig, da eventuell um einen Monat verschoben
       isDecember = date.getMonth().equals(Month.DECEMBER);
       paymentDay = isDecember ? 15 : 25;
     }

     date = date.withDayOfMonth(paymentDay);

     if (date.getDayOfWeek() == DayOfWeek.SATURDAY ||
         date.getDayOfWeek() == DayOfWeek.SUNDAY)
     {
         if (isDecember)
           date = date.with(TemporalAdjusters.nextOrSame(DayOfWeek.MONDAY));
         else
           date = date.with(TemporalAdjusters.previousOrSame(DayOfWeek.FRIDAY));
      }
      return temporal.with(date);
   }
}
```

Prüfung

Weil diese Implementierung schon recht komplex ist, empfiehlt sich dringend ein gründlicher Unit Test mit ein paar Testfällen. Nachfolgend lernen Sie, wie man mithilfe von JUnit 5 sehr aussagekräftige Tests erstellen kann. Dazu nutzen wir folgende Tricks:

- Wir verwenden einen zusätzlichen für den Testfall nicht genutzten Parameter, der Infos zum Testfall bereitstellt.
- Bisher haben wir `@CsvSource` genutzt, um kommasepariert Werte angeben zu können. Das ist hier nicht ohne Weiteres möglich, da in den Hinweistexten mit Kommas gearbeitet wird. Als Abhilfe erlaubt es `@CsvSource`, durch die Angabe von `delimiter` das gewünschte Trennzeichen vorzugeben.

```
@ParameterizedTest(name="adjustToPayday({0}) => {1}, {2}")
@CsvSource(value= { "2020-06-21; 2020-06-25; normale Anpassung",
                    "2020-05-27; 2020-06-25; normale Anpassung " +
                                             "auf nächsten Monat",
                    "2020-07-21; 2020-07-24; Freitag, falls 25. " +
                                             "am Wochenende",
                    "2019-12-06; 2019-12-16; Dezember: Mitte Monat " +
                                             "und Montag nach Wochenende",
                    "2019-12-23; 2020-01-24; nächster Monat und Freitag, " +
                                             "falls 25. am Wochenende"},
           delimiterString=";")
void adjustInto(LocalDate startDay, LocalDate expected, String info)
{
    final TemporalAdjuster paydayAdjuster = new NextPaydayAdjuster();

    final Temporal result = paydayAdjuster.adjustInto(startDay);

    assertEquals(expected, result);
}
```

Weil das Ganze wirklich auch den letzten Testmuffel überzeugen sollte, möchte ich noch die informative Ausgabe in Eclipse abbilden.

Abbildung 8-4 *Protokollierung des Unit Tests für* `NextPaydayAdjuster`

8.3 Interoperabilität mit Legacy-Code

Für Migrationen von den alten Datums-APIs ist es von Interesse, welche Klassen sich von der Idee her im alten JDK und im Date and Time API aus JDK 8 entsprechen und wie man zwischen Instanzen der alten und neuen Klassen hin und her konvertieren kann. Insbesondere besteht eine Analogie zwischen den Klassen `Date` und `Instant`. Die Klasse `GregorianCalendar` kann man am ehesten mit der Klasse `ZonedDateTime` vergleichen. Dabei gibt es einige Konvertierungsmethoden – bitte beachten Sie, dass sich diese ausschließlich im alten Datums-API befinden, damit man im eingeführten modernen Date and Time API keine Abhängigkeit auf das alte hat. Allerdings gibt es leider keine Möglichkeit, aus einem `Date` in die gebräuchlichen Klassen `LocalDate`, `LocalTime` oder `LocalDateTime` zu konvertieren – später dazu mehr. Nun möchte ich im folgenden Listing einige der obigen Methoden zur Konvertierung nutzen und darüber hinaus noch die Methode `ofInstant(Instant, ZoneId)` einsetzen, die sowohl in der Klasse `LocalDateTime` als auch in der Klasse `ZonedDateTime` existiert:

```
public static void main(final String[] args)
{
    // Berechnungen basierend auf Date
    final Date now = new Date();
    final Instant nowAsInstant = now.toInstant();
    final Date nowFromAsInstant = Date.from(nowAsInstant);

    final ZoneId systemZone = ZoneId.systemDefault();
    final LocalDateTime localDateTime = LocalDateTime.ofInstant(nowAsInstant,
                                                                systemZone);
    final ZoneId zoneCalifornia = ZoneId.of("America/Los_Angeles");
    final ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(nowAsInstant,
                                                                zoneCalifornia);

    // Berechnungen basierend auf Calendar
    final GregorianCalendar nowAsCalendar = new GregorianCalendar();
    final ZonedDateTime nowAsZonedDateTime = nowAsCalendar.toZonedDateTime();
    final GregorianCalendar calendarFromZoned =
                            GregorianCalendar.from(nowAsZonedDateTime);
    final Instant instant = nowAsZonedDateTime.toInstant();

    System.out.println("LocalDateTime: " + localDateTime);
    System.out.println("ZonedDateTime: " + zonedDateTime);
    System.out.println("Instant:       " + instant);
}
```

Listing 8.22 *Ausführbar als* **'LEGACYEXAMPLE'**

Das Programm LEGACYEXAMPLE erzeugt in etwa folgende Ausgaben:

```
LocalDateTime: 2020-09-16T12:51:05.744
ZonedDateTime: 2020-09-16T03:51:05.744-07:00[America/Los_Angeles]
Instant:       2020-09-16T10:51:05.825Z
```

Konvertierung `Date` in `LocalDateTime` usw. und zurück

Die Konvertierung von `Date` in Instanzen von `LocalDateTime` usw. ist leider nicht so leicht, wie man es sich wünschen würde. Statt einer direkten Konvertierung muss man einen Umweg über einen `Instant` nehmen. Aus diesem kann man dann ein `LocalDateTime`-Objekt erzeugen – allerdings ist dabei eine Zeitzone anzugeben. Danach kann man die jeweils spezifischen Klassen `LocalDate`- und `LocalTime`-Objekte durch einen einfachen Methodenaufruf ermitteln:

```
public static void main(String[] args)
{
    // Konvertierungen von ZonedDateTime -> Date
    final ZonedDateTime nowZdt = ZonedDateTime.now();
    final Date nowAsDate1 = Date.from(nowZdt.toInstant());
    System.out.println("From ZonedDateTime:      " + nowAsDate1);

    // Konvertierungen von LocalDateTime -> Date
    final LocalDateTime startLdt = LocalDateTime.parse("2020-09-169T12:54:38");

    // Variante 2 mit Umwandlung erst in ZonedDateTime per atZone()
    final ZonedDateTime zdt = startLdt.atZone(ZoneId.systemDefault());
    final Date nowAsDate2 = Date.from(zdt.toInstant());
    System.out.println("From LocalDateTime V2:   " + nowAsDate2);

    // Variante 3 mit ZoneOffset
    final Date nowAsDate3 = Date.from(startLdt.toInstant(zdt.getOffset()));
    System.out.println("From LocalDateTime V3:   " + nowAsDate3);

    // Konvertierungen von Date -> LocalDateTime usw.
    final Instant instant = Instant.ofEpochMilli(nowAsDate3.getTime());
    final LocalDateTime ldt = LocalDateTime.ofInstant(instant,
                                                      ZoneId.systemDefault());
    System.out.println("LocalDateTime from Date: " + ldt);

    // Umwandlung in LocalDate bzw. LocalTime
    final LocalDate ld = ldt.toLocalDate();
    System.out.println("LocalDate:               " + ld);
    final LocalTime lt = ldt.toLocalTime();
    System.out.println("LocalTime:               " + lt);
}
```

Listing 8.23 *Ausführbar als* '**DATETOLOCALDATEEXAMPLE**'

Das Programm DATETOLOCALDATEEXAMPLE produziert etwa folgende Ausgaben:

```
From ZonedDateTime:      Wed Sep 16 12:54:28 CEST 2020
From LocalDateTime V2:   Wed Sep 16 12:54:38 CEST 2020
From LocalDateTime V3:   Wed Sep 16 12:54:38 CEST 2020
LocalDateTime from Date: 2020-09-16T12:54:38
LocalDate:               2020-09-16
LocalTime:               12:54:38
```

9 Applikationsbausteine

Applikationen auf Basis vorgefertigter, funktionierender Bausteine zu konstruieren, statt Dinge immer wieder von Grund auf neu zu entwerfen, macht die Softwareentwicklung zu einer Ingenieurdisziplin. Man vermeidet so, das Rad ständig neu zu erfinden. In diesem Kapitel werden zu einem kleinen Teil eigene wiederverwendbare Bausteine konstruiert sowie insbesondere Funktionalitäten aus frei verfügbaren Bibliotheken beschrieben und eingesetzt. Dazu empfiehlt sich eine Internetrecherche, bevor man loslegt. Jon Bentley bemerkt in seinem empfehlenswerten Buch »Perlen der Programmierkunst« [3] dazu treffend: »Wenn eine vom System bereitgestellte [...] Funktion Ihren Bedürfnissen genügt, sollten Sie noch nicht einmal daran denken, eine eigene zu schreiben.« Auf diesen Sachverhalt weist auch Joshua Bloch in seinem hervorragenden Buch »Effective Java« [6] in Item 47 »Know and use the libraries« hin. Demzufolge bedienen wir uns nachfolgend ausgiebig in verschiedenen Bibliotheken.

Während ich in den ersten beiden Auflagen dieses Buchs vor allem eigene Entwicklungen von Utility-Klassen gezeigt habe, tue ich dies seit der dritten Auflage nur noch exemplarisch und konzentriere mich stattdessen auf die Bibliothek Google Guava und gehe ab und an auf Apache Commons als Alternative und Ergänzung ein. Für den Fall, dass man doch einmal eine eigene Utility-Klasse erstellt, sollte die Funktionalität durch Unit Tests gründlich geprüft werden. Die später vorgestellten Bausteine der Bibliothek Google Guava sind mit rund 300.000 (!) Tests abgesichert und werden zudem in einer Vielzahl von Google-Produkten (und mittlerweile auch vielen anderen) genutzt. Eine derartige Qualität wird man wohl kaum selbst erreichen können.

In Abschnitt 9.1 motiviere ich, dass sich durch den Einsatz von Bibliotheken Aufgaben einfacher realisieren lassen und der entstehende Sourcecode besser lesbar wird. Das zeigt Abschnitt 9.2 unter Verwendung der Bibliothek Google Guava. Diese bietet verschiedene nützliche Funktionalitäten, die einem das Leben als Java-Entwickler erleichtern können. Abschnitt 9.3 thematisiert Wertebereichs- und Parameterprüfungen, die dabei helfen, ungültige Parameterwerte und Objektzustände zu vermeiden. Aufgedeckte Probleme sollte man für spätere Analysen oder Fehlersuchen protokollieren, wozu sich ein Logging-Framework anbietet. Dieses kann zur Aufzeichnung von Aktionen oder zur Protokollierung von Wertebelegungen in Fehlersituationen dienen. Die Thematik Logging beschreibe ich in Abschnitt 9.4. Abschließend stellt Abschnitt 9.5 das Thema Konfiguration vor, unter anderem die Auswertung von Kommandozeilenparametern und die Parametrierung mithilfe der JDK-Klasse `Properties`.

9.1 Einsatz von Bibliotheken

Der Umfang des JDKs und die Anzahl frei verfügbarer Bibliotheken nimmt ständig zu, und es wird dadurch immer schwieriger, die Übersicht zu behalten. Auf jeden Fall sollte man einige zentrale Elemente des JDKs kennen und sicher anwenden können, unter anderem das bereits besprochene Collections-Framework.

Muss man sich in ein Thema einarbeiten, so bietet sich eine Internetrecherche an. Die Oracle-Seiten liefern einen ersten Überblick. Allerdings sind die dort vorgestellten Beispiele zum Teil nicht für den Einsatz in Produktivsoftware geeignet, sondern stellen eher sogenannte Schönwettersoftware dar. Diese Beispiele enthalten nämlich meistens keine angemessene Fehlerbehandlung und sollten daher nur als Ideen für eigene Applikationsbausteine dienen.

Eine umfangreiche Quelle für Utility-Klassen ist die Sammlung der Apache Commons-Bibliotheken[1]. Die dort angebotenen Bausteine können einem das Programmiererdasein sehr erleichtern. Ähnliches gilt für die Bibliothek Google Guava[2]. Während Apache Commons ihre Funktionalität in jeweils separaten JAR-Dateien bereitstellt, umfasst Google Guava nur eine JAR-Datei. Exemplarisch sind gebräuchliche Bestandteile bzw. deren Referenzierung für die Gradle-Build-Datei nachfolgend (Stand August 2020) angegeben:

```
// Apache Commons
compile 'org.apache.commons:commons-lang3:3.11'
compile 'commons-io:commons-io:2.7'
compile 'org.apache.commons:commons-collections4:4.4'

// Google Guava
compile 'com.google.guava:guava:29.0-jre'
```

Motivation zum Einsatz von Bibliotheken

Beim Programmieren sollte man das Ziel verfolgen, möglichst verständlich und elegant zu entwickeln. Damit ist unter anderem gemeint, dass keine unnötige Komplexität entsteht, der Sourcecode gut lesbar ist usw. Die Realität sieht aber leider häufig anders aus. Man findet unkommentierte, teils schlecht nachvollziehbare oder auch wüst strukturierte sowie oftmals viel zu lange Methoden. Darüber hinaus muss man sich mit diversen Implementierungsdetails wie komplizierten Indexberechnungen in `for`-Schleifen, Sonderbehandlungen von Randfällen oder tief verschachtelten `if`-Zweigen mit komplexen Bedingungen beschäftigen. Gebilde wie das Folgende kennen Sie bestimmt:

```
// Beispiel: Zu viel Komplexität
if (!(str == null || str.isEmpty()))
{
    if (skipValidation || (str.length() > 5 && lengthCheckActivated))
    {
```

[1] http://commons.apache.org/
[2] https://code.google.com/p/guava-libraries/

All dies lenkt von der eigentlichen Business-Funktionalität ab und macht den Sourcecode teilweise so schlecht nachvollziehbar, dass der Sinn und zum Teil der Ablauf nicht mehr erkennbar sind. Durch den Einsatz der praktischen Funktionalitäten aus Apache Commons und Google Guava lässt sich mit weniger eigenem Sourcecode viel mehr erreichen, als wenn man alles selbst programmieren würde – außerdem muss man sich nicht zum 100sten Mal um Dinge kümmern, die schon längst gelöst sind.

Um diese Argumentation nachzuvollziehen, betrachten wir als Beispiel eine Gültigkeitsprüfung. Nehmen wir an, wir sollten für Texteingaben absichern, dass diese weder `null` noch leer sind und auch nicht nur aus Whitespaces bestehen (z. B. für GUI-Textfelder). Dazu nutzen wir Java-Bordmittel, Google Guava und Apache Commons:

```
public static boolean isValidString_Using_JDK(final String input)
{
   // Ab JDK 11: return input != null && !input.isBlank();
   return input != null && !input.trim().isEmpty();
}

public static boolean isValidString_Using_Guava(final String input)
{
   return !Strings.isNullOrEmpty(input) && !input.trim().isEmpty();
}

public static boolean isValidString_Using_Commons(final String input)
{
   return StringUtils.isNotBlank(input);
}
```

Wie man sieht, nimmt die Komplexität beim Einsatz einer passenden Bibliothek ab und die Lesbarkeit und Verständlichkeit steigt.[3] Offensichtlich ist die Klasse `StringUtils` aus Apache Commons am besten für die hier benötigte Prüfung geeignet. In anderen Anwendungsfällen ist Google Guava etwas angenehmer in der Handhabung. Allerdings holt diesbezüglich auch die als Kommentar gezeigte Java-11-Variante auf.

Hinweis: Lizenzproblematik beim Einsatz von Fremdbibliotheken

Auch wenn in der Regel der Einsatz von Bibliotheken empfehlenswert ist, um gewisse Funktionalitäten nicht selbst realisieren zu müssen, so sollten Sie sich über das Lizenzmodell der jeweiligen Bibliothek informieren, bevor Sie sie nutzen. Wichtige Lizenzen sind unter anderem GPL (GNU Public License), LGPL (Lesser GPL) sowie die Apache License. Stellen Sie sicher, dass die Lizenz den Einsatz der Bibliothek in Ihrem Kontext erlaubt. GPL beispielsweise fordert, dass die eigenen Sourcen veröffentlicht werden müssen, falls man die GPL-lizenzierten Bibliotheken in eigene Programme einbindet. Für viele kommerzielle Projekte ist das nicht akzeptabel.

Die hier beschriebenen Bibliotheken Apache Commons und Google Guava stehen unter der recht lockeren Apache License Version 2.0, wodurch sie problemlos auch in kommerziellen Projekten verwendet können, ohne dass man die eigenen Sourcen veröffentlichen muss.

[3] Allerdings sieht man durch die Indirektion nicht genau, was geprüft wird, und muss sich darauf verlassen, dass dort auch das Richtige geschieht.

Beispiel: Apache Commons Lang `StringUtils`

Verschiedene Aktionen und Prüfungen auf Strings sind sehr gebräuchlich, z. B. der Test auf einen leeren String oder das links- bzw. rechtsbündige Ausrichten. Früher wurde wohl in nahezu jedem Projekt eine eigene `StringUtils`-Klasse erstellt. Diesen »Eigengewächsen« ist etwa die Klasse `org.apache.commons.lang3.StringUtils` aus den Apache Commons Lang vorzuziehen. Sie bietet u. a. folgende Methoden:

- `isEmpty(String)` – Prüft, ob der übergebene String leer (`""` oder `null`) ist.
- `isBlank(String)` – Prüft, ob der übergebene String nur Whitespaces enthält, leer oder gar `null` ist.
- `leftPad(String, int, String)` bzw. `rightPad(String, int, String)` – Füllt einen String vom Anfang bzw. vom Ende mit dem angegebenen Zeichen bzw. der Zeichenfolge bis zur gewünschten Länge auf.
- `abbreviate(String, int)` – Verkürzt einen String auf die übergebene Länge, indem der hintere Teil abgeschnitten und durch drei Auslassungspunkte, eine sogenannte Ellipsis (`...`), ersetzt wird.
- `abbreviateMiddle(String, String, int)` – Verkürzt einen String auf die übergebene Länge, indem in der Mitte Zeichen entfernt werden und als Auslassungszeichenfolge die übergebene genutzt wird.

Setzen wir diese Methoden wie folgt ein:

```
public static void main(final String[] args)
{
    // Spezielle Prüfungen
    System.out.println("isEmpty:          " + StringUtils.isEmpty("   "));
    System.out.println("isBlank:          " + StringUtils.isBlank("   "));
    System.out.println("isBlank/null:     " + StringUtils.isBlank(null));

    // Ausrichtung
    final String rightAligned = StringUtils.leftPad("Right", 15, "*");
    System.out.println("leftPad:          " + rightAligned);

    // Abkürzungen
    final int maxWidth = 15;
    final String longText = "This is a long text that has to be shortened!";
    final String shortened = StringUtils.abbreviate(longText, maxWidth);
    final String shortened2 = StringUtils.abbreviateMiddle(longText,
                                                           "...", maxWidth);
    System.out.println("abbreviate:       " + shortened);
    System.out.println("abbreviateMiddle: " + shortened2);
}
```

Listing 9.1 *Ausführbar als* **'STRINGUTILSEXAMPLE'**

Vom Programm STRINGUTILSEXAMPLE werden folgende Ausgaben produziert:

```
isEmpty:          false
isBlank:          true
isBlank/null:     true
leftPad:          **********Right
abbreviate:       This is a lo...
abbreviateMiddle: This i...tened!
```

Fallstrick – Do It Yourself

Bestimmt kennen Sie folgendes Phänomen: Obwohl mittlerweile viele ausgezeichnete Bibliotheken mit umfangreicher und sehr gut getesteter Funktionalität existieren, gibt es vereinzelt leider immer noch Entwickler, die verschiedene Hilfsfunktionen lieber selbst programmieren, anstatt auf die Bibliotheken zurückzugreifen. Allerdings ist ein Großteil dessen, was an Utility-Funktionalitäten selbst realisiert wird, auch Bestandteil beispielsweise in Apache Commons oder Google Guava – mit dem Vorteil, dass die dort implementierte Funktionalität erprobt und ausgereift ist.

Inzwischen hat die komfortable Verwaltung von Abhängigkeiten durch Tools wie Maven oder Gradle das Einbinden von Fremdbibliotheken derart vereinfacht, dass glücklicherweise nur noch seltener eigene Utility-Klassen entstehen, wenn es Vergleichbares bereits gibt.

Hinweis: Korrekturen in (eigenen) Utility-Klassen

Früher hat man vielfach Utility-Funktionalität selbst erstellt. Mitunter hat sich dabei (trotz Unit Tests) auch mal ein Fehler eingeschlichen. Wenn man diesen nun behebt, gilt es Folgendes zu bedenken: Möglicherweise haben Aufrufer um einige Fehler herumprogrammiert oder ein Fehlverhalten sogar als Feature missbraucht. Dann führt eine Korrektur in der Bibliothek zu Problemen bei diesen Nutzern. Eine allgemeingültige Lösung für diese Problematik gibt es nicht. Im besten Fall, wenn alle Aufrufer bekannt sind, können diese über eine Korrektur informiert werden, sodass benötigte Anpassungen in nutzendem Sourcecode erfolgen können.

In der Regel kennt man jedoch nicht alle Klienten. Dadurch wird die Lage etwas diffiziler. Wenn die Fehlerkorrektur in den eigenen Utility-Klassen jedoch Priorität vor Rückwärtskompatibilität hat, kann man nicht immer Rücksicht auf sich falsch verhaltende Nutzer nehmen. Eine mögliche Variante, die weniger Probleme bei Klienten macht, besteht darin, eine fehlerhafte Methode als veraltet zu markieren (Annotation `@Deprecated` und optional Javadoc-Kommentar `@deprecated`) und eine neue Methode ins API aufzunehmen.

Die bisherige Argumentation führt aber auch zu dem Schluss, das ähnliche Probleme mit Fremdbibliotheken auftreten können. Durch ein Update auf eine neuere Version handelt man sich unter Umständen unerwartet Inkompatibilitäten ein. Demnach sollte man nicht blindlings auf neue Versionen von Fremdbibliotheken aktualisieren.

9.2 Google Guava im Kurzüberblick

Nach einem ersten kurzen Blick auf Apache Commons wollen wir uns nun mit Google Guava beschäftigen. Dabei werde ich auf ähnliche Funktionalität in Apache Commons hinweisen.

Obwohl die im JDK bereitgestellte Funktionalität durchaus beachtlich ist, gibt es immer wieder mal Bereiche und Anwendungsfälle, wo man spezifische Hilfs-

funktionalität benötigt, die im JDK nicht vorhanden ist. Dann empfiehlt sich der Einsatz von diesen Bibliotheken. Google Guava hat sich als ergiebige Quelle nützlicher Dinge entwickelt, unter anderem Folgender:

- String-Aktionen
- Stringkonkatenation und -extraktion
- Erweiterungen für Collections
- Weitere Utility-Funktionalitäten

Verbesserungen durch JDK 7 und 8 Mit Java 7 und insbesondere Java 8 findet man viel Funktionalität, die zuvor nur in Fremdbibliotheken existierte, nun in ähnlicher Form im JDK selbst. Auch Java 9, 10 und 11 bringen einige hilfreiche Neuerungen.

Folgende Tabelle 9-1 zeigt, wie sich Funktionalitäten aus Guava auf die neuen Bestandteile in JDK 7 und 8 abbilden lassen:

Tabelle 9-1 *Abbildung von Guava auf JDK-Funktionalität*

Klasse in Guava	Analogon in JDK 7 / 8
`com.google.common.base.Objects`	`java.util.Objects`
`com.google.common.base.Preconditions`	`java.util.Objects`
`com.google.common.collect.Ordering`	`java.util.Comparator`
`com.google.common.base.Optional`	`java.util.Optional`
`com.google.common.io.Files`	NIO 2
`com.google.common.base.Function`	Lambdas

9.2.1 String-Aktionen

In Google Guava stehen diverse Stringmanipulationsmöglichkeiten bereit – jedoch sind diese nicht so umfangreich wie die in der schon kurz vorgestellten Klasse `StringUtils` aus Apache Commons. Schauen wir uns nun die Klasse `Strings` aus Guava an, die unter anderem folgende Methoden bietet:

- `isNullOrEmpty(String)` – Prüft, ob ein String leer oder `null` ist.
- `nullToEmpty(String)` – Wandelt eine `null`-Referenz in einen Leerstring um.
- `emptyToNull(String)` – Ein Leerstring wird auf den Wert `null` abgebildet.
- `padStart(String, int, char)` bzw. `padEnd(String, int, char)` – Fügt einem String ein bestimmtes Zeichen so lange am Anfang bzw. am Ende hinzu, bis der String die gewünschte angegebene Länge erreicht hat. Damit kann man eine links- bzw. rechtsbündige Ausrichtung erzielen.

Ähnlich wie schon zuvor schreiben wir ein kleines Programm zur Demonstration der Funktionalität der Prüfmethoden, des Mappings und der Ausrichtung:

```
public static void main(final String[] args)
{
    // Prüfmethoden
    System.out.println(Strings.isNullOrEmpty(""));
    System.out.println(Strings.isNullOrEmpty("   "));

    // Mapping
    System.out.println("'" + Strings.emptyToNull("")+ "'");
    System.out.println("'" + Strings.nullToEmpty(null) + "'");

    // Ausrichtung
    final String rightAligned = Strings.padStart("Right", 15, '*');
    System.out.println(rightAligned);
}
```

Listing 9.2 *Ausführbar als* '**STRINGSEXAMPLE**'

Startet man das Programm STRINGSEXAMPLE, so kommt es zu folgender Ausgabe:

```
true
false
'null'
''
**********Right
```

Analog in Apache Commons Ähnliche Funktionalitäten wie in der Klasse `Strings` findet man in Apache Commons mit der bereits vorgestellten Klasse `StringUtils`. Letztere bietet noch mehr Funktionalität, etwa diverse Prüfungen und die Manipulation von Strings, z. B. das Abkürzen von Texten. Für weitere Manipulationsfunktionalität lohnt sich ein Blick auf die Klasse `WordUtils` in Apache Commons Lang.

9.2.2 Stringkonkatenation und -extraktion

Im JDK existierte vor JDK 8 wenig Funktionalität zur Stringmanipulation – insbesondere zur Konkatenation sowie Extraktion. Google Guava bietet die folgenden zwei Klassen:

- `Joiner` – Verknüpfung von Strings mit optionaler Parametrierung
- `Splitter` – Extraktion basierend auf einer Parametrierung und Konfiguration

Die Klasse `Joiner`

Die Klasse `com.google.common.base.Joiner` ermöglicht mit der Methode `join()` das Zusammenfügen von Zeichenketten unter Verwendung von konfigurierbaren Trennzeichen. Konfigurierbar ist auch, wie `null`-Werte verarbeitet werden. Wird dies nicht spezifiziert und enthält die Eingabe `null`-Werte, wird beim Aufruf von `join()` eine

`NullPointerException` ausgelöst. Manchmal sollen die einzelnen Werte in Hochkommata eingeschlossen werden – das ist etwa dann nützlich, wenn man Leerzeichen in den Werten sichtbar machen möchte. Dazu sind lediglich Hochkommata vor und nach dem Aufruf sowie im Verknüpfungsmuster selbst anzugeben:

```
public static void main(final String[] args)
{
   // Verschiedene Demodaten
   final List<String> words = Arrays.asList("Text", "is", "concatenated");
   final List<String> withNulls = Arrays.asList("Skip", "null", null, "values");
   final List<String> names = Arrays.asList("Tim B. ", " Mike I.", " Andy S. ");

   // Einfache Verknüpfung mit Trennzeichenfolge
   System.out.println(Joiner.on("-+-").join(words));

   // Verknüpfung mit null-Werten
   System.out.println(Joiner.on(" ").skipNulls().join(withNulls));

   // Spezielle Markierung von Werten mit Hochkommata
   final String markedValues = "'" + Joiner.on("','").join(names) + "'";
   System.out.println(markedValues);
}
```

Listing 9.3 *Ausführbar als* '**JOINEREXAMPLE**'

Im Listing sehen wir die Fabrikmethode `on()`, über die das Trennzeichen bzw. die Trennzeichenfolge festgelegt wird. Das Verbinden der einzelnen Bestandteile geschieht dann durch Aufruf der Methode `join()` und liefert im obigen Fall folgendes Ergebnis:

```
Text-+-is-+-concatenated
Skip null values
'Tim B. ',' Mike I.',' Andy S. '
```

Alternativen in Java 8 Das Zusammenfügen von Texten war bis Java 8 leider nicht Bestandteil des JDKs. Seit Java 8 erlauben es Streams (vgl. Abschnitt 7.1) in Zusammenarbeit mit einem Lambda und der Utility-Funktionalität aus der Klasse `Collectors`, die Stringkonkatenation wie folgt zu schreiben:

```
final String markedValues = names.stream().map(input -> "'" + input + "'").
                                          collect(Collectors.joining(","));
```

Zum anderen kann man für einfache Verknüpfungen auf die Methode `join()` der Klasse `String` zurückgreifen. Allerdings kommt es bereits bei `null`-Werten zu möglicherweise unerwarteten Ausgaben, wenn man Folgendes schreibt:

```
System.out.println(String.join("-+-", words));
System.out.println(String.join(" ", withNulls));
```

Führt man diese Anweisungen aus, so kommt es zu dieser Ausgabe:

```
Text-+-is-+-concatenated
Skip null null values
```

Die Klasse `Splitter`

Mithilfe eines `com.google.common.base.Splitter`s kann man einen Eingabestring in eine Folge von Tokens zerlegen. Ähnliches kennen wir aus dem JDK mit einem `Scanner` (vgl. Abschnitt 5.5.4).

Betrachten wir als Beispiel eine kommaseparierte Eingabe von Namen, wobei bewusst auch ein paar Leereinträge eingefügt sind, um im Anschluss auf einige Besonderheiten eingehen zu können.

```
public static void main(final String[] args)
{
    final String input = "  Mike,,Florian ,    Tim ,, Erkan ";

    final Iterable<String> splittedNames = Splitter.on(',').split(input);
    for (final String name : splittedNames)
    {
        System.out.println(name);
    }
}
```

Listing 9.4 *Ausführbar als* '**SPLITTEREXAMPLE**'

Wie beim `Joiner` dient auch beim `Splitter` die Methode `on()` dazu, die Trennzeichen festzulegen. Das Programm SPLITTEREXAMPLE erzeugt folgende Ausgabe:

```
  Mike

Florian
    Tim

 Erkan
```

Es fällt auf, dass zum einen auch leere Eingaben als Leerstring behandelt werden und zum anderen alle Whitespaces rund um die Eingaben erhalten bleiben. Beides ist häufig in der Praxis so nicht gewünscht. Derartiges zu entfernen bedarf dann wieder einiger Sonderbehandlungen, wenn man nur Funktionalität aus dem JDK nutzt.

Betrachten wir, wie wir die Anforderungen durch Parametrierung des `Splitter`s lösen können. Dieser bietet unter anderem folgende Methoden: `trimResults()`, `omitEmptyStrings()` sowie `limit(int)`:

```
public static void main(final String[] args)
{
    final String input = "  Mike,,Florian ,    Tim ,, Erkan ";

    final List<String> splittedNames = Splitter.on(',').trimResults().
                                       omitEmptyStrings().limit(3).
                                       splitToList(input);

    for (final String name : splittedNames)
    {
        System.out.println(name);
    }
}
```

Listing 9.5 *Ausführbar als* '**SPLITTERIMPROVEDEXAMPLE**'

Mit der im Listing gezeigten Parametrierung überspringen wir leere Eingaben (`omitEmptyStrings()`) und beschneiden die Ergebnisse (`trimResults()`). Des Weiteren begrenzen wir die zu liefernden Treffer auf 3 (`limit(3)`). Hier verwenden wir den Aufruf von `splitToList(String)`, der als Ergebnis eine `List<String>` liefert. Das ist teilweise praktischer als ein `Iterable<String>`, wie es die zuvor genutzte Methode `split(String)` zurückgibt. Mit der Beschränkung auf drei Elemente erfolgt nach Erkennen von drei Vorkommen keine weitere Verarbeitung des Reststrings, wodurch bei der obigen Eingabe diese Ausgabe produziert wird:

```
Mike
Florian
Tim ,, Erkan
```

Hinweis: Diskussionswürdige Ausgabe

Mitunter hört man Verwunderung über den letzten Teil der Ausgabe, der durch die fehlende weitere Auswertung zustande kommt. Es wird argumentiert, dass die Ausgabe ähnlich wie bei einer Subliste wie folgt hätte aussehen sollen:

```
Mike
Florian
Tim
```

Analog in Apache Commons Ähnliches zu den Klassen `Joiner` und `Splitter` findet man in Apache Commons in der Klasse `StringUtils` und deren Methoden `split()` sowie `join()`.

Praxisbeispiel

Eingangs des Kapitels habe ich betont, dass der Einsatz von Bibliotheken die Arbeit erleichtern kann, weil man sich dadurch in der Regel mit weniger Details und Sonderfällen zu beschäftigen hat. Das wollen wir nun an den schon genutzten kommaseparierten Daten (CSV) nachvollziehen.[4] Einige Ideen hierzu habe ich aus Holger Staudachers Blog (`http://eclipsesource.com/blogs/2012/07/26/having-fun-with-guavas-string-helpers/`) mit dessen persönlicher Erlaubnis übernommen.

CSV-Daten sind in der Praxis weit verbreitet. Daher ist es nicht ungewöhnlich, dass man Klassen zur Konvertierung von Daten in und aus CSV erstellt. Obwohl das einfach klingt, gibt es dabei den einen oder anderen Stolperstein oder Spezialfall zu beachten: Bei der Aufbereitung von CSV muss nach dem letzten Eintrag kein Komma mehr geschrieben werden. Zum Teil sollen bei der Wandlung in CSV `null`-Werte nicht in

[4] CSV steht für Comma Separated Values. Allerdings spricht man häufig auch von CSV, obwohl als Trennzeichen z. B. ein Semikolon genutzt wird. Den Vorteil durch Einsatz von Bibliotheken haben Sie natürlich analog bei ähnlichen Trennzeichen.

die Ausgabe übernommen werden – das kann problematisch werden, wenn die Positionen spezielle Bedeutungen tragen oder Attributen zugeordnet sind. Dann darf man `null`-Werte nicht auslassen. Bitte beachten Sie, dass die hier gezeigte, vereinfachte CSV-Verarbeitung nur dann korrekt arbeitet, wenn die Eingaben selbst ***keine*** Kommas enthalten.

CSV-Verarbeitung mit JDK-Klassen Folgende Methode nutzt lediglich JDK-Bordmittel, um die Umwandlung einer Liste von Werten in einen kommaseparierten String zu realisieren, und sensibilisiert für Probleme:

```
public static String toCommaSeparatedString(final List<String> inputs)
{
    if (inputs == null || inputs.isEmpty())
        return "";

    final StringBuilder result = new StringBuilder();
    final Iterator<String> it = inputs.iterator();
    while (it.hasNext())
    {
        final String value = it.next();
        if (value != null)  // skip null
        {
            result.append(value);
            if (it.hasNext())
                result.append(",");
        }
    }
    return result.toString();
}
```

Es ist leicht ersichtlich, dass hier einige `if`-Abfragen und Sonderbehandlungen erfolgen, um leere Eingaben, `null`-Werte usw. korrekt zu behandeln. Dadurch ist die eigentliche Logik der kommaseparierten Aufbereitung nur noch mit Mühe erkennbar.

Der Vollständigkeit halber schauen wir kurz auf die inverse Aktion, die aus einem kommaseparierten String eine `List<String>` erzeugt:

```
public static List<String> fromCommaSeparatedString(final String input)
{
    if (input == null || input.isEmpty())
        return new ArrayList<>();

    final List<String> result = new ArrayList<>();
    final String[] tokens = input.split( "," );
    for (final String token : tokens)
    {
        result.add(token.trim());
    }
    return result;
}
```

Beide Methoden sind funktional so weit in Ordnung, aber reicht das? Es hängt stark davon ab: Die tiefe Implementierungsebene, die Sonderbehandlungen, die Komplexität und die Länge sind in beiden Fällen schon ein wenig zu kritisieren. Aber was ist beispielsweise mit der Situation, wenn Erweiterungen erfolgen sollen oder das Verhalten

für leere Eingabewerte geändert werden muss? Diese Änderungen ohne Fehler zu realisieren, ist nicht ganz so einfach. Dann profitiert man davon, wenn es einige Unit Tests gibt, die die Funktionalität prüfen. Details zum Unit-Testen finden Sie in Kapitel 20.

CSV-Verarbeitung mit Guava Betrachten wir die Umsetzung unter Einsatz von Google Guava, die hier ohne weitere Erklärung gezeigt wird, da wir die beiden genutzten Klassen `Joiner` und `Splitter` bereits zuvor etwas genauer kennengelernt haben. Ganz offensichtlich ist die folgende Lösung viel kürzer, besitzt nahezu keine sichtbare Komplexität und kommuniziert sehr klar, was gemacht werden soll.

```
public static String toCommaSeparatedString(final List<String> inputs)
{
    return Joiner.on(",").skipNulls().join(inputs);
}

public static List<String> fromCommaSeparatedString(final String input)
{
    return Splitter.on(",").omitEmptyStrings().trimResults().splitToList(input);
}
```

9.2.3 Erweiterungen für Collections

In den nachfolgenden Abschnitten schauen wir uns einige nützliche Funktionalitäten aus Guava rund um Collections an.

Mengenoperationen mit Sets

Die Methoden aus dem Interface `Collection<E>` und somit auch von `Set<E>` ermöglichen die Bestimmung der Differenz-, Schnitt- sowie Vereinigungsmenge (vgl. Abschnitt 6.1.3). Die dazu genutzten Methodennamen wie `retainAll()`, `removeAll()` usw. sind jedoch eher technischer Natur und beschreiben die algorithmische, nicht aber die mathematische Entsprechung – je nach Problemstellung ist das eine oder andere besser passend. Im mathematischen Kontext erscheint mir Guava von der Namensgebung her klarer und intuitiver. Dort existiert eine Utility-Klasse `com.google.common.collect.Sets` mit unter anderen diesen Methoden:

- `difference(Set<E> set1, Set<?> set2)` – Berechnet die Differenzmenge, also all diejenigen Elemente, die in `set1` sind, nicht aber in `set2`.
- `intersection(Set<E> set1, Set<?> set2)` – Ermittelt die Schnittmenge. Das sind die Elemente, die sowohl in `set1` als auch in `set2` enthalten sind.
- `union(Set<? extends E> set1, Set<? extends E> set2)` – Berechnet die Vereinigungsmenge, die alle Elemente aus `set1` und `set2` enthält.
- `symmetricDifference(Set<? extends E> set1, Set<? extends E> set2)` – Ermittelt die symmetrische Differenz. Diese ist definiert als die Menge der Elemente, die entweder in `set1` oder in `set2`, nicht aber in beiden enthalten sind. Formal ist es die Kombination von $difference(union, intersection)$.

Erstellen wir ein kleines Programm, um die Funktionalität zu erkunden:

```
public static void main(final String[] args)
{
    final Set<String> names = ImmutableSet.of("Andy", "Mike", "Tim", "Peter");
    final Set<String> readers = ImmutableSet.of("Sagi", "Mike", "Tim", "Jörg");

    final Set<String> intersection = Sets.intersection(names, readers);
    final Set<String> union = Sets.union(names, readers);
    final Set<String> difference1 = Sets.difference(names, readers);
    final Set<String> difference2 = Sets.difference(readers, names);
    final Set<String> symDifference1 = Sets.symmetricDifference(names, readers);
    final Set<String> symDifference2 = Sets.symmetricDifference(readers, names);
    final Set<String> symDifference3 = Sets.difference(union, intersection);

    System.out.println("intersection:   " + intersection);
    System.out.println("union:          " + union);
    System.out.println("difference1:    " + difference1);
    System.out.println("difference2:    " + difference2);
    System.out.println("symDifference1: " + symDifference1);
    System.out.println("symDifference2: " + symDifference2);
    System.out.println("symDifference3: " + symDifference3);
}
```

Listing 9.6 *Ausführbar als* '**SETSEXAMPLE**'

Die Programmausgaben verdeutlichen die oben beschriebene Arbeitsweise:

```
intersection:   [Mike, Tim]
union:          [Andy, Mike, Tim, Peter, Sagi, Jörg]
difference1:    [Andy, Peter]
difference2:    [Sagi, Jörg]
symDifference1: [Andy, Peter, Sagi, Jörg]
symDifference2: [Sagi, Jörg, Andy, Peter]
symDifference3: [Andy, Peter, Sagi, Jörg]
```

Analoges für Listen und Maps Google Guava enthält zwei weitere Utility-Klassen namens `Lists` und `Maps`, die für Listen und Maps adäquate Funktionalitäten anbieten. Diese Klassen findet man im Package `com.google.common.collect`.

Analog in Apache Commons Ähnliches zu den drei Klassen `Sets`, `Lists` und `Maps` findet man in Apache Commons Collections. Dort gibt es die Utility-Klasse `org.apache.commons.collections4.CollectionUtils`, die den allgemeineren Typ `Collection<E>` nutzt, wo also keine Einschränkung auf den Typ `Set<E>` vorgenommen wird. Das hat zum einen den Vorteil, für mehr Typen anwendbar zu sein, zum anderen aber den Nachteil, dass etwa die Vereinigung oder Differenz auf Listen nicht sofort intuitiv sind, z. B. im Hinblick auf doppelte Einträge.

Das Interface `Multimap`

Immer wieder gibt es Situationen, in denen man etwas komplizierte Abbildungen in Form einer Map realisieren muss, bei denen einem Schlüssel eine Menge von Werten zugewiesen werden kann. Im JDK steht für diese Anforderung keine passende Klasse bereit. Guava bietet dafür das Interface `com.google.common.collect.Multimap` und Spezialisierungen davon.

Das folgende Beispiel zeigt eine Abbildung von Ländern auf deren größte Städte:

```
public static void main(final String[] args)
{
    final Multimap<String, String> countryToBigCities =
                                   ArrayListMultimap.create();
    countryToBigCities.put("Germany", "Berlin");
    countryToBigCities.put("Germany", "Hamburg");
    countryToBigCities.put("Germany", "Munich");
    countryToBigCities.put("Germany", "Cologne");
    countryToBigCities.put("Switzerland", "Berne");
    countryToBigCities.put("Switzerland", "Zurich");
    countryToBigCities.put("Switzerland", "Geneve");

    System.out.println("Switzerland: " + countryToBigCities.get("Switzerland"));
    System.out.println("Germany:     " + countryToBigCities.get("Germany"));
}
}
```

Listing 9.7 *Ausführbar als* '**MULTIMAPEXAMPLE**'

Startet man das obige Programm MULTIMAPEXAMPLE, so kommt es zu folgenden Ausgaben, die die Mehrfachverknüpfung zeigen:

```
Switzerland: [Berne, Zurich, Geneve]
Germany:     [Berlin, Hamburg, Munich, Cologne]
```

Analog in Apache Commons Eine bis auf die Schreibweise gleichnamige Alternative zur Klasse `Multimap` findet man in Apache Commons mit dem Basisinterface `org.apache.commons.collections4.MultiMap`.

Umsetzung mit JDK-Mitteln Zur Demonstration, dass eine Realisierung mit JDK-Mitteln schwieriger als mit Guava ist, zeige ich hier eine Map, deren Werte eine Liste beinhalten. Beim Hinzufügen von Werten rufen wir eine Methode `putSpecial()` auf, die die Besonderheit mehrerer Werte für einen Schlüssel kapselt und bei Bedarf eine Liste zur Datenhaltung anlegt. Seit JDK 8 ist dies durch die Methoden `putIfAbsent()` und `computeIfPresent()` ein wenig lesbarer als früher mit JDK 7:

```
public static void main(final String[] args)
{
    final Map<String, List<String>> countryToBigCities = new HashMap<>();
    putSpecial(countryToBigCities, "Germany", "Berlin");
    putSpecial(countryToBigCities, "Germany", "Hamburg");
    putSpecial(countryToBigCities, "Germany", "Munich");
    putSpecial(countryToBigCities, "Germany", "Cologne");
```

```
    putSpecial(countryToBigCities, "Switzerland", "Berne");
    putSpecial(countryToBigCities, "Switzerland", "Zurich");
    putSpecial(countryToBigCities, "Switzerland", "Geneve");

    System.out.println("Switzerland: " + countryToBigCities.get("Switzerland"));
    System.out.println("Germany:     " + countryToBigCities.get("Germany"));
}

private static void putSpecial(final Map<String, List<String>> map,
                               final String key, final String value)
{
    map.putIfAbsent(key, new ArrayList<>());
    map.computeIfPresent(key, (key2, elements) -> { elements.add(value);
                                                    return elements; } );
}
```

Listing 9.8 *Ausführbar als* '**MULTIMAPJDKEXAMPLE**'

Wenn man an wiederverwendbare Bausteine denkt, dann würde man eine Klasse realisieren, die diese Details versteckt. Das wollen wir ganz bewusst hier nicht machen, da es die Funktionalität ja bereits in Google Guava und Apache Commons gibt.

Das Interface `BiMap`

Falls zum Teil die Anforderung einer bidirektionalen Abbildung existiert, also einem Schlüssel ein Wert zugeordnet ist und dieser wieder auf den Schlüssel verweist, lässt sich das zwar mithilfe zweier separater Maps realisieren, jedoch sind dabei dann ein paar Spezialfälle zu bedenken. Einfacher ist es, Spezialisierungen des Interface `com.google.common.collect.BiMap` aus Guava zu nutzen. Der normale Zugriff erfolgt wie üblich mit `get()`. Die Rückabbildung erfragt man auch über `get()`, jedoch nachdem man zuvor `inverse()` auf der Map aufgerufen hat.

Nachfolgend zeige ich diese Zugriffe für eine bidirektionale Abbildung von Ländern auf Hauptstädte (und zurück):

```
public static void main(final String[] args)
{
    final BiMap<String, String> countryToCity = HashBiMap.create();
    countryToCity.put("Germany", "Berlin");
    countryToCity.put("Switzerland", "Berne");

    System.out.println("Switzerland: " + countryToCity.get("Switzerland"));
    System.out.println("Berlin:      " + countryToCity.inverse().get("Berlin"));
}
```

Listing 9.9 *Ausführbar als* '**BIMAPEXAMPLE**'

Folgende Programmausgaben zeigen die bidirektionale Verknüpfung:

```
Switzerland: Berne
Berlin:      Germany
```

Analog in Apache Commons Ähnliches zur `BiMap` findet man in Apache Commons mit dem Basisinterface `org.apache.commons.collections.BidiMap`.

9.2.4 Weitere Utility-Funktionalitäten

In diesem Abschnitt möchte ich Ihnen verschiedene weitere Utility-Funktionalitäten vorstellen. Dabei verzichte ich auf die Darstellung von Guava-Varianten, wenn es eine passende Alternative aus dem JDK gibt.

Die Klasse `Objects`

Die Klasse `com.google.common.base.Objects` erleichtert die Implementierung der Methoden `equals(Object)` und `hashCode()`.

Die Klasse `Objects` aus dem JDK als Alternative Seit JDK 7 kann man alternativ zur Klasse `Objects` aus Guava die gleichnamige Utility-Klasse aus dem JDK nutzen – wobei der Unterschied im einsetzenden Sourcecode minimal ist.

Wir nutzen folgende Klasse `Person` und die Aufzählung `MaritalStatus` zum Familienstand als Ausgangsbasis:

```
public class Person
{
    private final String name;
    private final Color eyecolor;
    private MaritalStatus maritalStatus = MaritalStatus.SINGLE;
    private String nickname = null;
    private int sizeInCm = 0;

    // ...

    @Override
    public boolean equals(final Object object)
    {
        // Kurzform, kompatibel zum Kontrakt, wenn die Klasse final ist
        if (object instanceof Person)
        {
            final Person other = (Person) object;
            return Objects.equals(this.name, other.name) &&
                   Objects.equals(this.eyecolor, other.eyecolor) &&
                   Objects.equals(this.maritalStatus, other.maritalStatus) &&
                   Objects.equals(this.nickname, other.nickname) &&
                   Objects.equals(this.sizeInCm, other.sizeInCm);
        }
        return false;
    }

    @Override
    public int hashCode()
    {
        // Familienstand nicht nutzen, weil veränderlich
        return Objects.hash(name, eyecolor, nickname, sizeInCm);
    }

    @Override
    public String toString()
    {
        return "Person [name=" + name + ", eyecolor=" + eyecolor +
               ", maritalStatus=" + maritalStatus + ", nickname=" +
               nickname + ", sizeInCm=" + sizeInCm + "]";
    }
}
```

```
enum MaritalStatus
{
    SINGLE, MARRIED, DIVORCED, WIDOWED, UNKNOWN;
}
```

Es gibt in Eclipse die Möglichkeit, die Methoden `equals(Object)` und `hashCode()` recht bequem über das Menü SOURCE –> GENERATE HASHCODE() AND EQUALS()... generieren zu lassen. Wenn man dabei jedoch nicht aufpasst und die Checkbox zur Verwendung der Utility-Klasse `Objects` anklickt, dann ist das entstehende Resultat aber komplex und unleserlich.

Auch für die Implementierung von `toString()` kann die Automatik von Eclipse nützlich sein (vgl. Abschnitt 5.1.1), die die zuvor abgebildete `toString()`-Methode generiert. Eine Kleinigkeit in Zusammenhang mit `null`-Werten gibt es noch zu erwähnen: Der entstehende Sourcecode wird recht unschön, wenn man die Option SKIP NULL VALUES anwählt. Dadurch wird folgende wenig lesbare Implementierung generiert:

```
@Override
public String toString()
{
    return "Person [" + (name != null ? "name=" + name + ", " : "") +
           (eyecolor != null ? "eyecolor=" + eyecolor + ", " : "") +
           (maritalStatus != null ? "maritalStatus=" + maritalStatus +
           ", " : "") + (nickname != null ? "nickname=" + nickname +
           ", " : "") + "sizeInCm=" + sizeInCm + "]";
}
```

Zum einen erkennt man, dass hier `null`-Prüfungen für Attribute generiert werden, die nicht optional sind und keine `null`-Werte enthalten dürfen. Zum anderen ist kaum auszumachen, welche Attribute tatsächlich ausgegeben werden. Die Variante mit Guava ist deutlich besser lesbar. Sie ist zudem leichter erweiter- und veränderbar.

Tipp: Vereinfachte Nutzung von APIs durch Utility-Klassen

Zum Teil sind die JDK-APIs wenig intuitiv oder kompliziert anzuwenden. Wie bereits aus den Kapiteln 5 und 6 bekannt, kann selbst die Implementierung der gebräuchlichen Objektmethoden `toString()`, `equals(Object)`, `hashCode()` und `compareTo(T)` mühsam sein.

Durch den Einsatz nicht intuitiver APIs erhöht sich in der Regel der Sourcecode-Umfang, da es zu komplizierten oder merkwürdigen Lösungen kommt. Wenn um API-Probleme herum programmiert wird, so verschlechtert sich dadurch die Lesbarkeit. ***Soll der eigentliche Applikationscode frei von solchen Implementierungsdetails gehalten werden, schreiben oder nutzen wir Utility-Klassen, die die Verwendung unhandlicher APIs erleichtern.***

Die Klasse `Preconditions`

In den vorangegangenen Kapiteln habe ich es immer mal wieder anklingen lassen, dass die Wertebelegung von Attributen oder Variablen bestimmten Bedingungen genügen sollte, etwa dass Eingaben nicht `null` sein dürfen oder einem bestimmten Wertebereich entstammen müssen. Auch hatte ich im Rahmen der Beschreibung des Objektzustands auf Prüfungen hingewiesen, ebenso wie bei der Behandlung von Fehlersituationen.

Zur Verdeutlichung betrachten wir folgende Methode `filter()`, die ihre Parameter »von Hand« wie folgt prüft:

```
public static <T> void filter(final List<T> list, final Condition<T> condition)
{
    if (condition == null)
        throw new IllegalArgumentException("condition must not be null");
    if (list == null)
        throw new IllegalArgumentException("list must not be null");
    if (list.isEmpty())
        throw new IllegalArgumentException("list must not be empty");

    performFilter(list, condition);
}
```

Obwohl Sicherheitsprüfungen dabei helfen können, Fehler schneller zu entdecken, so verschlechtern sie hier doch die Lesbarkeit, da sie auf tieferer Implementierungsebene realisiert sind. Die Verständlichkeit lässt sich durch den Einsatz der Klasse `com.google.common.base.Preconditions` aus Guava deutlich verbessern:

```
public static <T> void filter(final List<T> list, final Condition<T> condition)
{
    Preconditions.checkNotNull(condition, "condition must not be null");
    Preconditions.checkNotNull(list, "list must not be null");
    Preconditions.checkArgument(!list.isEmpty(), "list must not be empty");

    performFilter(list, condition);
}
```

Das ist schon recht gut, aber die Lesbarkeit lässt sich durch den statischen Import nochmals erhöhen – das gilt übrigens auch für diverse andere Utility-Klassen und deren Funktionalität:

```
public static <T> void filter(final List<T> list, final Condition<T> condition)
{
    checkNotNull(condition, "condition must not be null");
    checkNotNull(list, "list must not be null");
    checkArgument(!list.isEmpty(), "list must not be empty");

    performFilter(list, condition);
}
```

Eine Kleinigkeit muss man bei der Methode `checkArgument()` beachten: Die boolesche Bedingung ist für derartige Prüfungen leicht unleserlich, weil man die teilweise benötigte Negation schnell mal übersieht.

Alternative im JDK Seit JDK 7 kann man einige der Prüfungen über die Methode `Objects.requireNonNull()` lösen. Die von der Klasse `Preconditions` gebotenen Möglichkeiten sind jedoch umfangreicher.

Analog in Apache Commons Auch in Apache Commons lassen sich Preconditions prüfen. Dazu gibt es dort die Klasse `org.apache.commons.lang3.Validate`. Im abschließenden Beispiel möchte ich nochmals zeigen, dass Guava die leicht elegantere Variante bietet. Vergleichen Sie die vorherige mit dieser Methode:

```
public static <T> void filter(final List<T> list, final Condition<T> condition)
{
    notNull(condition, "condition must not be null");
    notNull(list, "list must not be null");
    isTrue(!list.isEmpty(), "list must not be empty");

    performFilter(list, condition);
}
```

Tipp: Precondition-Prüfungen und schnelles Fehlschlagen

Ganz allgemein kann man sagen, dass es (während der Entwicklung) praktisch ist, wenn Dinge möglichst schnell schiefgehen (Motto Fail-fast), weil man so Fehlersituationen direkt entdecken und gut beheben kann. Dabei ist es hilfreich, wenn dazu eine Exception ausgelöst wird und diese sachdienliche Hinweise auf die Ursachen eines Problems enthält. Fail-fast ist sicher besser, als irgendwo im Programm aufgrund einer fehlerhaften Eingabe mit einer für den Aufrufer unerwarteten Exception abzustürzen. Schlimmer noch sind sogenannte Silent Fails, die den Fehler verschlucken und einfach so tun, als ob alles in Butter wäre, aber ein falsches Berechnungsergebnis produzieren.

Zusätzliche Vereinfachungen und Erweiterungen

Google Guava bietet über die hier beschriebene Funktionalität hinaus verschiedene Vereinfachungen, etwa bei der Definition von Komparatoren oder für Dateiaktionen. Ebenso gilt dies für die Darstellung optionaler Werte. Auf diese Themen bzw. die korrespondierenden Funktionalitäten in Google Guava gehe ich nachfolgend nicht weiter ein, weil es in JDK 7 bzw. JDK 8 adäquate Alternativen gibt. Für die funktionale Programmierung wurden in Java 8 die Sprachfeatures Lambdas, Methodenreferenzen und funktionale Interfaces hinzugefügt (vgl. Kapitel 4). Auch die Funktionalität von Komparatoren wurde erweitert (vgl. Abschnitt 6.2.4). Darüber hinaus gibt es seit Java 8 die Klasse `Optional<T>` im JDK, die optionale Werte als Objekte modelliert (vgl. Abschnitt 6.5).

9.3 Wertebereichs- und Parameterprüfungen

In diesem Abschnitt stelle ich einige Bausteine zur Wertebereichsprüfung vor. Eine solche Prüfung kann schnell recht aufwendig werden, etwa wenn der erlaubte Wertebereich nicht zusammenhängend ist und man die Prüfung selbst implementiert. Einige Programmierer scheuen dann den mit der Abfrage und der Ausgabe von fehlerhaften Werten verbundenen Aufwand, wodurch eine spätere Fehlersuche allerdings erschwert wird. Mögliche Probleme werden als BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN in Abschnitt 16.3.8 diskutiert. Häufig kann bereits der Einsatz von `enum`-Aufzählungen oder des ENUM-Musters (vgl. Abschnitt 3.4.4) die Arbeit erleichtern. Sind einige Konstanten beispielsweise als `int`-Literale definiert, so können wir gemäß dem Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM (vgl. Abschnitt 17.4.12) vorgehen, um einen eigenständigen Typ zu definieren. Bei umfangreichen Wertebereichen ist der Einsatz dieser Techniken jedoch nicht immer sinnvoll möglich, da man die Werte nicht einzeln benennen möchte oder kann.

Die im Folgenden vorgestellten Bausteine helfen bei Wertebereichsprüfungen. Dass eigene Implementierungen aufwendig werden können, wurde in den vorherigen Auflagen dieses Buchs gezeigt. Weil es deutlich sinnvoller ist, gut getestete Basisbausteine statt selbst geschriebener Hilfsmethoden zu verwenden, nutzen wir nachfolgend die Klassen `Range` und `RangeSet` aus dem Package `com.google.common.collect` aus Google Guava.

Die Klasse `Range`

Die Klasse `com.google.common.collect.Range` erlaubt es auf einfache Weise, Wertebereiche zu beschreiben. Dazu müssen die zu verarbeitenden Typen das Interface `Comparable<T>` erfüllen. Dann lassen sich mithilfe verschiedener Erzeugungsmethoden offene und geschlossene Intervalle konstruieren.[5] Laut mathematischer Definition beinhaltet ein offenes Intervall die Intervallgrenzen nicht, ein geschlossenes schon:

```
public static void main(final String[] args)
{
    final Range<Integer> closed = Range.closed(0, 100);
    final Range<Integer> open = Range.open(0, 100);
    final Range<Integer> openClosed = Range.openClosed(0, 100);
    final Range<Integer> closedOpen = Range.closedOpen(0, 100);

    System.out.println("Closed:     " + closed);
    System.out.println("Open:       " + open);
    System.out.println("openClosed: " + openClosed);
    System.out.println("closedOpen: " + closedOpen);
}
```

Listing 9.10 *Ausführbar als* '**RANGEEXAMPLE**'

[5] Weiterführende Informationen zu den diversen Varianten der Erzeugung finden Sie unter `https://code.google.com/p/guava-libraries/wiki/RangesExplained`.

Startet man das obige Programm RANGEEXAMPLE, so werden die verschiedenen Varianten des Wertebereichs von 0 bis 100 ausgegeben, die sich in der Art der öffnenden und schließenden Klammer unterscheiden: Hier kommt die mathematische Notation mit eckigen und runden Klammern zum Einsatz. Zur Erinnerung: Eine eckige Klammer besagt, dass der Wert ein Teil des Wertebereichs ist, eine runde Klammer gibt an, dass der Wert nicht Bestandteil des Wertebereichs ist.

```
Closed:     [0..100]
Open:       (0..100)
openClosed: (0..100]
closedOpen: [0..100)
```

Weiter gehende Möglichkeiten Die Klasse `Range` bietet aber deutlich mehr als das, was dieses einführende Beispiel zeigt: So kann man etwa auch nach unten oder oben unbegrenzte Wertebereiche definieren sowie abfragen, ob einzelne Werte oder eine Wertemenge in einem `Range` enthalten sind:

```
public static void main(final String[] args)
{
    final Range<Integer> lessThan100 = Range.atMost(99);
    final Range<Integer> moreThan10 = Range.atLeast(11);

    // Prüfe Enthaltensein
    System.out.println("-50 in [...99]: " + lessThan100.contains(-50));
    System.out.println("500 in [...99]: " + lessThan100.contains(500));
    System.out.println("10  in [11...]: " + moreThan10.contains(10));
    System.out.println("100 in [11...]: " + moreThan10.contains(100));

    // Vereinigung
    final Range<Integer> intersection = moreThan10.intersection(lessThan100);
    System.out.println("Intersection:   " + intersection);

    // Obermenge
    final Range<Integer> range_10_25 = Range.closed(10,  25);
    final Range<Integer> range_40_60 = Range.closed(40,  60);
    final Range<Integer> span = range_10_25.span(range_40_60);
    System.out.println("span:           " + span);

    // Prüfe, ob der Wertebreich umschlossen wird, oder erzeuge einen solchen
    final Range<Integer> lessThan1000 = Range.atMost(999);
    System.out.println(lessThan1000.encloses(lessThan100));
    System.out.println(Range.encloseAll(Arrays.asList(5, 100, 500)));
}
```

Listing 9.11 *Ausführbar als* **'RANGEIMPROVEDEXAMPLE'**

Das Programm RANGEIMPROVEDEXAMPLE produziert folgende Konsolenausgaben:

```
-50 in [...99]: true
500 in [...99]: false
10  in [11...]: false
100 in [11...]: true
Intersection:   [11..99]
span:           [10..60]
true
[5..500]
```

Man erkennt sowohl die Prüfungen auf Enthaltensein und die Vereinigung von Wertebereichen als auch die Erweiterung zweier Wertebereiche auf einen umschließenden Wertebereich mit `span()`. Darüber hinaus kann man sich mit `encloseAll()` einen Wertebereich basierend auf einer Menge einzelner Werte erstellen.

Das Interface `RangeSet`

Spezialisierungen von `com.google.common.collect.RangeSet` beschreiben eine Menge nicht verbundener Wertebereiche, die in Form der zuvor vorgestellten Klasse `Range` definiert sind. Man kann einem `RangeSet` verschiedene Wertebereiche hinzufügen, wieder entfernen und prüfen, ob Werte in der Wertebereichsmenge enthalten sind. Als Besonderheit werden beim Hinzufügen sich überlappender Wertebereiche diese miteinander verbunden. Im folgenden Listing sollen die Wertebereiche 0 – 9 und 50 – 70 aus verschiedenen Subwertebereichen kombiniert werden. Normalerweise würde man diese direkt angeben. Hier gestalten wir alles etwas komplizierter, um die Möglichkeiten mit dem Interface `RangeSet` zu erkunden: Der Wertebereich 0 – 9 wird in Form zweier Bestandteile 0 – 5 und 6 – 9 definiert. Letzteres wird wiederum als Zusammenschluss der Intervalle $[6 - 8)$ und $[8 - 10)$ realisiert.

Das eben erlangte Wissen wollen wir nutzen, um verschiedene Aktionen auf einem `RangeSet` auszuführen. Dazu schauen wir uns das folgende Beispiel an:

```
public static void main(final String[] args)
{
    final RangeSet<Integer> rangeSet = TreeRangeSet.create();

    // Bereich 0-9 aus den Teilen 0-5 und 6-9 erstellen
    rangeSet.add(Range.closed(0, 5));
    System.out.println(rangeSet);

    // Zweiter Bereich aus zwei Subbereichen erstellen
    rangeSet.add(Range.closedOpen(6, 8));  // eigenständiger Teil
    rangeSet.add(Range.closedOpen(8, 10)); // wird verbunden zu [6, 10)
    System.out.println(rangeSet);

    // Wertebereich (40-70]
    rangeSet.add(Range.openClosed(40, 70));
    System.out.println(rangeSet);

    // Teilbereich (40-50) entfernen
    rangeSet.remove(Range.open(40, 50));
    System.out.println(rangeSet);

    // Prüfungen auf Enthaltensein
    System.out.println(rangeSet.contains(7));
    System.out.println(rangeSet.contains(55));
    System.out.println(rangeSet.contains(777));
}
```

Listing 9.12 *Ausführbar als* **'RANGESETEXAMPLE'**

Startet man das Programm RANGESETEXAMPLE, so erhält man folgende Ausgaben, die die Arbeitsweise verdeutlichen:

```
[[0..5]]
[[0..5], [6..10)]
[[0..5], [6..10), (40..70]]
[[0..5], [6..10), [50..70]]
true
true
false
```

Fazit

Aus Platzgründen konnte nur ein Einstieg in Wertebereichsprüfungen gegeben werden, aber selbst damit wird deutlich, dass man mithilfe der Klasse `Range` und des Interface `RangeSet` und seinen Spezialisierungen komplexere Wertebereichsprüfungen auf einfache und nachvollziehbare Weise lösen kann. Wollte man dies per Hand realisieren, würde das in deutlich mehr Sourcecode enden. Schlimmer noch: Es wäre auch wesentlich schwieriger wart- und erweiterbar.

9.4 Logging-Frameworks

Neben dem Debugging (vgl. Abschnitt 2.5) stellt das sogenannte ***Logging***, also Programmausgaben über den Zustand des Programms selbst, ein wichtiges Hilfsmittel zur Fehlersuche dar: Durch Log-Ausgaben können interne Abläufe, derzeitige Wertebelegungen usw. im Detail protokolliert werden. Sinnvolle Log-Ausgaben erleichtern sowohl eine mögliche Fehlersuche als auch das Nachvollziehen und Verständnis des Programmablaufs. Diverse Java-Anwendungen produzieren Ausgaben und nutzen dazu im einfachsten Fall eine Konsolenausgabe per `System.out`. Empfehlenswert ist jedoch der Einsatz von Logging-Frameworks. Dieser Abschnitt beginnt mit einer kurzen Beschreibung der Vorteile bei der Verwendung eines solchen Frameworks. Dann stelle ich in Abschnitt 9.4.1 mit log4j2 den Nachfolger des weitverbreiteten Frameworks log4j vor, das für mindestens 10 Jahre der De-facto-Standard war. Mittlerweile gibt es jedoch einige Alternativen und mit slf4j eine Fassade, die die Details des genutzten Logging-Frameworks weitestgehend abstrahiert. Den Abschluss der Beschreibung von log4j2 bildet in Abschnitt 9.4.2 eine Zusammenstellung verschiedener Tipps und Tricks beim Einsatz von Logging.

Gründe für Logging-Frameworks

Ausgaben über `System.out` können zwar in Prototypen oder für erste Gehversuche oder Tests ausreichend sein, für die meisten Anwendungen sind sie es aus vielfältigen Gründen aber nicht. Der entscheidende Nachteil ist, dass die Ausgaben nicht persistent gespeichert werden, sondern lediglich über die Konsole »rauschen« und damit unwiederbringlich verschwinden. Zudem macht es einen unprofessionellen Eindruck, wenn ein Anwender alle möglichen (Fehler-)Meldungen auf der Konsole sieht. Daher ist es hilfreich, ein Logging-Framework zu nutzen: Dadurch kann man Informationen strukturiert, in verschiedenen Detaillierungsgraden und bei Bedarf sogar in unterschiedlichen Log-Dateien konfigurierbarer Größe und Anzahl sammeln. Nutzt man zur Protokollierung lediglich Ausgaben über `System.out`, so besitzt man diese Flexibilität nicht: Sollen mehr oder weniger Informationen ausgegeben werden oder will man diese komplett ein- oder ausschalten, so bedarf es Programmänderungen. Das erfordert dann auch eine Neukompilierung und eine neue Auslieferung der Software. Bei Logging-Frameworks ist das nicht erforderlich, denn diese erlauben eine Konfiguration sogar zur Laufzeit. Damit kann man temporär die Protokollierung zur Fehlersuche detaillierter gestalten. Außerdem lässt sich das Format der Log-Ausgaben beeinflussen, z. B. können Thread- und Datumsinformationen automatisch ergänzt werden.

9.4.1 Apache log4j2

Apache log4j2 ist ein recht populäres Logging-Framework. Es steht frei unter `https://logging.apache.org/log4j/2.x/download.html` zur Verfügung. Die benötigten Dateien müssen Sie nach dem Download in den `CLASSPATH` Ihres Projektverzeichnisses aufnehmen. Einfacher ist es, folgende Zeilen in die Build-Datei einzufügen, wodurch dann Gradle den Download und das Einbinden für Sie erledigt:

```
compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.13.3'
compile group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.13.3'
```

log4j-Grundbegriffe

Eine Instanz der Klasse `org.apache.logging.log4j.Logger` bildet die zentrale Stelle zur Ausgabe von Meldungen. Das folgende Listing zeigt ein Beispiel für den Einsatz der Klasse `Logger` und der Methoden `info(String)` und `error(String)`:

```
private static final Logger LOGGER = LogManager.getLogger(LoggingExample.class);

public static void main(final String[] args)
{
    //Log-Meldungen ausgeben
    LOGGER.info("Info-Meldung aus LoggingExample.");
    LOGGER.error("Error-Meldung aus LoggingExample.");
}
```

Listing 9.13 *Ausführbar als* **'LOGGINGEXAMPLE'**

Tipp: Korrekte und konsistente Wahl der Log-Level

Beim Logging kann über den Log-Level die Granularität und Wichtigkeit der auszugebenden Informationen bestimmt werden. Eine möglichst einheitliche Verwendung erleichtert eine spätere Auswertung von Meldungen. Es haben sich die in der folgenden Aufzählung genannten Vorschläge bewährt. Die Log-Level sind in aufsteigender Reihenfolge genannt – der gravierendste zuletzt:

- `TRACE` – Ausgaben auf dieser niedrigsten Ebene sind nur für ganz spezielle Anwendungsfälle für Entwickler zum Nachvollziehen von Programmabläufen interessant, die man durch Debugging eventuell nicht gut verfolgen kann.[a]
- `DEBUG` – Ausgaben auf dieser Ebene sind lediglich für Entwickler relevant und sollten im Normalbetrieb ausgeschaltet sein. Sie lassen sich bei Bedarf selektiv einschalten, um bestimmte Fehlerursachen aufspüren zu können.
- `INFO` – Auf dieser Ebene werden Informationen ausgegeben, die den Programmfluss sichtbar und verständlich machen. Dazu gehört etwa die Protokollierung von Methodenaufrufen an externen Schnittstellen.
- `WARN` – Wie der Name schon sagt, werden hier Warnungen ausgegeben: Es werden also Betriebssituationen gemeldet, die man zwar berücksichtigt hat, die aber von der erwarteten Situation abweichen, etwa dass eine Datei nicht geöffnet werden kann. Das System zeigt als Folge möglicherweise ein etwas anderes Verhalten als gewünscht.
- `ERROR` – Abgefangene Exceptions sollten auf dieser Ebene protokolliert werden, wenn der folgende Programmablauf dadurch wahrscheinlich gestört wird, etwa bei unerwarteten oder inkonsistenten Objektzuständen. Auch durch Wiederholung nicht zu beseitigende Fehler einer Netzwerk- oder Datenbankverbindung gehören dazu.
- `FATAL` – Der Log-Level `FATAL` bedeutet, dass ein schwerwiegender Fehler aufgetreten ist und die weitere Programmausführung nicht mehr sinnvoll möglich ist. In diesem Fall sollte das Programm beendet werden.

Zur Analyse von Programmfehlern ist es hilfreich, die obigen Hinweise zu beachten. Die Ausgabe von Informationen auf einem unangebrachten Log-Level kann eine Fehlersuche in eine falsche Richtung lenken. Schauen auch Dritte auf Log-Ausgaben, kann bei einer großen Anzahl von Ausgaben im `WARN`- oder `ERROR`-Level ein falscher Eindruck der Programmstabilität entstehen.

[a] Ich selbst nutze `TRACE` selten, sondern bevorzuge den Level `DEBUG`, da dies für meine Bedürfnisse nahezu immer ausreichend ist. Für Testsysteme kann man mit `TRACE` aber für spezielle Debug-Einsätze noch feiner protokollieren.

Neben den bereits verwendeten Methoden `info()` und `error()` gibt es für jeden Log-Level entsprechende Methoden wie `trace()`, `debug()`, `warn()` und `fatal()`.

Ein Logger versteckt den komplizierten Ausgabeprozess und bietet eine einheitliche Schnittstelle zur formatierten Ausgabe auf verschiedene Ausgabemedien mithilfe sogenannter ***Appender***.

Damit das Ganze funktioniert, benötigt log4j2 eine Konfiguration, am einfachsten in Form der Datei `log4j2.properties` im Verzeichnis `src/main/resources`:

```
name=PropertiesConfig
appenders = console
monitorInterval = 2

rootLogger.level = INFO
rootLogger.appenderRefs = Console
rootLogger.appenderRef.console.ref = Console

appender.console.type = Console
appender.console.name = Console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{yyy-MM-dd HH:mm:ss} %-5p [%t] %c: %m%n
```

Wir sehen die Definition eines Appenders für die Konsole. Über ein sogenanntes ***Layout*** kann die Formatierung der Log-Ausgabe festgelegt werden. Dazu gibt es verschiedene Spezialisierungen, hier kommt ein `PatternLayout` zum Einsatz. Dadurch werden beim Start des Programms folgende Ausgaben produziert:

```
2017-05-05 16:49:23 INFO  [main] ch09_applikationsbausteine.logging.
      LoggingExample: Info-Meldung aus LoggingExample.
2017-05-05 16:49:23 ERROR [main] ch09_applikationsbausteine.logging.
      LoggingExample: Error-Meldung aus LoggingExample.
```

Erweiterungen für Dateiausgaben Neben dem Konsole-Appender existieren weitere. In der Praxis wird man die Log-Ausgaben in Dateien persistieren wollen, etwa wie folgt – wobei der Name des Loggers mit dem Root-Package übereinstimmen muss:

```
loggers=file
logger.file.name=com.inden
logger.file.level = INFO
logger.file.appenderRefs = File
logger.file.appenderRef.file.ref = LOGFILE

appender.file.type = File
appender.file.name = LOGFILE
appender.file.fileName=logs/java-profi.log
appender.file.layout.type=PatternLayout
appender.file.layout.pattern=[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} -
       %msg%n
```

Statt des oberen Blocks hätte man den Appender auch dem Root-Logger wie folgt hinzufügen können:

```
rootLogger.level = INFO
rootLogger.appenderRefs = Console, File
rootLogger.appenderRef.console.ref = Console
rootLogger.appenderRef.file.ref = LOGFILE
```

Man sieht, dass log4j2 für den ersten Einsatz etwas schwierig zu konfigurieren und zu verstehen ist, deswegen verweise ich Sie für die weitere Konfiguration auf die Webseite.

Zudem ist Folgendes zu bedenken: Die Log-Dateien werden bei umfangreichen Log-Ausgaben schnell recht groß. Um eine Weiterverarbeitung und Auswertung zu vereinfachen, ist es sinnvoll, die Log-Dateien in handliche »Häppchen« aufzuteilen. Mögliche Kriterien sind die Dateigröße oder das Datum der Log-Ausgaben. Dabei hilft der Typ `RollingFile`. Damit können bei Bedarf fortlaufend neue Log-Dateien erzeugt werden. Das heißt, wenn die Größe der momentanen Log-Datei einen gewissen Wert übersteigt, dann wird automatisch eine neue Log-Datei erzeugt und die bisherige umbenannt. Die Anzahl der so entstehenden Backup-Dateien und die maximal gewünschte Dateigröße sind konfigurierbar.

Konfiguration der Log-Ausgaben für Packages und Klassen Neben der Konfiguration der Logger, Appender und Layouts können spezielle Log-Level für bestimmte Packages oder Klassen in der Konfigurationsdatei `log4j2.properties` nach folgendem Muster als Schlüssel-Wert-Paare definiert werden:

```
logger.<package>.<klasse>=<Log-Level>
```

Durch diese Einträge kann die Ausgabe von Log-Meldungen bis auf Klassenebene feingranular in ihrem Log-Level eingestellt werden. Dabei kann statt eines Packages oder eines konkreten Klassennamens auch die Angabe einer logischen Kategorie, etwa `AudioIn` oder `AudioOut`, erfolgen. Diese werden im folgenden Abschnitt benutzt, um spezielle, klassenübergreifende Log-Ausgaben zu steuern.

```
# Hier folgt die Konfiguration für die Packages und Klassen
# Hier ändern für LogReadConfigExample-Beispiel
logger.LogReadConfigExample.name=ch09_applikationsbausteine.logging.
      LogReadConfigExample
logger.LogReadConfigExample.level=INFO

# bewusster Schreibfehler
logger.ProvideStackTrace.name=applikationsbausteine.logging.ProvideStackTrace
logger.ProvideStackTrace.level=WARN

# Ausgaben Package-basiert
logger.ch22_optimierungen.name=ch22_optimierungen
logger.ch22_optimierungen.level=INFO

# Logische Log-Kategorien
logger.AudioIn.name=AudioIn
logger.AudioIn.level=INFO
logger.AudioOut.name=AudioOut
logger.AudioOut.level=INFO
```

Praktischerweise registriert log4j2 automatisch Änderungen an der Datei und konfiguriert das Log-System um, wenn man den Parameter `monitorInterval` gesetzt hat. Das können Sie nachvollziehen, indem Sie den Log-Level in der Konfigurationsdatei `build/resources/main/log4j2.properties` ändern, während Sie folgendes Programm ausführen:

```
private static final Logger LOGGER =
                            LogManager.getLogger(LogReadConfigExample.class);

public static void main(final String[] args) throws InterruptedException
{
    LOGGER.info("LogReadConfigExample started");

    while (!Thread.interrupted())
    {
        LOGGER.trace("TRACE");
        LOGGER.debug("DEBUG");
        LOGGER.info("INFO");
        LOGGER.warn("WARN");
        LOGGER.error("ERROR");
        LOGGER.fatal("FATAL");

        TimeUnit.SECONDS.sleep(2);
    }
}
```

Listing 9.14 *Ausführbar als* '**LOGREADCONFIGEXAMPLE**'

Achtung: Fallstricke bei der Angabe von Klassen- und Package-Namen

Ganz wichtig ist es, den voll qualifizierten Namen der Klasse oder des Packages korrekt anzugeben. Wird hierbei ein Tippfehler gemacht, so wirken sich die Konfigurationen nicht aus. In der Praxis ist das ein häufiges Problem, wenn nach Refactorings zum Teil vergessen wird, den korrespondierenden Pfad oder Klassennamen in der Konfigurationsdatei `log4j2.properties` anzupassen. Im obigen Beispiel fehlt im unteren Fall das Präfix `ch09_` vor `applikationsbausteine`.

9.4.2 Tipps und Tricks zum Einsatz von Logging mit log4j2

Im Folgenden stelle ich Ihnen ein paar Tipps und Tricks rund um das Logging vor.

Abfrage der Log-Level

Mitunter sieht man Sourcecode, der Abfragen auf den Log-Level etwa wie folgt nutzt:

```
if (LOGGER.isDebugEnabled())
{
    LOGGER.debug("Logging in user " + user.getName() +
                 " with birthday " + user.getBirthday());
}
```

Dies geschieht oftmals, weil man als Entwickler eine unnötige Stringkonkatenation vermeiden möchte: Durch die vorherige Abfrage des Log-Levels, etwa `isDebugEnabled()`, kann man die Aufbereitung aufwendiger Log-Ausgaben nur bei Bedarf durchführen und auch nur dann in die Log-Datei schreiben.

Allerdings wird der Sourcecode durch diese Abfragen recht unübersichtlich. Daher sollte man die Abfrage der Log-Level lediglich vor der Aufbereitung komplexer Log-

Ausgaben und nicht einfach immer durchführen. Für einzeilige Ausgaben ist folgende Alternative besser, weil die Parameter nur in den String eingefügt werden, wenn der Log-Level passend ist:

```
LOGGER.debug("Logging in user {} with birthday {}",
             user.getName(), user.getBirthday());
```

Allerdings erfolgen dabei immer noch die Aufrufe der Methoden zum Ermitteln der Übergabewerte. Sofern dies eine aufwendige Operation ist, bietet es sich seit Java 8 an, einen Lambda nur für eine bei Bedarf ausgeführte Berechnung wie folgt zu nutzen:

```
LOGGER.debug("Result of some expensive operation is {}",
             () -> someLongRunningOperation());
```

Logging von Strings

Bei Ausgaben von textuellen Werten ist es sinnvoll, diese in einfache Anführungszeichen oder alternativ in spitze Klammern einzuschließen:

```
LOGGER.debug("Name = '" + name + "', Age = " + age);
LOGGER.debug("Name = <" + name + ">, Age = " + age);
```

Dies macht Leerzeichen vor und nach dem eigentlichen Text sichtbar und hilft dabei, tückische Fehler, bei denen Leerzeichen oder unsichtbare Sonderzeichen zu Problemen führen, leichter erkennen zu können (vgl. Abschnitt 5.1.1): Wird etwa mit einem solchen Eingabetext als Schlüssel in einer Map gesucht, so wird man dafür nichts finden.

Logging von Exceptions

Beim Auftreten von Exceptions ist es zur späteren Fehleranalyse meistens hilfreich, einen Stacktrace in die Log-Datei zu schreiben. Leider sieht man häufiger den Fehler, bei einer Log-Ausgabe wichtige Informationen abzuschneiden, beispielsweise dadurch, dass eine Stringrepräsentation einer Exception folgendermaßen erzeugt wird:

```
private static final Logger LOGGER =
                           LogManager.getLogger("ExceptionLoggingExample");

public static void main(final String[] args)
{
    try
    {
        methodThrowingException();
    }
    catch (final IOException e)
    {
        // SCHLECHT: nur String-Info ohne Stacktrace!
        LOGGER.error("An I/O error occurred! " + e);
        LOGGER.error("An I/O error occurred! " + e.getMessage());
    }
}
```

Listing 9.15 *Ausführbar als* **'EXCEPTIONLOGGINGEXAMPLE'**

Führt man die Beispielapplikation aus, so erhält man in etwa folgende Ausgaben:

```
2017-05-05 17:29:15 ERROR [main] ExceptionLoggingExample: An I/O error occurred!
       java.io.IOException: Text
2017-05-05 17:29:15 ERROR [main] ExceptionLoggingExample: An I/O error occurred!
       Text
```

Man erkennt, dass der Stacktrace und damit der zur Fehleranalyse oftmals wichtigste Teil der Exception nicht ausgegeben wird. Die Ausgabemethoden von log4j2 besitzen jeweils eine überladene Variante, die durch die folgende kommaseparierte Angabe einer Exception zusätzlich den Stacktrace protokolliert:

```
private static final Logger LOGGER =
                           LogManager.getLogger("ExceptionLoggingImproved");

public static void main(final String[] args)
{
    try
    {
        methodThrowingException();
    }
    catch (final IOException e)
    {
        // String-Info MIT Stacktrace
        LOGGER.error("An I/O error occurred!", e);
    }
}
```

Listing 9.16 *Ausführbar als* '**EXCEPTIONLOGGINGIMPROVED**'

Eine Ausgabe mit Stacktrace erleichtert ein Nachvollziehen des Wegs durch das Programm bis zu der Stelle, die den Fehler verursacht hat:

```
2017-05-05 17:29:52 ERROR [main] ExceptionLoggingImproved: An I/O error occurred
      !
java.io.IOException: Text
    at ch08_applikationsbausteine.logging.ExceptionLoggingImproved.second(
          ExceptionLoggingImproved.java:46) ~[bin/:?]
    at ch08_applikationsbausteine.logging.ExceptionLoggingImproved.first(
          ExceptionLoggingImproved.java:41) ~[bin/:?]
    at ch08_applikationsbausteine.logging.ExceptionLoggingImproved.
          methodThrowingException(ExceptionLoggingImproved.java:35) ~[bin/:?]
    at ch08_applikationsbausteine.logging.ExceptionLoggingImproved.main(
          ExceptionLoggingImproved.java:23) [bin/:?]
```

Stacktrace ausgeben

Manchmal ist eine Fehlersuche schwierig, etwa weil man in einigen Situationen keinen Debugger verwenden kann oder Fehler extrem selten auftreten und sich dadurch nicht nachstellen lassen. Zur Fehleranalyse ist es wünschenswert, an gewissen Stellen im Programm den aktuellen Stacktrace ausgeben zu können, ohne das Programmverhalten zu verändern. Eine Ausgabe in die Log-Datei erreicht man durch folgenden Trick: Es wird eine Exception beliebigen Typs (bevorzugt aber `IllegalStateException`) lediglich erzeugt, aber ***nicht*** geworfen, sondern stattdessen direkt an die Log-Ausgabe-Methode

übergeben. Als Folge erhält man durch den mitgelieferten Stacktrace einen Einblick in die internen Programmabläufe.

```
private static void provideStackTrace()
{
    LOGGER.info("Stacktrace: ", new IllegalStateException("Stacktrace!"));
}
```

Verwende logische Log-Kategorien

Mit Logging-Frameworks kann man nicht nur auf Klassenebene loggen, sondern es lassen sich dafür auch eigene Log-Kategorien verwenden. Dadurch kann man eine logische, klassenübergreifende Sicht definieren und eine semantische Trennung erreichen. Es werden dazu mehrere Logger pro Datei instanziiert. Beispielsweise kann ein Logger klassenbasierte Ausgaben vornehmen, während andere Logger auf logischer Ebene arbeiten. Nehmen wir an, es wären mehrere Klassen an einer Funkkommunikation beteiligt. Man definiert dann die Kategorien `AudioIn` und `AudioOut`, um alle Aktionen im Funkeingang und -ausgang klassenunabhängig und logisch gruppiert auszugeben:

```
public final class LogCategoryExample
{
    private static final Logger LOGGER =
                                LogManager.getLogger(LogCategoryExample.class);

    private static final Logger audioInLog  = LogManager.getLogger("AudioIn");
    private static final Logger audioOutLog = LogManager.getLogger("AudioOut");

    LogCategoryExample()
    {
        LOGGER.info("LogCategoryExample created");
    }

    private void send(final byte[] msg)
    {
        LOGGER.debug("send()");
        audioOutLog.info("Sending " + ByteUtils.byteArrayToString(msg));
    }

    private byte[] receive(final InputStream inStream)
    {
        LOGGER.debug("receive()");

        final byte[] msg = getMsgFromStream(inStream);
        audioInLog.info("Receiving " + ByteUtils.byteArrayToString(msg));
        return msg;
    }

    public static void main(final String[] args)
    {
        final LogCategoryExample logExample = new LogCategoryExample();
        logExample.send("Hello".getBytes());
    }
    // ...
```

Listing 9.17 *Ausführbar als* '**LOGCATEGORYEXAMPLE**'

Die jeweiligen Log-Kategorien werden mit in die Log-Ausgabe geschrieben. Wenn alle Log-Einstellungen auf `INFO` stehen, kommt es beispielsweise zu folgender Ausgabe:

```
2017-05-05 17:21:41 INFO  [main] ch08_applikationsbausteine.LogCategoryExample:
      LogCategoryExample created
2017-05-05 17:21:41 INFO  [main] AudioOut: Sending 'Hello' = '[72 101 108 108
      111 ]'
```

Logging der Aufrufe an Schnittstellen zu anderen Komponenten

Bei einer Zusammenarbeit mit anderen Komponenten oder Systemen kann man durch die Ausgabe der aufgerufenen öffentlichen Methoden mitsamt ihrer Parameter den Programmablauf nachvollziehbar gestalten. Auf diese Weise lassen sich recht schnell Probleme durch inkorrekte Abläufe oder falsch übergebene Daten erkennen.

Automatische Auswertungsmöglichkeiten

Die in den Log-Dateien protokollierten Meldungen lassen sich mit einem Texeditor anschauen und mit dessen Suchfunktion analysieren. Bei umfangreichen und feingranularen Log-Ausgaben kann dieses Vorgehen zeitaufwendig und unbequem sein.

Weiter gehende Möglichkeiten der Auswertung bieten Batch-Skripte. Damit kann man Log-Dateien analysieren und daraus bestimmte Teile extrahieren, etwa auf Basis eines speziellen Log-Levels. Manchmal lässt sich dann mit etwas Erfahrung bereits anhand der Dateigröße der Extrakte erkennen, ob es sich um eine normale Situation handelt oder ob man eine detailliertere Analyse vornehmen muss. Hilfreicher als eine Analyse nach Log-Level ist es in der Regel, nach Vorkommen spezieller Log-Ausgaben in einer Log-Datei zu suchen, etwa Exceptions oder Warnmeldungen. Das `grep`-Tool[6] leistet dabei gute Dienste. Allerdings lassen sich damit nur einzeilige Meldungen besonders gut auswerten – mehrzeilige Stacktraces bereiten dagegen Probleme.

```
grep "ERROR" SampleApplication.log* > All_Errors.txt
grep "WARN"  SampleApplication.log* > All_Warns.txt
```

Tipp: Auswertung von Log-Ausgaben

Man sollte Log-Ausgaben so gestalten, dass sie leicht maschinenlesbar sind. Daraus lassen sich dann Informationen zum Programmablauf gewinnen.

[6] Für Windows findet man unter `http://gnuwin32.sourceforge.net/packages/grep.htm` eine freie Version.

9.5 Konfigurationsparameter und -dateien

Viele Anwendungen bieten Möglichkeiten zur Konfiguration. Im Folgenden zeige ich dazu verschiedene Varianten. Abschnitt 9.5.1 beschreibt, wie man Kommandozeilenparameter auswerten kann. Eine Konfiguration mithilfe der Klasse `java.util.Properties` wird in Abschnitt 9.5.2 vorgestellt. Konfigurationswerte können auch kommasepariert (CSV) vorliegen. In Abschnitt 9.5.3 werden einige Ideen zum Einlesen derart gespeicherter Werte vorgestellt.

9.5.1 Einlesen von Kommandozeilenparametern

Eine Möglichkeit zur Konfiguration einer Java-Applikation besteht darin, dieser bei deren Start eine beliebige Anzahl an Parametern zu übergeben, etwa wie folgt:

```
java MyApp Parameter1 Parameter2 "Text mit Leerzeichen"
```

Dabei ist zu beachten, dass Leerzeichen die Kommandozeilenargumente trennen: ***Wenn ein String mit Leerzeichen übergeben werden soll, so muss dies in Anführungszeichen geschehen.*** Im Beispiel würde ansonsten der Parameterwert `Text mit Leerzeichen` als Angabe von drei Parametern ausgewertet.

Als Beispiel zur Auswertung von Kommandozeilenargumenten betrachten wir eine Parameterübergabe, um damit sowohl die Größe des Applikationsfensters zu ändern als auch optional spezielle Debug-Informationen zu aktivieren.

Notationsformen für Parameter

Folgende zwei Varianten bei der Parameterübergabe sind gebräuchlich:

1. **Fixe Reihenfolge** – Die Bedeutung eines übergebenen Werts wird analog zu einem Methodenaufruf anhand seiner Position ermittelt. Der Aufruf könnte etwa

   ```
   java MyApp 500 300 debug
   ```

   lauten. ***Diese starre Zuordnung ist wenig flexibel, birgt die Gefahr der Verwechslung und ist außerdem problematisch bei optionalen Parametern***. Zudem kann nur ein optionaler Parameter am Ende der Parameterliste angegeben werden.
2. **Benannte Parameter** – Jeder Parameter wird durch einen Namen oder ein entsprechendes Kürzel eingeleitet und beschreibt so seine Intention, etwa wie folgt:

   ```
   java MyApp width 500 height 300 debug
   ```

   Diese Variante erlaubt eine flexible Reihenfolge und eine beliebige Zahl optionaler Parameter, jedoch zulasten einer komplexeren Auswertung als bei Variante 1.

Gebräuchlicher ist die zweite Variante, wobei dem Parameternamen meist ein Minuszeichen vorangestellt wird.

Die Übergabe von Werten kann folgendermaßen geschehen:

- **-<name> <Wert>**, etwa `-width 200`
- **-<name>=<Wert>**, etwa `-width=200`

Die erste Form besteht aus zwei Kommandozeilenparametern, die eine Einheit (Name und Wert) bilden. Die zweite Form entspricht einem Kommandozeilenparameter, was ein zusätzliches Parsing zur Trennung von Parameter und Wert erfordert. Boolesche Flags werden durch Parameter ohne Wertangabe, etwa `-debug`, ausgedrückt. Zusätzliche Komplexität entsteht, wenn man auch Kurzformen von Parameternamen unterstützen möchte, etwa `-h` statt `-help` oder `-d` statt `-debug`.

Auswertung von Parametern

Kommandozeilenargumente werden als `String[]` an die `main()`-Methode übergeben. Zur Bereitstellung von Werten definieren wir das `String[]` namens `sampleArgs`. Zur Auswertung werden die Werte in einer Schleife mit den erwarteten Parameterkürzeln bzw. Parameternamen verglichen. Zudem wird für jeden Parameter eine Variable definiert und mit einem Defaultwert belegt:

```
public static void main(final String[] args)
{
    // Defaultwerte, wenn Wert nicht in der Kommandozeile übergeben
    int width = 700;
    int height = 100;
    boolean debug = false;
    boolean showHelp = false;

    // Test mit festdefinierten Werten
    final String[] sampleArgs = { "-h", "-w=500", "-height=550" };
    for (final String cmdArg : sampleArgs)
    {
        if (cmdArg.startsWith("-d"))            // check debug
            debug = true;
        else if (cmdArg.startsWith("-h"))       // check help
            showHelp = true;
        else if (cmdArg.startsWith("-w="))      // check width
            width = Integer.parseInt(cmdArg.substring(3));
        else if (cmdArg.startsWith("-width="))
            width = Integer.parseInt(cmdArg.substring(7));
        // check height !!! UNREACHABLE !!!
        else if (cmdArg.startsWith("-h="))
            height = Integer.parseInt(cmdArg.substring(3));
        else if (cmdArg.startsWith("-height="))
            height = Integer.parseInt(cmdArg.substring(8));
    }
    // ...
```

Listing 9.18 *Ausführbar als* '**ARGSPARSINGEXAMPLENAIV**'

Bereits für wenige Parameter ist diese Art der Auswertung mit `if-else`-Gebilden unübersichtlich. Zudem enthält das Beispiel bewusst einen Flüchtigkeitsfehler: Parameter mit gleichem Startbuchstaben werden nicht unterschieden. Hier gilt das für die Abfrage `startsWith("-h")` zur Aktivierung der Hilfe. Dadurch werden Angaben zur Höhe

des Fensters mit `-h=` oder `-height=` niemals ausgewertet. Ein Start des Programms ARGSPARSINGEXAMPLENAIV zeigt das: Die Höhe des Fensters entspricht dem vordefinierten Wert von 100 Pixel und nicht dem als String übergebenen Wert `-height=550` (vgl. Abbildung 9-1).

Abbildung 9-1 *Applikationsfenster, Größe basierend auf Kommandozeilenparametern*

Ein erster Ansatz zur Korrektur und damit zur Auswertung aller Parameter trotz gleicher Startzeichenfolge ist, Alternativen jeweils mit `if` statt mit `else if` auszudrücken, sodass immer alle `if`-Zweige betrachtet werden. Dabei kommt es jedoch zu anderen Fallstricken: Eine Angabe von `-h=100` setzt nicht nur den Wert für die Höhe. Es wird zusätzlich unerwartet auch der Wert für die Aktivierung der Hilfe auf `true` gesetzt. Außerdem können weitere Probleme durch die Auswertung mit `startsWith(String)` auftreten. Beispielsweise wird die Angabe von `-hello` aber wie der Aufruf von `-h` interpretiert. Mehr noch: Das Auslesen von Werten ist durch die Abstützung auf sogenannte Magic Strings und Magic Numbers fragil.[7] Wird der Name eines Parameters umbenannt, besteht die Gefahr, dass übersehen wird, die korrespondierende Längenangabe im Aufruf von `substring(int)` anzupassen.

Verbesserungen Ein möglicher Lösungsansatz besteht darin, Parameternamen und Kurzformen als Konstanten zu definieren. Dies lässt sich recht elegant als `enum`-Aufzählung `AppParameter` wie folgt realisieren:

```
enum AppParameter
{
    DEBUG_SHORT("d"),
    HELP_SHORT("?"),
    WIDTH_SHORT("w="), WIDTH("width="),
    HEIGHT_SHORT("h="), HEIGHT("height=");

    final String paramName;
    final int    valueOffset;

    AppParameter(final String name)
    {
        this.paramName = "-" + name;
        this.valueOffset = paramName.length();
    }
}
```

[7] Das sind String- bzw. Zahlenliterale im Sourcecode, etwa `-h=` oder der Wert `7`. Die Magie dieser Angaben liegt darin, dass die Intention normalerweise verborgen bleibt. Häufig ist es sinnvoller und besser verständlich, Konstanten zu definieren. Ausgenommen davon sind oftmals nur die Werte 0 und -1 als Startwert bzw. zur Berechnung der Abbruchbedingung in Schleifen.

Diese Art der Realisierung besitzt den Vorteil, dass an einer zentralen Stelle übersichtlich alle erlaubten Parameter definiert werden. Zudem werden nur einmalig die Zeichen '–' (Präfix für alle Parameter) und '=' (signalisiert Parameter mit nachfolgendem Wert) mit dem Parameternamen verbunden. Das hält den Applikationscode frei von diesen Details. Weiterhin lässt sich die Länge des Parameterkürzels inklusive der Zeichen '–' sowie '=' direkt bei der Konstruktion berechnen, sodass selbst bei Änderungen in den Parametern keine Modifikationen in einsetzenden Applikationen nötig werden.

Um die Auswertung der Parameter zu erleichtern, nachdem sie aus dem `String[]` ausgelesen worden sind, definieren wir folgende drei Hilfsmethoden:

```
private static boolean hasBooleanParam(final String cmdArg,
                                       final AppParameter parameter)
{
    return cmdArg.equals(parameter.paramName);
}

private static boolean hasValuedParam(final String cmdArg,
                                      final AppParameter parameter)
{
    return cmdArg.startsWith(parameter.paramName) &&
           cmdArg.length() > parameter.valueOffset;
}

private static int extractInt(final String cmdArg, final AppParameter parameter)
{
    return Integer.parseInt(cmdArg.substring(parameter.valueOffset));
}
```

Nutzen wir die `enum`-Aufzählung und die obigen Methoden bei der Auswertung der Parameter, so ergibt sich folgender Sourcecode:

```
for (final String cmdArg : sampleArgs)
{
    if (hasBooleanParam(cmdArg, AppParameter.DEBUG_SHORT))
        debug = true;
    if (hasBooleanParam(cmdArg, AppParameter.HELP_SHORT))
        showHelp = true;

    if (hasValuedParam(cmdArg, AppParameter.WIDTH_SHORT))
        width = extractInt(cmdArg, AppParameter.WIDTH_SHORT);
    if (hasValuedParam(cmdArg, AppParameter.WIDTH))
        width = extractInt(cmdArg, AppParameter.WIDTH);

    if (hasValuedParam(cmdArg, AppParameter.HEIGHT_SHORT))
        height = extractInt(cmdArg, AppParameter.HEIGHT_SHORT);
    if (hasValuedParam(cmdArg, AppParameter.HEIGHT))
        height = extractInt(cmdArg, AppParameter.HEIGHT);
}
```

Listing 9.19 *Ausführbar als* **'ARGSPARSINGEXAMPLEWITHENUM'**

Der Sourcecode ist nun verständlicher und deutlich besser erweiterbar. Dies liegt zum einen am Einsatz des Aufzählungstyps und zum anderen an den drei eingeführten Hilfsmethoden, die zur Lesbarkeit beitragen. Wir erkennen, dass sich beispielsweise die zuvor komplizierte und fehleranfällige Abfrage mit

```
else if (cmdArg.startsWith("-width="))
    width = Integer.parseInt(cmdArg.substring(7));
```

durch den Einsatz des Aufzählungstyps und der Hilfsmethoden lesbar, verständlich und typsicher wie folgt schreiben lässt:

```
else if (hasValuedParam(cmdArg, AppParameter.WIDTH))
    width = extractInt(cmdArg, AppParameter.WIDTH);
```

Außerdem werden durch gleiche Startbuchstaben für verschiedene Parameter verursachte Probleme unwahrscheinlich. Zum einen sind alle Parameter innerhalb eines Aufzählungstyps definiert und so fallen gleiche Parameternamen schneller auf als in verschachtelten `if`-Anweisungen. Zum anderen erfolgt in den beiden Methoden `hasBooleanParam(String, AppParameter)` und `hasValuedParam(String, AppParameter)` die Abfrage auf Übereinstimmung unterschiedlich: Boolesche Parameter werden auf exakte textuelle Gleichheit geprüft, für Werteparameter wird weiterhin `startsWith(String)` genutzt.

Verbliebene Schwachstellen und Lösungen Trotz der erzielten Verbesserungen erfordert die Unterscheidung und Auswertung von Kurz- oder Langform eines Parameters mindestens zwei `if`-Anweisungen und ist dadurch noch nicht so gut lesbar und erweiterbar, wie es wünschenswert wäre.

Darüber hinaus fällt beim genaueren Betrachten des Sourcecodes auf, dass an verschiedenen Stellen Zugriffe auf Variablen erfolgen. Steigt die Anzahl der auszuwertenden Parameter, so kann man für mehr Klarheit sorgen, wenn man Variablen gemäß dem Muster VALUE OBJECT (vgl. Abschnitt 3.4.5) zusammenfasst. Im weiteren Verlauf werden wir daher folgendes Parameter Value Object `AppParameterVO` verwenden. Durch dieses erfolgt eine zentrale Definition aller durch Kommandozeilenparameter beeinflussten Variablen. Bei der Implementierung wird bewusst auf `get()`- und `set()`-Methoden verzichtet, da es sich lediglich um einen einfachen Behälter für Werte handelt, der zudem nur innerhalb des Packages selbst verwendet werden soll. Daher werden die Attribute zur Einschränkung von Zugriffsmöglichkeiten Package-private definiert:

```
class AppParameterVO
{
    // Defaultwerte, wenn der Wert nicht in der Kommandozeile übergeben wird
    int     width    = 700;
    int     height   = 100;
    boolean debug    = false;
    boolean showHelp = false;

    @Override
    public String toString()
    {
        return "AppParameterVO [debug=" + debug + ", height=" + height + ", " +
                               "showHelp=" + showHelp + ", width=" + width + "]";
    }
}
```

Dies Auswertung von Hand lässt sich nur noch schwierig verbessern. Es ist an der Zeit, eine Bibliothek einzusetzen, um die Auswertung zu erleichtern.

Apache Commons CLI

Die Bibliothek Apache Commons CLI (`http://commons.apache.org/cli/`) ist nützlich, wenn die Anzahl zu verarbeitender Parameter zunimmt oder die Auswertung komplexer wird. Zum Einbinden fügt man Folgendes in der Build-Datei hinzu:

```
compile 'commons-cli:commons-cli:1.4'
```

Definition von Parametern Parameter werden als sogenannte Optionen repräsentiert. Die vom Programm unterstützten Parameter werden dazu in einem Container vom Typ `Options` gesammelt. Einzelne Parameter (`Option`-Objekte) werden durch Aufruf der Methode `addOption(String opt, String longOpt, boolean hasArg, String description)` dem `Options`-Container hinzugefügt.[8]

Wollten wir lediglich einen Parameter zur Anzeige von Hilfetexten erlauben, so kann man einen Container `allowedOptions` vom Typ `Options` erzeugen und die Hilfeoption wie folgt hinzufügen:

```
final Options allowedOptions = new Options();
allowedOptions.addOption("?", "help", false, "show help");
```

Die CLI-Bibliothek erwartet bei der Definition – im Gegensatz zur späteren Auswertung – für die Kurz- und Langnamen kein Minuszeichen. Über den dritten Wert, hier `false`, steuert man, ob ein Parameter Argumente hat. Zusätzlich kann eine Parameterbeschreibung übergeben werden, die beim Aufbau einer Hilfeseite nützlich ist.

Vereinfachte Definition von Parametern Statt jeden Parameter einzeln durch einen Aufruf der Methode `addOption()` hinzuzufügen, wäre eine geschicktere Realisierung wünschenswert. Das ist relativ einfach durch kleinere Änderungen möglich. Dazu wird die bereits existierende `enum`-Aufzählung `AppParameter` dahingehend erweitert, dass sie alle Informationen bereitstellen kann, die die Methode `addOption(String, String, boolean, String)` der Klasse `Options` als Eingabe benötigt. Dazu definieren wir die folgende `enum`-Aufzählung `AppParameterCLI`, die alle zur erforderlichen Informationen vorhält:

```
public enum AppParameterCLI
{
    WIDTH("w", "width", true, "set the width"),
    HEIGHT("h", "height", true, "set the height"),
    HELP("?", "help", false, "show help"),
    DEBUG("d", "debug", false, "activate debug mode");
```

[8] Man beachte hier den feinen textuellen, aber großen semantischen Unterschied zwischen `Options` ≡ Container und `Option` ≡ Parameter.

```
    final String  shortname;
    final String  name;
    final boolean hasArgs;
    final String  helptext;

    AppParameterCLI(final String shortname, final String name,
                    final boolean hasArgs, final String helptext)
    {
        this.shortname = shortname;
        this.name = name;
        this.hasArgs = hasArgs;
        this.helptext = helptext;
    }
}
```

Mit der Methode `values()` der `enum`-Aufzählung lässt sich eine elegante Definition aller zu unterstützenden Parameter wie folgt implementieren:

```
public final class CommandLineParsingExample
{
    public static void main(final String[] args)
    {
        final Options allowedOptions = new Options();

        // Alle Options basierend auf enum AppParameterCLI hinzufügen
        for (final AppParameterCLI parameter : AppParameterCLI.values())
        {
            allowedOptions.addOption(parameter.shortname, parameter.name,
                                     parameter.hasArgs, parameter.helptext);
        }
        //...
```

Listing 9.20 *Ausführbar als* **'COMMANDLINEPARSINGEXAMPLE'**

Auswertung von Parametern Nachdem alle gewünschten Parameter so definiert wurden, können beim Start eines Programms Eingabewerte aus der Kommandozeile geparst werden. Dazu wird zunächst ein `CommandLineParser`-Objekt erzeugt. Mit dessen Methode `parse(String[])` erhält man ein `CommandLine`-Objekt. Dieses kann man über die Methode `hasOption()` befragen, ob eine gewünschte Option aktiviert wurde, d. h. als Parameter angegeben ist. Im folgenden Listing, das das obige Listing COMMANDLINEPARSINGEXAMPLE fortsetzt, ist dies gezeigt:

```
        //...
        // Test mit festdefinierten Werten
        final String[] sampleArgs = new String[] { "-d", "-?", "-w", "444",
                                                   "--height", "99" };

        final AppParameterVO parameters = new AppParameterVO();

        try
        {
            // Kommandozeilenparameter parsen
            final CommandLineParser parser = new PosixParser();
            final CommandLine line = parser.parse(allowedOptions, sampleArgs);

            // Prüfen der Optionen
            parameters.debug = hasOption(line, AppParameterCLI.DEBUG);
            parameters.showHelp = hasOption(line, AppParameterCLI.HELP);
```

```
            if (hasOption(line, AppParameterCLI.WIDTH))
                parameters.width = extractInt(line, AppParameterCLI.WIDTH);

            if (hasOption(line, AppParameterCLI.HEIGHT))
                parameters.height = extractInt(line, AppParameterCLI.HEIGHT);

            final JFrame appFrame = new AppFrame("CommandLineParsingExample",
                                                 parameters);
            appFrame.setVisible(true);
        }
        catch (final ParseException exp)
        {
            printHelp(allowedOptions);
        }
    }

    private static boolean hasOption(final CommandLine line,
                                     final AppParameterCLI parameter)
    {
        return line.hasOption(parameter.name);
    }

    private static int extractInt(final CommandLine line,
                                  final AppParameterCLI parameter)
    {
        return Integer.parseInt(line.getOptionValue(parameter.name));
    }

    private static void printHelp(final Options allowedOptions)
    {
        final HelpFormatter formatter = new HelpFormatter();
        formatter.printHelp("CommandLineParsingExample", allowedOptions);
    }
    // ...
```

Wie man leicht sieht, ist diese Art der Auswertung deutlich übersichtlicher und birgt wesentlich weniger Potenzial für Fehler. Zudem ist eine Fehlerbehandlung in das Parsing der CLI-Bibliothek integriert: Wurde versehentlich eine unbekannte Option übergeben, so wird eine `org.apache.commons.cli.ParseException` geworfen. Eine angemessene Reaktion kann beispielsweise die Ausgabe aller unterstützten Parameter sein. Wenn man dazu die Klasse `HelpFormatter` und deren Methode `printHelp()` nutzt, dann wird Folgendes aufbereitet:

```
usage: CommandLineParsingExample
 -?,--help          show help
 -d,--debug         activate debug mode
 -h,--height <arg>  set the height
 -w,--width <arg>   set the width
```

Dies ist insofern praktisch, als eine Aufbereitung einer Hilfeseite ansonsten eine aufwendige und fehleranfällige Aufgabe ist.

9.5.2 Verarbeitung von Properties

Neben der Übergabe von Kommandozeilenparametern lassen sich Einstellungen auch aus Property-Dateien oder aus Umgebungsvariablen auslesen. In beiden Fällen liegen die Informationen in Form von Schlüssel-Wert-Paaren vor, wobei sowohl Schlüssel als auch Wert vom Typ `String` sind. Wenn solche Werte aus einer Konfigurationsdatei ermittelt werden sollen, bietet sich der Einsatz der Klasse `java.util.Properties` an. Diese erledigt das Einlesen der Datei und das Parsen der Wertepaare und bietet anschließend Zugriff über die Methode `getProperty(String)`.

Erste Realisierung mit der Klasse `Properties`

Betrachten wir die Verarbeitung von Konfigurationsdaten mit der Klasse `Properties` konkret am Beispiel der folgenden Property-Datei `AppConfig.properties` aus dem Konfigurationsordner `src/main/resources` unseres Projektverzeichnisses. Nehmen wir an, diese enthielte den Pfad auf ein extern zu startendes Programm und eine Parametrierung eines Datenbankzugriffs:

```
pdf.viewer=AcroRd32.exe

db.user=myuser
db.password=mypwd
```

Zum Einlesen von Konfigurationsdaten aus einem `InputStream` bietet die Klasse `Properties` die Methode `load(InputStream)`. Danach kann über die Methode `getProperty(String)` auf die Konfigurationswerte zugegriffen werden:

```
public static final void main(final String[] args) throws IOException
{
    final Properties props = new Properties();

    try (final InputStream in = new BufferedInputStream(
         new FileInputStream("src/main/resources/AppConfig.properties")))
    {
        props.load(in);

        final String appPdfViewer = properties.getProperty("pdf.viewer");
        System.out.println("PdfViewer = '" + appPdfViewer + "'");

        final String dbUser = properties.getProperty("db.user");
        final String dbPwd = properties.getProperty("db.password");
        System.out.println("DB-USER/PWD = '" + dbUser + "'/'" + dbPwd + "'");
    }
}
```

Listing 9.21 *Ausführbar als* '**APPPROPERTIESFIRST**'

Für dieses Beispiel kommt es erwartungsgemäß zu folgender Ausgabe:

```
PdfViewer = 'AcroRd32.exe'
DB-USER/PWD = 'myuser'/'mypwd'
```

Falls eine Konfigurationsdatei tatsächlich nur einige wenige Schlüssel-Wert-Paare beherbergt, dann ist die gewählte Umsetzung in Ordnung, wenn auch nicht elegant. Man erkennt bereits jetzt die Fragilität: Durch hartcodierte Strings als Schlüsselwerte ist es möglich, eine Abfrage mit jedem beliebigen und nicht nur den definierten Namen für Konfigurationsparameter durchzuführen. Jeder Tippfehler in der Konfigurationsdatei oder bei einer Abfrage verursacht unerwartete Resultate.

Betrachten wir aber noch ein subtileres Problem, das auftreten kann, wenn man ein wenig unvorsichtig programmiert: Die Klasse `Properties` ist von der Klasse `Hashtable<Object, Object>` abgeleitet (Implementierungsvererbung) und bietet dadurch sämtliche darin definierten Methoden. Neben der typsicheren Speicherung von Schlüsseln und Werten vom Typ `String` über die Methode `setProperty(String, String)` sind demnach auch folgende direkte Aufrufe der `put(Object, Object)`-Methode der `Hashtable<K,V>` möglich, aber wenig sinnvoll:

```
public static final void main(final String[] args) throws IOException
{
    final Properties props = new Properties();

    try (final InputStream in = new BufferedInputStream(
         new FileInputStream("src/main/resources/AppConfig.properties")))
    {
        props.load(in);

        // unerwartet kann man beliebige Objekte in Properties speichern
        // Einfügen eines Person-Objekts
        props.put("MIC", new Person("Micha", "Aachen", 39));

        // kein Zugriff auf Property mit getProperty()
        System.out.println("getProperty()='" + props.getProperty("MIC") + "'");

        // aber Zugriff über die Basisklasse mit get()
        System.out.println("get()='" + props.get("MIC") + "'");
    }
}
```

Listing 9.22 *Ausführbar als* '**APPPROPERTIESPROBLEMS**'

Startet man das Programm APPPROPERTIESPROBLEMS, so sieht man, dass unerwarteterweise ein neues, beliebiges Property mit dem Schlüssel `MIC` und einem `Person`-Objekt als Wert eingefügt und auch ausgelesen werden kann:

```
getProperty()='null'
get()='Person: Name='Micha' City='Aachen' Age='39''
```

In der Regel ist die Anzahl der Konfigurationsparameter und die Komplexität der Zugriffe deutlich größer als in den obigen Beispielen. Dann gewinnen Themen wie eine bessere Abstraktion, ein Fehler vermeidender Zugriff und eine zentrale Fehlerbehandlung mehr an Bedeutung.

Aus den vorherigen Kritikpunkten ergeben sich die folgenden Verbesserungswünsche beim Zugriff auf Konfigurationsparameter:

- **Definition eines Aufzählungstyps** – Festlegung auf eine fest definierte Menge an Schlüsseln zum Zugriff auf die Konfigurationswerte
- **Kapselung** – Einschränkung des Zugriffs auf die Aufzählungswerte und bei der Speicherung auf Werte vom Typ `String`
- **Fehlerbehandlung** – Zentrale Behandlung von Fehlern beim Dateizugriff mit der Möglichkeit, in einsetzenden Applikationen darauf reagieren zu können
- **Zentralisierung** – Zentrale Definition und Abstraktion von einer konkreten Konfigurationsdatei

Verbesserung durch Definition eines Aufzählungstyps Weil es sich bei Konfigurationsparametern meistens um eine überschaubare Anzahl an Werten handelt, bietet es sich an, eine `enum`-Aufzählung mit gültigen Property-Namen zu definieren, wodurch man Magic Strings beim Zugriff auf die Properties vermeidet.

```
enum PropertyName
{
    PDF_VIEWER("pdf.viewer"), DB_USER("db.user"), DB_PASSWORD("db.password");

    final String propertyKey;

    PropertyName(final String propertyKey)
    {
        this.propertyKey = propertyKey;
    }
}
```

Durch diese Art der Definition können mögliche Probleme durch Tippfehler (nahezu) ausgeschlossen werden. Das gilt – wie bereits bemerkt – für Tippfehler in der Applikation. Derartige Fehler in der Konfigurationsdatei erschlägt man damit nicht.

Verbesserung der Kapselung Um die Verarbeitung von Konfigurationsdateien zu vereinfachen und um Implementierungsdetails zu verstecken, ist es sinnvoll, in eigenen Applikationen Zugriffe auf die Klasse `Properties` zu kapseln. Wir definieren dazu eine Klasse `AppProperties`. Anstatt von der Klasse `Properties` abzuleiten, nutzen wir hier Delegation. Somit können wir die Schnittstelle selbst festlegen und verhindern ungewünschte Zugriffe auf die Klasse `Properties` bzw. deren Basisklasse `Hashtable<K,V>`. Um auch Inkonsistenzen bei konkurrierenden Zugriffen bei Multithreading zu vermeiden, sind einige Methoden synchronisiert.

Durch den Einsatz von Delegation muss kein fremdes API realisiert werden. Stattdessen lässt sich die Schnittstelle der eigenen Klasse anforderungsgerecht zuschneiden und nur tatsächlich benötigte Methoden anbieten. Weiterhin können Zugriffsmethoden ausschließlich mit Aufzählungswerten arbeiten, sodass der Zugriff einerseits typsicher und andererseits nur auf vorhandene Elemente möglich ist. Anwendungsfehler durch

falsche Schlüsselwerte, wie sie bei Schlüsseln vom Typ `String` auftreten können, werden somit automatisch ausgeschlossen, weil der Zugriff auf Konfigurationswerte lediglich über eine wohldefinierte Menge von Schlüsselwerten möglich ist.

```
public synchronized String getProperty(final PropertyName key)
{
    return properties.getProperty(key.propertyKey);
}

public synchronized void setProperty(final PropertyName key, final String value)
{
    properties.setProperty(key.propertyKey, value);
}
```

Verbesserung der Fehlerbehandlung Die Behandlung von Fehlern wird in aufrufenden Programmteilen erleichtert, indem eine Zugriffsmethode `getPropertyFilePath()` bereitgestellt wird, die den voll qualifizierten Pfad zur Konfigurationsdatei zurückliefert:

```
public static String getPropertyFilePath()
{
    return new File(FILE_PATH).getAbsolutePath();
}
```

Verbesserung durch Zentralisierung Zur Erleichterung eines zentralen Zugriffs auf die Konfigurationsdaten empfiehlt es sich, die Klasse `AppProperties` als SINGLETON (vgl. Abschnitt 18.1.4) zu realisieren. Dort wird der Dateizugriff durch die Methode `readAppProperties()` bereitgestellt. Das Speichern geschieht mit der korrespondierenden Methode `writeAppProperties()`:

```
public final class AppProperties
{
    private static final String FILE_PATH =
                                "src/main/resources/AppConfig.properties";

    private static final AppProperties INSTANCE = new AppProperties();

    private final Properties properties = new Properties();

    public static final AppProperties getInstance()
    {
        return INSTANCE;
    }

    public synchronized void readAppProperties() throws IOException
    {
        try (final InputStream in = new BufferedInputStream(
                                    new FileInputStream(FILE_PATH)))
        {
            properties.load(in);
        }
    }
```

```
public synchronized void writeAppProperties() throws IOException
{
    try (final OutputStream out = new BufferedOutputStream(
                                  new FileOutputStream(FILE_PATH)))
    {
        properties.store(out, "");
    }
}
// ...
```

Bei Dateizugriffen kann es immer zu Fehlern kommen. ***Es wäre schlechtes Design, eine Fehlerbehandlung in einer Verwaltungsklasse durch Darstellen einer Fehlermeldungsbox zu realisieren.*** Warum? Dadurch könnten höhere Schichten der Applikation nicht mehr auf die Fehlersituationen reagieren. Um das aber zu erlauben, werden möglicherweise auftretende `IOException`s propagiert. Außerdem wird der zum Einlesen verwendete Stream automatisch geschlossen, damit keine Implementierungsdetails, in diesem Fall eine Referenz auf den Stream, nach außen gegeben werden müssen.

Fazit

Mit den sukzessive umgesetzten Erweiterungen ließen sich die angesprochenen Probleme lösen. Weiterhin ist es mit der geänderten Realisierung sogar möglich, Verhalten zu variieren. Konfigurationsdaten können bei Bedarf mithilfe anderer Klassen verwaltet oder in einer anderen Form gespeichert werden, z. B. als XML-Datei. Der Zugriff auf die konkrete Form der Datenhaltung lässt sich verstecken. Beachten Sie, dass wir keinen dieser Vorteile hätten, wenn wir von der Klasse `Properties` ableiten würden! Wir erkennen hier, warum es häufig empfehlenswert ist, Komposition und Delegation statt Vererbung zur Erweiterung von Funktionalität zu nutzen.

Bewertung der Klasse `Properties`

Ein Einsatz der Klasse `Properties` bietet sich bei hauptsächlich lesendem Zugriff von Konfigurationswerten an. Der große Vorteil ist die menschenlesbare Speicherung der Daten und die damit verbundene leichte Änderbarkeit der Konfigurationswerte mit einem beliebigen Texteditor.

Werden die Voreinstellungen modifiziert und sollen diese anschließend wieder persistiert werden, so macht sich die Implementierungsvererbung von der Klasse `Hashtable<K,V>` negativ bemerkbar: Beim Speichern mit `store()` besitzen die Werte keine feste Reihenfolge, und zudem gehen per Hand eingefügte Kommentare verloren. Schwerwiegender sind aber folgende konzeptuelle Unzulänglichkeiten, die unabhängig von der Nutzung einer `Hashtable<K,V>` sind:

- Es wird nicht zwischen globalen und benutzerspezifischen Einstellungen unterschieden.
- Eine hierarchische Strukturierung der Daten wird nicht unterstützt, sondern lediglich eine ungeordnete Sammlung verschiedener Schlüssel-Wert-Paare.

Beide Nachteile könnten durch Nutzung mehrerer `Properties` behoben werden. Eine einfache Strukturierung der Konfigurationswerte wird häufig durch eine Punktnotation in der Art `Gliederungsebene1.Ebene2.Wert` nachgebildet. Dies wurde bereits für die Trennung von Parametern zur PDF-Verarbeitung mit Präfix `pdf` und zur Konfiguration einer Datenbank mit Präfix `db` eingesetzt.

Tipp: System-Properties und Umgebungsvariablen

System-Properties Standardmäßig sind diverse Informationen in sogenannten System-Properties gespeichert, etwa Informationen über das Betriebssystem, die Java-Version usw.

```
public static void main(final String[] args)
{
    final Properties sysProps = System.getProperties();
    final Map<String, String> sortedProperties = new TreeMap<>(sysProps);

    for (final Map.Entry<String, String> entry : sortedProperties.entrySet())
    {
        System.out.println("Key = " + entry.getKey() +
                           " / Value = " + entry.getValue());
    }
}
```

Listing 9.23 *Ausführbar als* '**SYSTEMPROPERTIESEXAMPLE**'

Das Programm SYSTEMPROPERTIESEXAMPLE gibt diverse System-Properties aus, von denen einige nützliche in Tabelle 9-2 aufgeführt sind.

Beim Start eines Java-Programms können zusätzliche Werte per Kommandozeilenoption '`-D`' und der Syntax '`-DmyPropKey=myValue`' angegeben werden. Dabei folgt der Name des Schlüssels, hier »`myPropKey`«, direkt dem '`-D`'. Die so angegebenen Werte werden zu den normalen System-Properties hinzugefügt. Die Abfrage einzelner Werte erfolgt über die Klasse `System` und deren Methode `getProperty(String)`. ***Es empfiehlt sich jedoch, eigene Properties zu verwenden, um die System-Properties nicht zu »verschmutzen«.***

Umgebungsvariablen Über die statische Methode `getenv()` der Klasse `System` erhält man Zugriff auf Umgebungsvariablen des Betriebssystems:

```
public static void main(final String[] args)
{
    final Map<String, String> systemEnv = System.getenv();

    for (final Map.Entry<String, String> entry : systemEnv.entrySet())
    {
        System.out.println("Key = " + entry.getKey() +
                           " / Value = " + entry.getValue());
    }
}
```

Listing 9.24 *Ausführbar als* '**SYSTEMENVEXAMPLE**'

Tabelle 9-2 *Einige nützliche System-Properties*

Property-Name	Inhalt
java.version	Java-Versionsnummer
java.home	Installationsverzeichnis des JREs
java.class.version	Versionsnummer der Java-`.class`-Dateien (JDK 8 = 52.0, JDK 9 = 53.0, JDK 10 = 54.0, JDK 11 = 55.0, JDK 12 = 56.0, JDK 13 = 57.0, JDK 14 = 58.0, JDK 15 = 59.0)
java.class.path	Aktueller `CLASSPATH`
os.name	Name des Betriebssystems (Mac OS X usw.)
user.name	Name des angemeldeten Benutzers
user.home	Home-Verzeichnis des angemeldeten Benutzers
user.dir	Aktuelles Arbeitsverzeichnis

9.5.3 Weitere Möglichkeiten zur Konfigurationsverwaltung

Mit CSV- bzw. XML-Dateien lassen sich auch komplexere Datenstrukturen abbilden und man ist flexibler als bei Schlüssel-Wert-Paaren, repräsentiert durch `Properties`. An einem Beispiel lernen wir nun die Verarbeitung von CSV-Daten kennen. Statt trockener Anwendungsdaten nutzen wir als Eingabe eine Liste von Spielständen.

Beispiel: Einlesen von Highscores aus einer CSV-Datei

Stellen wir uns eine x-beliebige Spieleapplikation vor, die es einem Spieler erlaubt, entsprechende Punktzahl vorausgesetzt, sich in einer Highscore-Liste zu verewigen. Es soll eine Speicherung in einer Datei und ein Einlesen beim Programmstart erfolgen. Die Highscores werden dazu als Liste von Elementen der im folgenden Listing gezeigten Immutable-Klasse `Highscore` verwaltet – wir nutzen hier das Java-14-Feature der `record`s.

```
record Highscore(String name, int points, int level, LocalDate day)
{
}
```

Nehmen wir weiterhin an, die Spielstände wären etwa wie folgt in kommaseparierter Form in einer Datei `Highscores.csv` gespeichert. In diesem Beispiel sind bewusst auch fehlerhafte Einträge dargestellt, die dazu dienen, die Implementierung einer robusten Fehlerbehandlung zu demonstrieren und zu testen:

```
# Name, Punkte, Level, Datum
Matze,    1000,     7,   12.12.2019
Peter,     985,     6,   11.11.2019
ÄÖÜßöäü,   777,     5,   10.10.2019

# Fehlender Datumswert
Peter,     985,     6

# Falsches Format des Levels
Peter,     985,    A6,   11.11.2019

# Fehlerhaftes Datumsformat
Micha,     100,     1,   1/1/2020
```

Betrachten wir die Implementierung der Klasse, die die dargestellte Datei mit Spielständen einliest. Erwähnenswert ist, dass im Sourcecode verschiedene mit JDK 8 eingeführte Neuerungen, etwa `Files.readAllLines()`, `Iterable<E>.forEach()`, `Optional<T>` und Lambdas, zum Einsatz kommen. Bei `readAllLines()` ist es möglich, das Encoding der Eingabedaten anzugeben, hier durch die Konstante `ISO_8859_1` repräsentiert:

```
public final class HighscoresFromCsvImporter
{
    private static final Logger LOGGER =
                         LogManager.getLogger("HighscoresFromCsvImporter");

    public static List<Highscore> readHighscoresFromCsv(final String fileName)
    {
        final List<Highscore> highscores = new LinkedList<Highscore>();

        try
        {
           // JDK 8: readAllLines(), forEach(), Optional<T> & Lambdas
           final List<String> lines = Files.readAllLines(
                              Paths.get(fileName), StandardCharsets.ISO_8859_1);
           lines.forEach(line ->
           {
               final Optional<Highscore> optHighscore =
                                         extractHighscoreFrom(line);
               optHighscore.ifPresent(highscore -> highscores.add(highscore));
           });
        }
        catch (final IOException e)
        {
            LOGGER.warn("processing of file '" + fileName + "' failed: " + e);
        }

        return highscores;
    }

    // ...
}
```

Die Extraktionsarbeit wird durch die im Anschluss gezeigte Methode `extractHighscoreFrom(String)` geleistet. Der Datenstrom wird durch Aufruf der Methode `split(String)` der Klasse `String` ausgewertet und in einzelne Tokens unterteilt. Wir stellen zuerst sicher, dass die erwartete Anzahl an Tokens vorliegt, und protokol-

lieren Verstöße. Beim Umwandeln der textuellen Informationen nutzen wir die Methoden `Integer.parseInt()` für Zahlen und `LocalDate.parse()` für Datumsangaben. Weil Letztere im deutschen Format vorliegen, verwenden wir hier einen speziellen `DateTimeFormatter`. Leer- oder Kommentarzeilen erkennen wir durch die Methode `isEmptyLineOrComment()` und überspringen diese.

Liegen alle Daten im gewünschten Format vor, wird daraus ein unveränderliches `Highscore`-Objekt erstellt. Unvollständige oder ungültige Objektzustände sind somit ausgeschlossen.

```
private static Optional<Highscore> extractHighscoreFrom(final String line)
{
    final int VALUE_COUNT = 4;

    // Spalte die Eingabe mit ';' oder ',' auf
    final String[] values = line.split(";|,");

    //  Behandlung von Leerzeilen, Kommentaren und unvollständigen Einträgen
    if (isEmptyLineOrComment(values) || values.length != VALUE_COUNT)
    {
        return Optional.empty();
    }

    try
    {
        //  Auslesen der Werte als String + Typprüfung + Konvertierung
        final String name = values[0].trim();
        final int points = Integer.parseInt(values[1].trim());
        final int level = Integer.parseInt(values[2].trim());
        final String dateAsString = values[3].trim();

        final DateTimeFormatter dateTimeFormatter =
                                DateTimeFormatter.ofPattern("dd.MM.yyyy");
        final LocalDate day = LocalDate.parse(dateAsString, dateTimeFormatter);

        return Optional.of(new Highscore(name, points, level, day));
    }
    catch (final NumberFormatException e)
    {
        LOGGER.warn("Skipping invalid point or level value '" + line + "'");
    }
    catch (final DateTimeParseException e)
    {
        LOGGER.warn("Skipping invalid date value '" + line + "'");
    }
    return Optional.empty();
}

private static boolean isEmptyLineOrComment(final String[] values)
{
    return (values.length == 1 && (values[0].trim().length() == 0) ||
           // Ignoriere Kommentare, die auch ',' oder ';' enthalten
           (values.length >= 1 && values[0].trim().startsWith("#")));
}
```

Das folgende Listing zeigt die Klasse `ReadHighscoresFromCsvExample`, die die zuvor beschriebene Funktionalität aufruft:

```
public static final void main(final String[] args)
{
    final String path = "src/main/resources/Highscores.csv";
    final List<Highscore> highscores =
                          HighscoresFromCsvImporter.readHighscoresFromCsv(path);

    new AppFrame(highscores).setVisible(true);
}
```

Listing 9.25 *Ausführbar als* '**READHIGHSCORESFROMCSVEXAMPLE**'

Das Programm READHIGHSCORESFROMCSVEXAMPLE liest alle Werte aus der Datei `Highscores.csv` ein und filtert nicht valide Daten heraus. Anschließend werden dann alle korrekten Daten in Form einer Tabelle in einem Swing-GUI dargestellt. Das ist in Abbildung 9-2 gezeigt.

Abbildung 9-2 *Applikationsfenster der Highscore-Liste*

Nachdem wir – eine Ausführung des Programms vorausgesetzt – die Klasse konkret in Aktion erlebt haben, möchte ich noch auf einige Dinge etwas genauer eingehen.

Behandlung von Fehlern beim Öffnen der Datei Im Falle einer Exception beim Öffnen der Datei `Highscores.csv` protokollieren wir ein derartiges Problem und beenden die Verarbeitung. In jedem Fall werden durch den Einsatz von ARM (vgl. Abschnitt 5.6.3) automatisch Aufräumarbeiten durchgeführt. Das sorgt dafür, dass allozierte Systemressourcen wieder freigegeben werden.

Auch die Applikation kann geregelt weiterarbeiten. Sie erhält in diesem Fall entweder eine leere Highscore-Liste oder eine solche, die mit den Daten gefüllt ist, die bis zum Auftreten des Problems eingelesen werden konnten.[9] Alternativ könnte man Exceptions an die Applikation durchreichen und dort eine Fehlermeldung erzeugen.

[9] Das Verhalten kann manchmal problematisch sein, etwa wenn die Forderung nach »alles oder nichts« besteht.

Behandlung von Leerzeilen und Kommentaren Die eingelesenen Zeilen werden mithilfe der Methode `split(String)` der Klasse `String` gemäß einer übergebenen Trennzeichenfolge in Tokens zerlegt. Als Ergebnis erhält man ein `String[]`, das der Hilfsmethode `isEmptyLineOrComment(String[])` als Eingabe dient. Dort erfolgen verschiedene Prüfungen, die es erlauben, sowohl Leerzeilen als auch Kommentarzeilen bei der Auswertung zu überspringen.

Solche Hilfsmethoden sorgen für mehr Lesbarkeit im eigentlichen Applikationscode und halten diesen frei von Implementierungsdetails.

Ignorieren von unvollständigen Eingaben Beim Einlesen von textuellen Daten aus CSV-Dateien kann man Inkonsistenzen in den gespeicherten Daten nicht ausschließen. So etwas sollte die Auswertung berücksichtigen und fehlertolerant darauf reagieren. In diesem Beispiel wird bei unvollständiger Angabe von Informationen eine Warnmeldung in einer Log-Datei protokolliert, die entsprechende Zeile übersprungen und das Parsing mit der nächsten Zeile der Eingabe fortgesetzt. Alternativ kann es aber im Falle von Fehlern sinnvoll sein, entweder gar keine Daten zu lesen oder zusätzlich noch eine Liste mit Fehlern aufzubereiten.

Auswertung der Informationen Sind alle benötigten Daten vorhanden, d. h., liegen genau vier Tokens in der Eingabe vor, so werden zunächst führende und nachfolgende Leerzeichen der Eingabewerte durch einen Aufruf der Methode `trim()` der Klasse `String` entfernt. Dadurch ist es bei der Angabe in der CSV-Datei erlaubt, beliebig viele Leerzeichen zur Ausrichtung der Daten zu verwenden, ohne dass sich dies auf die Daten selbst auswirkt. Der textuelle Wert für den Namen wird ohne weitere Prüfung übernommen. Die beiden Zahlenwerte werden durch die statische Methode `parseInt(String)` der Klasse `Integer` in einen `int`-Wert umgewandelt. Wird in der Eingabe keine Zahl angegeben, so löst dies eine `NumberFormatException` aus und es wird kein `Highscore`-Objekt erzeugt. Die Angabe des Datums wird mithilfe der Klasse `java.time.format.DateTimeFormatter` (siehe Abschnitt 8.1.12) auf Gültigkeit untersucht. Neben den eher syntaktischen Prüfungen auf Zahl oder Datum könnte man mithilfe einer Prüfmethode `validateInputs()` eine weiter gehende semantische Prüfung durchführen, um etwa negative Level- oder Punktzahlen oder gar Datumswerte aus der Zukunft zurückzuweisen. Darüber hinaus gibt es noch eine Obergrenze der Level usw. Sind alle Werte gültig, so werden diese dem Konstruktor der Klasse `Highscore` als Parameter zur Erzeugung eines neuen Objekts übergeben.

Hinweis: Einsatz einer Bibliothek zur Konfigurationsverwaltung

Durch den Einsatz der Bibliothek Apache Commons Configuration kann die Konfiguration von eigenen Programmen vereinfacht werden. Damit lassen sich leichter verschiedenste Arten von Konfigurationen, etwa aus Properties, aus XML-Dateien oder aus Servlet-Parametern, verarbeiten. Weitere Informationen finden Sie online unter `http://commons.apache.org/configuration/`.

10 Multithreading-Grundlagen

Das Thema Multithreading und Parallelverarbeitung mehrerer Aufgaben gewinnt durch Computer mit mehreren Prozessoren (CPUs) oder Prozessoren mit mehreren Rechenkernen (Multicores) zunehmend an Bedeutung. Ziel ist es, komplexe Aufgaben innerhalb von Programmen in voneinander unabhängige Teilaufgaben zu untergliedern, die parallel abgearbeitet werden können. Ist dies der Fall, spricht man von Nebenläufigkeit (Concurrency).

Java bietet zwar einen einfachen Zugang zur Programmierung mit Threads, allerdings verleitet dies manchmal dazu, Multithreading einzusetzen, ohne die resultierenden Konsequenzen zu beachten. Dadurch kommt es in der Praxis aber immer mal wieder zu Problemen. Dieses Kapitel soll helfen, Multithreading mit seinen Möglichkeiten (aber auch Fallstricken) kennenzulernen. Generell gilt zudem zu bedenken, dass sich das JDK und die angebotenen Möglichkeiten enorm entwickelt haben, sodass man sich eher selten noch auf die tieferen Ebenen hinab begeben wird. Die neueren Konzepte stellt das folgende Kapitel vor.

Mit diesen Hinweisen im Hinterkopf wollen wir im Anschluss die Basisbausteine kennenlernen und diverses Grundlagenwissen aufbauen. Eine Einführung in das Thema Multithreading mit `Thread` und `Runnable` gibt Abschnitt 10.1. Wenn in einem Programm Aufgaben durch mehrere Threads bearbeitet werden, müssen deren Berechnungsergebnisse abgeglichen oder zusammengeführt werden. Das ist Thema von Abschnitt 10.2. Auch die Kommunikation zwischen Threads hat Einfluss auf eine erfolgreiche Zusammenarbeit. Darauf gehe ich in Abschnitt 10.3 ein. Ebenso spielt das Java-Memory-Modell eine wichtige Rolle. Einige Details dazu liefert Abschnitt 10.4. Die Einführung in das Themengebiet Multithreading wird in Abschnitt 10.5 mit der Vorstellung verschiedener Besonderheiten beim Einsatz von Threads abgerundet.

Grundlagen zu Parallelität und Nebenläufigkeit

Moderne Betriebssysteme beherrschen sogenanntes ***Multitasking*** und führen mehrere Programme gleichzeitig aus oder vermitteln dem Benutzer zumindest die Illusion, dass verschiedene Programme gleichzeitig ausgeführt würden. Man spricht in diesem Zusammenhang von Parallelität oder Nebenläufigkeit. Da diese Begriffe in der Literatur nicht einheitlich gebraucht werden, werde ich im Folgenden auch keine Unterscheidung vornehmen.

Wie eingangs erwähnt, besitzen Computer heutzutage normalerweise Prozessoren mit mehreren Kernen oder gar mehrere Prozessoren und erlauben so, dass das Betriebssystem verschiedene Programme direkt auf alle verfügbaren Prozessoren bzw. Kerne aufteilen kann. Allerdings kann zu jedem Zeitpunkt jede Recheneinheit (Prozessor bzw. Kern) nur Anweisungen genau eines Programms abarbeiten. Sollen mehrere Programme parallel ausgeführt werden, müssen diese intelligent den jeweiligen Recheneinheiten zugeordnet werden. Bei nur einer Recheneinheit ist dies bereits bei zwei Programmen der Fall. Ganz allgemein gilt für eine n-Prozessor-Maschine[1], dass eine Verteilung bei mehr als n Programmen notwendig wird. Ein sogenannter ***Scheduler*** legt fest, welches Programm auf welcher Recheneinheit ausgeführt wird, und schaltet bei Bedarf zwischen Programmen um. Dadurch wird ein zuvor unterbrochenes Programm etwas später an gleicher Stelle fortgesetzt. Diese Vorgehensweise sorgt dafür, dass die Ausführung jeweils in kleinen Schritten erfolgt, und erlaubt es, andere Programme minimal zeitlich versetzt und damit scheinbar parallel ausführen zu können.

Laufende Programme können wiederum aus »schwergewichtigen« ***Prozessen*** und »leichtgewichtigen« ***Threads*** bestehen. Prozesse sind im Gegensatz zu Threads bezüglich des von ihnen verwendeten Speichers voneinander abgeschottet und belegen unterschiedliche Bereiche. Dadurch beeinflusst ein abstürzender Prozess andere Prozesse (meistens) nicht. Allerdings erschwert diese Trennung auch die Zusammenarbeit und Kommunikation der Prozesse untereinander. Dazu wird die ***Interprozesskommunikation*** notwendig. Diese kann nicht durch normale Methodenaufrufe erfolgen, sondern man nutzt eine spezielle ***Middleware***, die den Transport von Daten zwischen Prozessen oder gar Anwendungen, das sogenannte ***Messaging***, realisiert. Beispiele sind JMS (Java Messaging Service) sowie SOAP (Simple Object Access Protocol) oder das leichtgewichtigere REST (REpresentational State Transfer).

Ein Java-Programm wird im Betriebssystem durch einen Prozess einer JVM ausgeführt. Ein solcher Prozess kann wiederum verschiedene eigene Threads starten. Alle Threads in einer JVM teilen sich dann den gleichen Adressraum und Speicher. Änderungen an Variablen sind dadurch theoretisch für alle Threads sichtbar. In der Realität ist es durch das Java-Memory-Modell jedoch etwas komplizierter. Abschnitt 10.4 geht detailliert darauf ein. Damit sich zwei Threads beim Zugriff auf dieselben Variablen oder Ressourcen nicht gegenseitig stören, müssen sich diese abstimmen, wenn sie zugreifen wollen. Dazu kann man sogenannte ***kritische Bereiche*** definieren, die zu einer Zeit exklusiv immer nur von einem Thread betreten werden können. Damit vermeidet man konkurrierende Zugriffe auf gemeinsame Daten durch verschiedene Threads und erreicht so eine ***Synchronisierung***. Allerdings birgt dies die Gefahr, dass sich Threads gegenseitig behindern oder sogar blockieren. Eine solche Situation nennt man ***Verklemmung*** (oder ***Deadlock***) und muss vom Entwickler selbst verhindert werden. Strategien dazu stellt Abschnitt 10.2.1 vor. Beginnen wir nun mit dem Einstieg in das Multithreading mit Java und schauen uns Threads und deren Kommunikation relativ ausführlich an, da diese Themen elementare Grundlagen darstellen.

[1] Vereinfachend wird hier kein Unterschied zwischen n CPUs und n Rechenkernen gemacht.

10.1 Threads und Runnables

Jeder Thread wird in Java von einer Instanz der Klasse `java.lang.Thread` oder einer davon abgeleiteten Klasse modelliert. Dabei ist es wichtig, Folgendes zu unterscheiden:

1. Den tatsächlichen Thread, der eine Aufgabe ausführt und vom Betriebssystem erzeugt und verwaltet wird.
2. Das zugehörige `Thread`-Objekt, das den zuvor genannten Thread des Betriebssystems repräsentiert und Steuerungsmöglichkeiten auf diesen bietet.

Beim Programmstart erzeugt und startet die JVM automatisch den sogenannten `main`-Thread, der die `main()`-Methode der Applikation ausführt. Ausgehend von diesem Thread können weitere Threads erzeugt und gestartet werden.

Betrachten wir zunächst, wie wir eine abzuarbeitende Aufgabe beschreiben, bevor wir uns dem Ausführen von Threads zuwenden.

10.1.1 Definition der auszuführenden Aufgabe

Zur Definition der abzuarbeitenden Aktionen eines Threads nutzt man die Klasse `Thread` selbst oder das Interface `java.lang.Runnable`. Beide bieten eine `run()`-Methode, die mit »Leben gefüllt« werden muss. Das kann auf zwei Arten geschehen:

1. Ableiten von der Klasse `Thread` und Überschreiben der `run()`-Methode
2. Implementieren des Interface `Runnable` und damit der `run()`-Methode

Betrachten wir beide Varianten anhand der zwei einfachen Klassen `CountingThread` und `DatePrinter`. Abbildung 10-1 zeigt das zugehörige Klassendiagramm.

Abbildung 10-1 *Thread und Runnable*

Die Klasse `CountingThread` erweitert die Klasse `Thread`. Die Implementierung der `run()`-Methode erhöht im Abstand von einer Sekunde einen Zähler und gibt dessen Wert anschließend auf der Konsole aus. Die Klasse `DatePrinter` implementiert das Interface `Runnable` und gibt im Takt von fünf Sekunden das aktuelle Datum aus. Beide Klassen verwenden zum Warten einer Zeitspanne in Millisekunden eine Utility-Klasse `SleepUtils`, die wir später in Abschnitt 10.1.4 kennenlernen werden. Nur im `CountingThread` wird auf die Aufforderung zum Abbruch per `interrupt()` geprüft, wodurch auch vor dem 50. Durchlauf ein Abbruch möglich ist.

Hinweis: Beschränkungen von `run()`

Wie gesehen, lässt sich die abzuarbeitende Aufgabe beschreiben, indem man die `run()`-Methode implementiert. Diese besitzt durch das Interface `Runnable` eine vordefinierte Signatur ohne Aufrufparameter. Oftmals benötigt man zur Ausführung einer Aufgabe aber gewisse Kontextinformationen, die man jedoch nicht an `run()` übergeben kann. In eigenen Realisierungen von `Runnable` bzw. `Thread` nutzt man daher Attribute zur Parametrierung.

Eine weitere Einschränkung der `run()`-Methode besteht darin, dass diese keinen Rückgabewert definiert. Wenn diese Anforderung besteht, sollte man das später in Abschnitt 11.2 beschriebene Executor-Framework und seine Bestandteile `Callable<V>` und `Future<V>` aus dem Package `java.util.concurrent` verwenden. Damit wird eine Realisierung auf logischer statt eher technischer Ebene möglich und man vermeidet eigene, mühsame Implementierungen von `Runnable`. Dort müsste man zur Rückgabe von Berechnungsergebnissen Attribute sowie zugehörige Accessor-Methoden vorsehen, damit Aufrufer auf das Ergebnis der Verarbeitung zugreifen können.

10.1.2 Start, Ausführung und Ende von Threads

Starten von Threads

Die Konstruktion eines `Thread`-Objekts in einem Thread A führt nicht dazu, dass ein Ausführungspfad abgespalten wird. Erst durch Aufruf der `start()`-Methode entsteht ein eigenständiger Thread: Dieser Thread arbeitet danach unabhängig von anderen Threads und dem `main`-Thread weiter. In folgendem Beispiel werden die zwei vorgestellten unterschiedlichen Realisierungen von Threads jeweils erzeugt und gestartet:

```
public static void main(final String[] args)
{
    final Thread derivedThread = new CountingThread();
    derivedThread.start();

    final Thread threadWithRunnable = new Thread(new DatePrinter());
    threadWithRunnable.start();
}
```

Listing 10.1 *Ausführbar als* **'THREADEXAMPLE'**

Wir sehen, dass man ein `Thread`-Objekt durch Aufruf von `start()` direkt starten kann. Die nebenläufige Ausführung eines `Runnable` ist nicht direkt möglich, sondern dazu muss dieses einem `Thread`-Objekt im Konstruktor übergeben werden. In beiden Varianten wird die `run()`-Methode des Threads abgearbeitet. Dazu muss in eigenen Ableitungen von `Thread` die `run()`-Methode implementiert werden. Für `Runnable`s und die Basisklasse `Thread` delegiert die `run()`-Methode des `Thread`-Objekts die Ausführung an die `run()`-Methode des `Runnable`-Objekts. Das ist bereits in Abbildung 10-1 durch `target.run()` angedeutet. Die Variante basierend auf `Runnable` ist OO-technisch sauberer: Es findet keine Ableitung von der Utility-Klasse `Thread` statt und die Funktionalität ist als Einheit gekapselt. Bei Ableitungen von der Klasse `Thread` wird oftmals nur die `run()`-Methode überschrieben und es lassen sich dann keine `Runnable`s mehr damit ausführen, was gegen das LSP (vgl. Abschnitt 3.5.3) verstößt. Dieses Problem wurde zur Demonstration in den `CountingThread` integriert.

Ausführen von Threads

Nach Aufruf von `start()` kommt es zu einer Verarbeitung und zur Ausführung des Threads durch den Thread-Scheduler. In der Folgezeit wird die `run()`-Methode so lange ausgeführt, bis deren letztes Statement erreicht ist (oder ein Abbruch erfolgt, etwa durch Auslösen einer Exception). Bis dahin wird der Thread normalerweise bei der Abarbeitung der in der `run()`-Methode aufgeführten Anweisungen einige Male vom Thread-Scheduler unterbrochen, um andere Threads auszuführen. Über die Methode `isAlive()` kann man ein `Thread`-Objekt befragen, ob es noch aktiv ist und vom Thread-Scheduler verwaltet wird. Die Methode liefert dann den Wert `true`.

Ein `Thread`-Objekt führt seine durch die `run()`-Methode beschriebene Aufgabe nur genau einmal nebenläufig aus. Eine Wiederholung ist nicht möglich: Ein erneuter Aufruf von `start()` löst eine `java.lang.IllegalThreadStateException` aus.

Achtung: Versehentlicher direkter Aufruf von `run()`

Manchmal sieht man Sourcecode, der die Methode `run()` direkt aufruft, um einen neuen Thread zu erzeugen und die Aufgabe parallel auszuführen. ***Das funktioniert jedoch nicht korrekt!*** Stattdessen werden die Anweisungen der Methode `run()` einfach synchron im momentan aktiven Thread wie jede normale andere Methode ausgeführt. Dieser Fehler ist relativ schwierig zu finden, da die gewünschten Aktionen abgearbeitet werden – in diesem Fall allerdings nicht nebenläufig.

Ende der Ausführung von Threads

Nachdem das letzte Statement der `run()`-Methode ausgeführt wurde, endet auch der Thread. Das zugehörige `Thread`-Objekt bleibt jedoch weiter erhalten, stellt aber nur noch ein Objekt wie jedes andere dar. Man kann weiterhin auf die Attribute des `Thread`-Objekts zugreifen, um beispielsweise die Ergebnisse einer Berechnung zu ermitteln. Ein Aufruf von `isAlive()` liefert dann den Wert `false`.

Manchmal soll die Ausführung der `run()`-Methode zu einem beliebigen Zeitpunkt abgebrochen werden. Leider lassen sich Threads nicht so einfach beenden wie starten. Es gibt zwar im API eine zu `start()` korrespondierende Methode `stop()`, doch diese ist als `@deprecated` markiert und sollte nicht (mehr) verwendet werden, da sie verschiedene Probleme auslösen kann. Zum Verständnis der Details müssen wir zunächst einige andere Themen besprechen. Abschnitt 10.5.3 greift das Thema »Beenden von Threads« erneut auf und geht detaillierter auf zugrunde liegende Probleme und mögliche Lösungen ein.

Erwähnenswert ist noch, dass die JVM so lange läuft, wie noch mindestens ein Thread ausgeführt wird – das gilt nur für die bislang vorgestellten sogenannten User-Threads, für die in Abschnitt 10.5.1 besprochenen Daemon-Threads gilt das nicht.

Thread-Prioritäten

Jeder Thread besitzt eine Ausführungspriorität als Zahlenwert aus dem vordefinierten Bereich von 1 bis 10, entsprechend den Konstanten `Thread.MIN_PRIORITY` und `Thread.MAX_PRIORITY`. Diese Priorität beeinflusst, wie der Thread-Scheduler zu aktivierende Threads auswählt. Threads höherer Priorität werden normalerweise bevorzugt, allerdings wird dies nicht garantiert, und jede JVM oder das Betriebssystem können dies anders handhaben. Dadurch können z. B. Threads mit der Priorität n und $n+1$ durch den Thread-Scheduler ohne Unterschied zur Ausführung ausgewählt werden.

Beim Erzeugen übernimmt ein Thread die Priorität des erzeugenden Threads, die oftmals dem Wert `Thread.NORM_PRIORITY` (5) entspricht. Diese lassen sich nachträglich über die Methoden `getPriority()` und `setPriority(int)` abfragen und verändern. Dabei sind nur Werte aus dem durch die obigen Konstanten definierten Wertebereich erlaubt.[2] In der Regel muss die Priorität nicht angepasst werden. Hierbei gibt es zwei Ausnahmen: Für im Hintergrund zu erledigende, nicht zeitkritische Aufgaben kann z. B. der Wert 3 verwendet werden. Für zeitkritische Berechnungen bietet sich eine Priorität nahe `MAX_PRIORITY` an, etwa der Wert 8. Doug Lea gibt in seinem Buch »Concurrent Programming in Java« [48] folgende Empfehlungen:

Tabelle 10-1 *Empfehlungen für Thread-Prioritäten*

Priorität	Verwendungszweck
10	Krisenmanagement
7 – 9	Interaktive, ereignisgesteuerte Aufgaben
4 – 6	Normalfall und I/O-abhängige Aufgaben
2 – 3	Berechnungen im Hintergrund
1	Unwichtige Aufgaben

[2] Werte außerhalb dieses Wertebereichs führen zu einer `IllegalArgumentException`.

Threads und Thread-Gruppen

Eingangs erwähnte ich, dass beim Programmstart von der JVM automatisch der `main()`-Thread erzeugt und gestartet wird. Dabei wird auch eine sogenannte Thread-Gruppe erzeugt, in der verschiedene Threads zusammengefasst werden können und die im JDK einem `java.lang.ThreadGroup`-Objekt entspricht. Ein Thread ist immer genau einer Thread-Gruppe zugeordnet. Thread-Gruppen wiederum können hierarchisch kombiniert werden. Bei der Zugehörigkeit zu einer Thread-Gruppe verhält es sich wie bei der Priorität: Ein Thread wird automatisch in der Gruppe des erzeugenden Threads angelegt, sofern dies nicht bei der Konstruktion anders spezifiziert wird.

In der Praxis werden Sie eher selten mit Thread-Gruppen arbeiten – trotzdem kann deren Kenntnis und Einsatz nützlich sein, etwa um die aktiven Threads und ihre Thread-Gruppen analysieren zu können. Nachfolgend entwickeln wir die benötigte Funktionalität in Form einer Hilfsmethode `dumpThreads()` in einer Utility-Klasse `ThreadUtils`. In der Hilfsmethode ermitteln wir die Thread-Gruppe des aktuellen Threads durch Aufruf von `getThreadGroup()`. Für diese bestimmen wir dann mithilfe der Methode `activeCount()` die Anzahl der laufenden Threads der Gruppe und füllen anschließend durch Aufruf von `enumerate(Thread[])` ein `Thread`-Array mit den momentan aktiven Threads:

```
public final class ThreadUtils
{
   public static void dumpThreads()
   {
      final ThreadGroup group = Thread.currentThread().getThreadGroup();
      final int activeCount = group.activeCount();
      final Thread[] threads = new Thread[activeCount];
      group.enumerate(threads);

      System.out.println("Thread-Group " + group + " contains " + activeCount +
                                                  " threads");

      for (final Thread thread : threads)
      {
         System.out.println("Thread " + thread);
      }
   }
}
```

Verdeutlichen wir uns den Einsatz der erstellten Methode am bereits kennengelernten Beispiel THREADEXAMPLE, das wir wie folgt erweitern:

```
public static void main(final String[] args) throws InterruptedException
{
    final Thread derivedThread = new CountingThread();
    derivedThread.setName("CountingThread");
    derivedThread.start();

    final Thread threadWithRunnable = new Thread(new DatePrinter());
    // threadWithRunnable.setName("DatePrinter");
    threadWithRunnable.start();

    ThreadUtils.dumpThreads();
```

```
    Thread.sleep(7_000);
    threadWithRunnable.interrupt();
    derivedThread.interrupt();
}
```

Listing 10.2 *Ausführbar als* '**DUMPTHREADSEXAMPLE**'

Führen wir das Programm DUMPTHREADSEXAMPLE aus, so sehen wir, dass die beiden Threads parallel zum `main`-Thread in der Thread-Gruppe `main` gestartet werden. Es kommt in etwa zu folgender Ausgabe, wobei die Angaben in eckigen Klammern die Informationen Thread-Name, Priorität und Parent-Thread darstellen:

```
0
Thread-Group java.lang.ThreadGroup[name=main,maxpri=10] contains 3 threads
Thread Thread[main,5,main]
Thread Thread[CountingThread,5,main]
Thread Thread[Thread-1,5,main]        // Thread Thread[DatePrinter,5,main]
Sat Dec 31 18:28:16 CET 2016
1
2
...
```

Anhand des Sourcecodes und der Ausgabe des Programms erkennen wir noch einen Trick aus der Praxis: Es empfiehlt sich, Threads durch Aufruf der Methode `setName(String)` der Klasse `Thread` sprechende Namen zu geben. Tut man dies nicht, wird der Name des Threads einfach durchnummeriert. Sprechende Namen für Threads können eine Fehlersuche enorm erleichtern. Im obigen Konsolenauszug sieht man den Unterschied in der Verständlichkeit zwischen `CountingThread` und `Thread-1` (für den `DatePrinter`-Thread) bereits sehr deutlich.

10.1.3 Lebenszyklus von Threads und Thread-Zustände

Threads haben einen definierten Lebenszyklus, der durch eine festgelegte Menge an Zuständen beschrieben wird. Gültige Zustände sind als innere `enum`-Aufzählung `State` in der Klasse `Thread` definiert. Durch Aufruf der Methode `getState()` kann man den momentanen Thread-Zustand ermitteln. Mögliche Rückgabewerte sind in Tabelle 10-2 aufgelistet und kurz beschrieben.

Für das Verständnis der Zusammenarbeit von Threads ist es wichtig, den Lebenszyklus von Threads und die dabei eingenommenen Thread-Zustände zu kennen. Im Folgenden stelle ich daher die Bedingungen und Auslöser für Zustandswechsel vor.

Nachdem ein Thread erzeugt wurde, wechselt dieser durch den Aufruf seiner `start()`-Methode in den Zustand `RUNNABLE`. Meistens gibt es mehrere Threads in diesem Zustand. Da auf einer 1-Prozessor-Maschine jedoch jeweils nur ein Thread ausgeführt werden kann, ist es Aufgabe eines vom Betriebssystem bzw. der JVM bereitgestellten ***Thread-Schedulers***, alle Threads zu überwachen und den nächsten auszuführenden zu bestimmen, indem ein Thread aus allen denjenigen ausgewählt wird, die sich im Zustand `RUNNABLE` befinden. Der gewählte Thread wird dann aktiv (dies wird nicht durch einen eigenen Zustand beschrieben, man kann sich allerdings einen

Tabelle 10-2 *Thread-Zustände*

Thread-Zustand	Beschreibung
`NEW`	Der Thread wurde erzeugt, ist aber noch nicht gestartet.
`RUNNABLE`	Der Thread ist lauffähig oder wird gerade ausgeführt.
`BLOCKED`	Der Thread ist in seiner Ausführung blockiert und wartet auf den Zutritt in einen kritischen Bereich (vgl. Abschnitt 10.2.1).
`WAITING`	Der Thread wartet unbestimmte Zeit auf eine Benachrichtigung durch einen anderen Thread (vgl. Abschnitt 10.3).
`TIMED_WAITING`	Identisch mit `WAITING`, allerdings wird maximal eine angegebene Zeitdauer auf eine Benachrichtigung gewartet.
`TERMINATED`	Der Thread ist beendet.

»künstlichen« Unterzustand `ACTIVE` im Zustand `RUNNABLE` vorstellen). Auf einer 1-Prozessor-Maschine existiert immer genau ein aktiver Thread, auf einer Mehrprozessormaschine sind es in der Regel mehrere. Unabhängig davon kann es aber beliebig viele Threads geben, die zur Ausführung bereit sind (Zustand `RUNNABLE`). Abbildung 10-2 stellt die Thread-Zustände und mögliche Zustandswechsel dar.

Abbildung 10-2 *Thread-Zustände*

Nicht jeder Thread im Zustand `RUNNABLE` ist tatsächlich auch immer lauffähig. Er kann in einem speziellen »blockiert«-Zustand verharren, wenn er zwar läuft, aber gerade auf eine Ressource wartet, etwa einen Dateizugriff. Dadurch kann der Thread nicht weiterarbeiten und ist in seiner Ausführung blockiert. Der Zustand `RUNNABLE` wird dabei jedoch nicht verlassen. Dagegen beschreibt der Zustand `BLOCKED` das Warten auf die Zuteilung des Zutritts zu einem momentan durch einen anderen Thread belegten kritischen Bereich. Im Zustand `BLOCKED` erhält ein Thread keine Rechenzeit und konkurriert dadurch nicht mit anderen Threads um Zuteilung des Prozessors. Gleiches gilt für die beiden Wartezustände `WAITING` und `TIMED_WAITING`.

Zustandswechsel ausführen

Nehmen wir eine normale, nicht blockierende Abarbeitung der Kommandos des aktiven Threads an, so können durch Aufruf der nun vorgestellten Methoden spezielle Zustandswechsel selbst initiiert werden.

```
Thread.yield();
Thread.sleep(500); // => InterruptedException
someObj.wait(500); // => InterruptedException
```

`yield()` Der aktive Thread kann über den Aufruf der statischen Methode `yield()` pausiert werden, verbleibt aber im Zustand `RUNNABLE`. Dadurch wird anderen Threads die Gelegenheit zur Ausführung gegeben. Besonders für lang dauernde Berechnungen scheint es sinnvoll, diese Methode von Zeit zu Zeit aufzurufen. Die Wahl des nächsten auszuführenden Threads erfolgt durch den Thread-Scheduler prioritätsorientiert, aber zufällig. Daher kann möglicherweise der gerade angehaltene Thread sofort wieder gewählt werden, weshalb eine Aufruf von `sleep()` zu bevorzugen ist.

`sleep()` Durch Aufruf der statischen Methode `sleep(long)` wird der momentan aktive Thread für eine angegebene Zeitdauer »schlafen« gelegt. Der Thread wechselt in den Zustand `TIMED_WAITING` und verbraucht dadurch keine CPU-Zeit mehr. Zudem können dann andere Threads ausgeführt werden. Nachdem die angegebene Wartezeit abgelaufen ist, wechselt der ruhende Thread automatisch in den Zustand `RUNNABLE`, was jedoch nicht zum sofortigen Wiederaufnehmen seiner Ausführung führt. Gegebenenfalls werden zunächst noch andere Threads im Zustand `RUNNABLE` ausgeführt.

`wait()` Bei der Zusammenarbeit mehrerer Threads kann einer von diesen seine Ausführung unterbrechen wollen, bis eine spezielle Bedingung erfüllt ist, etwa, wenn Berechnungsergebnisse anderer Threads zur sinnvollen Weiterarbeit benötigt werden. Durch einen Aufruf der Objektmethode `wait()` erfolgt dann ein Wechsel in den Zustand `WAITING` bzw. `TIMED_WAITING`, je nachdem, ob eine maximale Wartezeit übergeben wurde oder nicht. In diesem Zustand verweilt der Thread, bis entweder ein anderer Thread die Methode `notify()` bzw. `notifyAll()` aufruft, um das Eintreten

einer Bedingung zu signalisieren, oder eine optional angegebene Wartezeit verstrichen ist. ***Die in diesem Absatz genannten Methoden `wait()`, `notify()` bzw. `notifyAll()` zur Kommunikation von Threads stammen nicht, wie man zunächst vermuten könnte, aus der Klasse `Thread`, sondern aus der Klasse `Object`***. Dies ist dadurch begründet, dass *mehrere* Threads auf zu schützenden Daten *eines* Objekts arbeiten und die Threads untereinander abgestimmt werden müssen (vgl. Abschnitt 10.2).

10.1.4 Unterbrechungswünsche durch Aufruf von `interrupt()`

Bei der Kommunikation von Threads, auf die ich in Abschnitt 10.3 detaillierter eingehe, kann ein Thread einen anderen Thread in seiner Abarbeitung unterbrechen bzw. beenden wollen. Dazu dient die Methode `interrupt()` der Klasse `Thread`. Diese hat jedoch keine unmittelbare unterbrechende Wirkung, sondern ist lediglich als Aufforderung oder Hinweis zu verstehen. Durch die JVM wird beim empfangenden Thread ein spezielles Interrupted-Flag gesetzt.

Ein Empfänger dieser Aufforderung sollte geeignet darauf reagieren. Einem gerade aktiven Thread ist dies unmittelbar möglich, indem er über einen Aufruf der Methode `isInterrupted()` der Klasse `Thread` prüft, ob er zwischenzeitlich eine solche Aufforderung empfangen hat, d. h., ob das Flag gesetzt ist. In diesem Fall sollte die Abarbeitung der Anweisungen in der `run()`-Methode beendet werden. Die dazu notwendige Prüfung ist gegebenenfalls mehrfach innerhalb der `run()`-Methode auszuführen, um eine zeitnahe Reaktion auf einen Unterbrechungswunsch zu garantieren. Nachfolgend ist dies symbolisch vor den drei Arbeitsschritten `doSomeWork1/2/3()` gezeigt:

```
while (!Thread.currentThread().isInterrupted())
{
    doSomeWork1();

    if (!Thread.currentThread().isInterrupted())
    {
        doSomeWork2();
    }

    if (!Thread.currentThread().isInterrupted())
    {
        doSomeWork3();
    }
}
```

Behandlung von `interrupt()` und `InterruptedException`s

Komplizierter wird die Abfrage, wenn ein Thread momentan nicht aktiv ist, weil er eine gewisse Zeitspanne pausiert (`sleep(long)`) oder aber auf ein bestimmtes Ereignis wartet (`wait()`). Kommt es während dieser Zeit zu einem Aufruf von `interrupt()` durch einen anderen Thread, so wird zwar in beiden Fällen das Flag gesetzt, eine Reaktion ist jedoch nicht sofort möglich, da der Thread noch nicht aktiv ist. Daher wird von

der JVM das Warten abgebrochen und der Thread wechselt in den Zustand `RUNNABLE`. Bei einer anschließenden Aktivierung durch den Thread-Scheduler wird von der JVM eine `java.lang.InterruptedException` ausgelöst.[3] Da es sich um eine Checked Exception handelt, muss diese propagiert oder mit einem `catch`-Block behandelt werden (vgl. Abschnitt 5.6). Dabei ist ein Detail zu beachten, auf das ich nun eingehe.

Existiert ein `catch (InterruptedException)`-Block, so werden seine Anweisungen ausgeführt. Leider sieht man dort häufig in etwa folgende »Behandlung«:

```
try
{
    Thread.sleep(duration);
}
catch (final InterruptedException e)
{
    // ACHTUNG: normalerweise unzureichende Behandlung
    e.printStackTrace();
}
```

Dieses Vorgehen ist nahezu immer problematisch, weil die JVM beim Eintritt in den `catch`-Block das zuvor gesetzte Interrupted-Flag löscht! Dadurch führt eine mit einem »leeren« `catch`-Block abgefangene `InterruptedException` dazu, dass ein Thread einen Unterbrechungswunsch nicht mehr erkennen kann und daher nicht terminiert.[4]

Wie geht man also mit der Situation um? Eine konsistente Abfrage des Interrupted-Flags mit der Methode `isInterrupted()` wird ermöglicht, wenn man im `catch`-Block über einen expliziten Aufruf der Methode `interrupt()` dieses Flag erneut setzt:

```
while (!Thread.currentThread().isInterrupted())
{
    doSomething();

    try
    {
        Thread.sleep(duration);
    }
    catch (final InterruptedException e)
    {
        // Flag erneut setzen und dadurch Abbruch ermöglichen
        Thread.currentThread().interrupt();
    }
}
```

Die gezeigte Schleifenkonstruktion kann man dazu nutzen, Threads gezielt zu beenden. Abschnitt 10.5.3 beschreibt gebräuchliche Alternativen.

---

[3] Daher ist die Exception in den Signaturen der Methoden `sleep(long)` und `wait()` definiert.

[4] Wenn man den `catch`-Block derart implementiert, so kommt es zu einem unterschiedlichen Programmverhalten abhängig vom momentanen Thread-Zustand, wartend oder aktiv. Eine solche Inkonsistenz ist zu vermeiden. Hängt das Programmverhalten von der zeitlichen Abfolge der Anweisungen ab, so spricht man auch von ***Race Conditions***. Details beschreibt Abschnitt 10.2.1.

Entwurf einer Utility-Klasse

Da Aufrufe von `Thread.sleep(long)` in der Praxis immer wieder zum Verzögern oder Warten eingesetzt werden und dabei fälschlicherweise häufig der `catch`-Block leer implementiert wird, bietet sich der Entwurf einer Utility-Klasse `SleepUtils` an. Diese stellt zwei überladene Methoden `safeSleep()` bereit, die für eine korrekte Verarbeitung der Exception sorgen und das Interrupted-Flag erneut setzen:

```
public final class SleepUtils
{
    public static void safeSleep(final TimeUnit timeUnit, final long duration)
    {
        safeSleep(timeUnit.toMillis(duration));
    }

    public static void safeSleep(final long durationInMilliSecs)
    {
        try
        {
            Thread.sleep(durationInMilliSecs);
        }
        catch (final InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }

    private SleepUtils()
    {}
}
```

Bei Zeitangaben bietet sich die Klasse `java.util.concurrent.TimeUnit` an, die im Listing zur Umrechnung von Zeitangaben beliebiger Einheiten in Millisekunden genutzt wird.

Hinweis: Leere `catch`-Blöcke für `InterruptedException`

Findet keine Kommunikation mit anderen Threads statt und ruft man die statische Methode `Thread.sleep(long)` auf, um die Ausführung des eigenen Threads kurzfristig zu unterbrechen, so kann der `catch`-Block in diesem Spezialfall leer implementiert werden. Der Grund ist einfach: Kennen sich Threads nicht, können sich diese auch nicht per `interrupt()` gegenseitig aufrufen. Eine `InterruptedException` kann daher zwar nicht auftreten, muss aber wegen der Definition in der Signatur behandelt oder propagiert werden. Kommunizieren Threads miteinander, deutet eine `InterruptedException` jedoch darauf hin, dass ein Thread einen anderen unterbrechen und stoppen möchte, und darf somit natürlich nicht ignoriert werden. Dazu kann man generell die zuvor gezeigte Lösung verwenden. Eine Abfrage des Interrupted-Flags ist dadurch jederzeit bei Bedarf konsistent möglich.

10.2 Zusammenarbeit von Threads

Bei der Zusammenarbeit von Threads und dem Zugriff auf gemeinsam benutzte Daten müssen einige Dinge beachtet werden. Wir schauen uns zunächst Situationen an, in denen es zu Zugriffsproblemen und Zweideutigkeiten kommt. Zu deren Vermeidung werden anschließend Locks, Monitore und kritische Bereiche vorgestellt.

10.2.1 Konkurrierende Datenzugriffe

Der Zugriff auf gemeinsame Daten muss beim Einsatz von Multithreading immer aufeinander abgestimmt erfolgen, um Konsistenzprobleme zu vermeiden. ***Ein einziger unsynchronisierter, konkurrierender Datenzugriff kann bereits Probleme auslösen.*** Meistens sind solche Situationen schwierig zu reproduzieren, wodurch eine Fehlersuche mühselig wird. Zudem können bereits minimale Änderungen im Zusammenspiel von Threads zu einem Verschwinden der Probleme oder zu ihrem erstmaligen Auftreten führen. Bei derartigen Situationen spricht man auch von ***Race Conditions***: Das Ergebnis einer Berechnung ist dann nicht deterministisch und durch die Reihenfolge oder die zeitlichen Abläufe bei der Ausführung einzelner Anweisungen geprägt.

Betrachten wir als Beispiel den Start einer Applikation, die aus mehreren Komponenten besteht. Diese greifen während ihrer Startphase auf eine Klasse, realisiert gemäß dem SINGLETON-Muster (vgl. Abschnitt 18.1.4), zu. Dies soll dafür sorgen, dass es genau nur eine Instanz einer Klasse gibt.

Angenommen ein Zugriff geschehe mithilfe der folgenden problematischen, aber doch mitunter selbst in Produktivcode zu findenden `getInstance()`-Methode:

```
// ACHTUNG: SCHLECHT !!!
public static BadSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BadSingleton();
    }
    return INSTANCE;
}
```

Untersuchen wir nun, was daran problematisch ist. Nehmen wir dazu vereinfachend an, während des Starts der Applikation würden zwei Komponenten (Thread 1 und Thread 2) auf diese `getInstance()`-Methode zugreifen. Erfolgt der Aufruf nahezu zeitgleich, so ist es möglich, dass einer der aufrufenden Threads direkt nach der `null`-Prüfung durch den Thread-Scheduler unterbrochen und anschließend der andere Thread aktiviert wird. Beide Threads »sehen« damit einen `null`-Wert für die Variable `INSTANCE`, weil ja noch keine Konstruktion und Zuweisung erfolgt ist. Somit wird der Konstruktoraufruf und die Zuweisung für jeden der beiden Threads ausgeführt und es existieren dann zwei Instanzen eines Singletons: Beide Threads besitzen ihre eigene Instanz! Dies widerspricht natürlich vollständig dem Gedanken des Singletons.

Die Wahrscheinlichkeit für eine solche Race Condition durch einen gemeinsamen Zugriff erhöht sich mit der Verweildauer eines Threads im Konstruktor der Klasse `Bad-`

`Singleton`, beispielsweise, wenn dort aufwendige Initialisierungen stattfinden. In diesem Szenario ruft Thread 2 die Methode `getInstance()` auf, während Thread 1 noch im Konstruktor beschäftigt ist: Die Variable `INSTANCE` ist dann aber noch nicht zugewiesen und hat damit immer noch den Wert `null`.

Durch das Java-Memory-Modell (JMM), dessen Implikationen wir in Abschnitt 10.4 genauer betrachten werden, ist es jedoch prinzipiell egal, ob Thread 1 bereits einen Wert `INSTANCE != null` produziert hat. Selbst wenn Thread 1 schon einige Zeit mit einer solchen Referenz arbeitet, ist nicht sichergestellt, dass Thread 2 die Änderungen von Thread 1 an der Variablen `INSTANCE` mitbekommt. Das liegt daran, dass Threads zum Teil einige Variablen in Thread-lokalen Caches zwischenspeichern und die Werte nur zu definierten Zeitpunkten mit dem Hauptspeicher abgleichen. Da für Thread 2 ohne das Schlüsselwort `synchronized` kein Zwang besteht, sich beim Zugriff auf `getInstance()` neu mit dem Hauptspeicher zu synchronisieren, ist es durchaus möglich, dass er mit einem alten Wert der Variablen `INSTANCE` arbeitet. Eine Möglichkeit zur Abhilfe besteht darin, eine Variable als `volatile` zu deklarieren:

```
private volatile Singelton INSTANCE;
```

Obwohl dadurch sowohl bei Lese- als auch bei Schreiboperationen ein Abgleich mit dem Hauptspeicher erfolgt, hilft dies nicht, wenn mehrere Aktionen – etwa eine Prüfung und anschließende Zuweisung – atomar auszuführen sind.

Anhand dieses Beispiels sehen wir, wie wichtig es ist, den Zugriff auf von mehreren Threads gemeinsam benutzte Variablen abzustimmen. Dies ist durch die Definition sogenannter ***kritischer Bereiche*** möglich, zu denen jeweils nur ein Thread zur gleichen Zeit »Zutritt« hat. Im einfachsten Fall kann man dazu das Schlüsselwort `synchronized` auf Methodenebene wie folgt nutzen:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

10.2.2 Locks, Monitore und kritische Bereiche

Um konkurrierende Zugriffe mehrerer Threads für kritische Bereiche zu verhindern, existiert eine Zugangskontrolle, auch ***Monitor*** genannt. Ein solcher Monitor sorgt dafür, dass der Zutritt zu einem kritischen Bereich immer nur einem einzelnen Thread gewährt wird. Das ermöglicht eine spezielle Sperre (***Lock***), die durch das Schlüsselwort `synchronized` gesteuert wird. In Java verwaltet jedes Objekt zwei Sperren: Eine davon bezieht sich auf die Objektreferenz (`this`) und dient dazu, Objektmethoden zu synchronisieren. Die andere wirkt auf die Klassenreferenz (`class`-Referenz), womit die Synchronisation für statische Methoden möglich ist.

Tipp: Locks und die Analogie zum Wartezimmer

Das Vorgehen beim Synchronisieren mit Locks und Monitoren kann man sich mit der Analogie zum Wartezimmer eines Arztes verdeutlichen. Der Arzt stellt die durch den Lock geschützte Ressource dar, die exklusiv von einem Patienten (Thread) belegt wird. Kommen weitere Patienten zum Arzt, so müssen diese zunächst im Wartezimmer (Warteliste) Platz nehmen und gehen dort in einen Wartezustand, einen Zustand der Blockierung für andere Dinge. Die Arzthelferin (Monitor) gewährt den Zutritt zum Doktor, sobald dieser wieder Zeit hat, also der vorherige Patient den Lock zurückgegeben hat.

Vor Eintritt in einen mit `synchronized` markierten kritischen Bereich prüft die JVM, ob der angeforderte Lock verfügbar ist. In diesem Fall wird der Lock an den anfragenden Thread vergeben und der Thread erhält Zutritt zu dem kritischen Bereich. Nach Abarbeitung des `synchronized`-Blocks erfolgt die Freigabe des Locks automatisch. Kann dagegen ein Lock nicht akquiriert werden, so muss der anfragende, momentan aktive Thread darauf warten, wird inaktiv und in eine spezielle Warteliste des Locks eingetragen. Das geschilderte Vorgehen führt zu einem gegenseitigen Ausschluss: ***Ein durch einen kritischen Bereich geschützter Zugriff auf eine gemeinsame Ressource serialisiert die Abarbeitung mehrerer Threads***. Statt einer parallelen Ausführung kommt es zu einer zeitlich versetzten Bearbeitung. Je mehr Threads um den Zugriff auf einen Lock konkurrieren, desto mehr leidet die Nebenläufigkeit. Abbildung 10-3 zeigt dies symbolisch für drei Threads T_1, T_2 und T_3, die eine ungeschützte und eine durch `synchronized` geschützte Methode ausführen.

Abbildung 10-3 *Serialisierung der Abarbeitung durch Locks*

Im ersten Fall kann die Abarbeitung parallel erfolgen (oder zumindest quasi parallel durch mehrfache Thread-Wechsel). Im zweiten Fall müssen Threads warten und werden sequenziell abgearbeitet (aber nicht unbedingt in der Reihenfolge des versuchten Eintritts in den kritischen Bereich).

Ein Monitor sorgt sowohl für die Vergabe eines verfügbaren Locks als auch dafür, dass ein anfragender Thread bei Nichtverfügbarkeit in die Liste der auf diesen Lock wartenden Threads aufgenommen wird. Bei Freigabe des Locks erhält ihn ein beliebiger Thread aus der Warteliste und kann daraufhin seine Ausführung fortsetzen. Diese Aktionen laufen für den Entwickler unsichtbar automatisch durch die JVM ab.

Kritische Bereiche und das Schlüsselwort `synchronized`

Zur Definition kritischer Bereiche gibt es in Java das Schlüsselwort `synchronized`. Dabei existieren zwei Varianten des Einsatzes:

- `synchronized`-Methode – Bei dieser gebräuchlichen Variante kennzeichnet man eine Methode mit dem Schlüsselwort `synchronized`.
- Synchronisationsobjekt – Etwas seltener sieht man, dass ein Block von Anweisungen durch einen `synchronized`-Block ummantelt wird.

`synchronized`-Methode Die bekannteste und bereits beschriebene Variante zur Definition eines kritischen Bereichs ist das ***Synchronisieren der gesamten Methode*** durch Erweitern der Signatur um das Schlüsselwort `synchronized`:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

Zum Ausführen einer synchronisierten Methode muss ein Thread zunächst den entsprechenden Lock zugeteilt bekommen, ansonsten wird seine Abarbeitung angehalten und der Thread wartet auf die Freigabe des benötigten Locks, um diesen zunächst zu akquirieren und anschließend mit der Berechnung fortfahren zu können.

Greifen weitere Methoden auf eine zu schützende Ressource bzw. ein Attribut zu, so müssen alle diese Methoden synchronisiert werden. Aufgrund der Arbeitsweise der Locks wissen wir, dass dadurch alle `synchronized`-Methoden des Objekts für Zugriffe durch andere Threads gesperrt sind. ***Diese Art der Synchronisierung erschwert somit die Nebenläufigkeit***. Die parallele Ausführung weiterer als `synchronized` deklarierter Methoden eines Objekts ist nicht möglich.

Was kann an dem `synchronized` für eine Methode außerdem problematisch sein? Ist beispielsweise die Ausführung einer geschützten Methode zeitintensiv, so müssen andere Threads potenziell recht lange warten, um den Lock zu erhalten. Damit ist klar, dass ein Synchronisieren über Methoden bei dem Wunsch nach viel Parallelität nicht optimal ist. Es gibt aber noch einen weiteren Fallstrick: ***Das synchronisierte Verhalten betrifft nur die deklarierende Klasse, sodass in Subklassen die Synchronisierung durch Überschreiben verloren geht!*** Betrachten wir nun eine elegantere Synchronisierungsvariante.

Synchronisationsobjekt Die weniger bekannte, aber oft sinnvollere Möglichkeit zur Definition eines kritischen Bereichs ist das ***Synchronisieren eines Abschnitts innerhalb von Methoden***. Man nutzt dazu ein sogenanntes ***Synchronisationsobjekt***. Prinzipiell kann jedes beliebige Objekt dazu verwendet werden, seinen Lock zur Verfügung

zu stellen. Um Anwendungsfehler zu vermeiden und die Semantik eines solchen Synchronisationsobjekts besonders herauszustellen, sollte jedoch bevorzugt eine finale Objektreferenz vom Typ `Object` genutzt werden, im folgenden Beispiel `LOCK` genannt. Zum Eintritt in einen durch `synchronized (LOCK)` geschützten kritischen Bereich muss ein Thread den Lock dieses speziellen Objekts erhalten.

Synchronisationsobjekte nutzt man, um feingranulare Sperren zu realisieren, d. h., um kleine Blöcke innerhalb von Methoden zu schützen. Am Beispiel der folgenden Methoden `calcValue1()` und `calcValue2()` zeige ich dies anhand von Berechnungen, die sich zu verschiedenen Zeitpunkten in ihrer Abarbeitung gegenseitig ausschließen sollen. Statt aber beide Methoden über `synchronized` exklusiv auszuführen und damit Nebenläufigkeit zu verhindern, werden in diesen Methoden nur die wirklich kritischen Abschnitte über `synchronized (LOCK)` vor einer gleichzeitigen Ausführung geschützt:

```
// Gemeinsam genutzte und daher zu schützende Daten
private final List<Integer> sharedData = new ArrayList<Integer>();

// Synchronisationsobjekt
private final Object LOCK = new Object();

public int calcValue1()
{
    // Blockiert calcValue2()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    // Aktionen ohne Lock, calcValue2() kann ausgeführt werden

    return 1;
}

public int calcValue2()
{
    // Blockiert calcValue1()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    // Aktionen ohne Lock, calcValue1() kann ausgeführt werden

    // Blockiert calcValue1()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    return 2;
}
```

Mehr Nebenläufigkeit kann man erreichen, wenn man mehrere solcher Lock-Objekte für diejenigen Ressourcen und Attribute einführt, die geschützt werden sollen. Dadurch werden konkurrierende Zugriffe auf andere gemeinsame Attribute nicht mehr blockiert

und jeweils immer nur der benötigte Zugriff exklusiv ausgeführt. Zum Teil lässt sich das zu schützende Objekt selbst als Synchronisationsobjekt verwenden. Allerdings ist dies nur dann möglich, wenn sichergestellt ist, dass sich die Referenz darauf nicht ändert, diese also `final` ist. Ansonsten kann es zu einer `java.lang.IllegalMonitorStateException` kommen. Darauf gehe ich in Abschnitt 10.3 genauer ein. Bedenken Sie aber, dass mit zunehmender Zahl von Lock-Objekten auch die Verständlichkeit und Nachvollziehbarkeit leidet.

Tipp: Fallstricke beim Synchronisieren

Ein Schutz vor konkurrierenden Zugriffen muss immer vollständig erfolgen. ***Ein einziger nicht synchronisierter Zugriff auf ein zu schützendes Attribut reicht aus, um Multithreading-Probleme zu verursachen.*** Dies kann schnell geschehen, wenn verschiedene Synchronisationsvarianten nicht konsequent, sondern beispielsweise gemischt verwendet werden. Somit hält man potenziell den Lock auf das »falsche« Synchronisationsobjekt. Lassen Sie mich – weil es wichtig ist – nochmals auf Folgendes hinweisen: ***Werden nicht alle Zugriffe über denselben Lock geschützt, kann es leicht zu Inkonsistenzen oder Deadlocks kommen***.

Analogie von Synchronisationsobjekt und `synchronized`-Methode

Der Einsatz von Synchronisationsobjekten bietet mehr Möglichkeiten als die Nutzung von `synchronized`-Methoden. Insbesondere kann man das Verhalten von `synchronized`-Methoden durch den Einsatz von Synchronisationsobjekten abbilden, die die `this`-Referenz verwenden, und den synchronisierten Abschnitt auf die komplette Methode ausdehnen. Eine Synchronisierung auf `this` wie folgt

```
void method()
{
    synchronized (this)
    {
        // ...
    }
}
```

entspricht dem Verhalten des Schlüsselworts `synchronized` auf Methodenebene:[5]

```
synchronized void method()
{
    // ...
}
```

Eine analoge Aussage gilt für statische Methoden. Dort wird der Lock durch das Synchronisieren auf die `class`-Variable beschrieben.

[5] Das bezieht sich auf die Semantik. Auf Ebene des Bytecodes gibt es Unterschiede. Schauen Sie es sich per `javap` an. Dieses Tool erlaubt die Aufbereitung von `.class`-Dateien in lesbaren Sourcecode sowie die Anzeige des zugehörigen Bytecodes.

Mächtigkeit von `synchronized`-Blöcken: Synchronisations-Proxy

Gerade haben wir gesehen, dass die Möglichkeiten mit `synchronized`-Blöcken vielseitiger sind als mit dem Schlüsselwort `synchronized`. Was man mit `synchronized`-Blöcken darüber hinaus noch machen kann, betrachten wir am Beispiel einer beliebigen, nicht Thread-sicheren Klasse `ThreadUnsafeClass`. Nehmen wir an, es wäre nun Thread-Sicherheit gefordert und als zusätzliche Randbedingung gelte, dass wir den Sourcecode der Klasse nicht besitzen oder ändern dürfen.

Wenn wir diese Klasse um Thread-Sicherheit oder genauer synchronisierte Zugriffe erweitern wollen, dann müssen wir dafür sorgen, dass alle Methoden in einem geschützten Bereich ausgeführt werden. Dazu können wir eine sogenannte Stellvertreter-Klasse gemäß dem in Abschnitt 18.3.6 beschriebenen PROXY-Muster definieren, die alle öffentlichen Methoden der zu schützenden Klasse implementiert (also deren öffentliche Schnittstelle erfüllt) und dort die Methodenaufrufe an die jeweiligen korrespondierenden Methoden der Originalklasse weiter delegiert. Außerdem muss jede derartige Methodendelegation in einen `synchronized`-Block eingeschlossen werden. Die Zugriffe auf das Objekt vom Typ `ThreadUnsafeClass` mit dem Interface `OriginalIF` werden dann durch Locks von der dekorierenden Klasse geschützt. Folgendes Listing deutet diese Form der Realisierung an:

```
public class SynchronizationProxy implements OriginalIF
{
    private final ThreadUnsafeClass original;

    public void doSomething()
    {
        synchronized (this)
        {
            original.doSomething();
        }
    }
}
```

Ähnlich zu dieser Umsetzung sind auch die `synchronized`-Wrapper des Collections-Frameworks realisiert (vgl. Abschnitt 6.3.2).

Thread-Sicherheit und Parallelität mit »normalen« Collections

Die `synchronized`-Wrapper des Collections-Frameworks ermöglichen einen Thread-sicheren Zugriff. Schauen wir uns kurz einige Besonderheiten an und betrachten folgende synchronisierte Liste von `Person`-Objekten als Ausgangsbasis unserer Diskussion:

```
final List<Person> syncPersons = Collections.synchronizedList(personList);
```

Bei parallelen Zugriffen auf diese Liste können sich Threads gegenseitig blockieren und stören. Die Synchronisierung stellt einen Engpass dar und serialisiert die Zugriffe: Es kommt dadurch zu (stark) eingeschränkter Parallelität. Eine derartige Ummantelung schützt zudem nicht vor möglichen Inkonsistenzen: Wenn man mehrere für sich Thread-sichere Methoden hintereinander aufruft, ist dadurch eine atomare Ausführung

als kritischer Bereich nicht garantiert. Dies habe ich bereits bei der Beschreibung der `synchronized`-Wrapper in Abschnitt 6.3.2 und bei der Darstellung eines Singletons und dessen `getInstance()`-Methode in Abschnitt 10.2.1 diskutiert.

Jeder einzelne Methodenaufruf einer synchronisierten Collection ist für sich gesehen Thread-sicher. Damit ist gemeint, dass Zugriffe mehrerer Threads keine Inkonsistenzen innerhalb der Collection selbst verursachen. Für eine nutzende Komponente sind solche feingranularen Sperren aber häufig nicht ausreichend. Vielmehr sollen Operationen mit mehreren Schritten atomar und Thread-sicher ausgeführt werden. Solche Mehrschrittoperationen sind etwa das Iterieren oder Methoden wie »`testAndGet()`«, die zunächst prüfen, ob ein gewisses Element enthalten ist, und nur dann einen Zugriff bzw. eine Modifikation ausführen. Ohne viel nachzudenken, könnte man auf die etwas naive Idee kommen, die Mehrschrittoperationen durch den Einsatz einzelner Thread-sicherer Methoden wie folgt zu realisieren:

```
// ACHTUNG: nicht Thread-sicher
public Person testAndGet(final int index)
{
    if (index < syncPersons.size())
    {
        // index < size gilt evtl. nicht mehr
        return syncPersons.get(index);
    }
    return null;
}
```

Dieser Ansatz ist – wie schon erwähnt – nicht Thread-sicher: Zur Erinnerung sei nochmals erwähnt, dass ***Thread-sichere Methoden in ihrer Kombination nicht Thread-sicher sind, da nach jedem geschützten Methodenaufruf ein Thread-Wechsel möglich ist.*** Erfolgt die Unterbrechung im obigen Beispiel etwa direkt nach Ausführung der `if`-Bedingungen und verändert ein anderer aktivierter Thread die Datenstruktur, so kann dies verheerende Auswirkungen bei Wiederaufnahme eines unterbrochenen Threads haben: Bei Zugriffen kommt es entweder zu Inkonsistenzen oder Exceptions. Daher müssen zusätzliche Synchronisierungsschritte ausgeführt werden, etwa durch ein Synchronisationsobjekt. In diesem Fall bietet sich die Collection selbst, genauer die finale Referenz `syncPersons`, an. Eine atomar ausgeführte Version der `testAndGet(int)`-Methode kann wie folgt implementiert werden – sofern alle Zugriffsmethoden dieses Synchronisationsobjekt nutzen.

```
public Person testAndGet(final int index)
{
    // Kritischer Bereich für Mehrschrittoperationen
    synchronized (syncPersons)
    {
        if (index < syncPersons.size())
        {
            return syncPersons.get(index);
        }
    }
    return null;
}
```

Tipp: Regeln zur Verwendung von `synchronized`

Um die im Anschluss beschriebenen Deadlocks und andere Probleme möglichst auszuschließen, sollte man Folgendes beachten:

- Ein Thread kann den Lock von einem oder die Locks von verschiedenen Objekten besitzen. Er erhält Zutritt zu den derart geschützten kritischen Bereichen.
- Ein geschützter Bereich sollte möglichst kurz sein. Ideal ist es, wenn nur die wirklich zu schützenden Operationen synchronisiert werden. Nicht zu schützende Teile sollten in andere Methoden verlagert oder außerhalb des geschützten Bereichs durchgeführt werden.
- Aus einem geschützten Bereich sollten keine blockierenden Aufrufe (z. B. `read()` eines `InputStream`) erfolgen, da der Thread den Lock während des Wartens nicht abgibt. Auch Aufrufe von `Thread.sleep(long)` sind zu vermeiden, da auch hier keine Freigabe des Locks erfolgt. Andere Threads werden ansonsten möglicherweise länger blockiert.
- Aus dem vorherigen Punkt ergibt sich indirekt folgender Tipp: ***Wenn man einen Lock hält, sollte man möglichst keine Methoden anderer Objekte aufrufen (oder dabei zumindest große Vorsicht walten lassen)***. Dadurch vermeidet man Aufrufe, die zu Deadlocks führen können.
- Ein Objekt kann mehrere Methoden `synchronized` definieren. Besitzt ein Thread den Lock, können diese Methoden sich gegenseitig aufrufen, ohne dass bei jedem Methodenaufruf erneut versucht wird, den Lock zu bekommen. Wäre dies nicht so, würde sich ein Thread selbst blockieren, wenn er versuchen würde, andere `synchronized`-Methoden aufzurufen. Locks nennt man daher eintrittsinvariant oder auch ***reentrant***[a]. Basierend darauf können `synchronized`-Methoden sich gegenseitig oder gar rekursiv aufrufen. Nur deshalb kam es zu keiner Blockierung beim obigen Beispiel mit der synchronisierten Liste!
- Nicht durch `synchronized` geschützte Methoden können jederzeit von jedem Thread aufgerufen werden.

[a] Man bezeichnet eine Methode als reentrant, wenn sie von mehreren Threads problemlos ohne gegenseitige Beeinflussung gleichzeitig ausgeführt werden kann.

10.2.3 Deadlocks und Starvation

Wie bereits eingangs diesen Kapitels erwähnt, spricht man von ***Deadlocks***, wenn Threads sich gegenseitig blockieren. Als ***Starvation*** bezeichnet man Situationen, in denen es für einen oder mehrere Threads kein Vorankommen im Programmfluss mehr gibt. Beim Einsatz von Multithreading kann beides leicht auftreten.

Deadlocks

Wenn ein beliebiger Thread 1 den Lock auf Objekt A belegt und versucht, den Lock auf Objekt B zu erhalten, kann es zu einem sogenannten ***Deadlock*** kommen, wenn ein anderer Thread 2 bereits den Lock auf Objekt B hält und seinerseits wiederum versucht, den Lock auf Objekt A zu bekommen. Beide warten daraufhin endlos. In der Informatik ist die exklusive Belegung von Ressourcen auch als das ***Philosophenproblem*** bekannt. Stark vereinfacht treffen sich zwei Philosophen P1 und P2 zum Essen an einem Tisch, allerdings haben sie nur eine Gabel und ein Messer. Dies zeigt Abbildung 10-4.

Abbildung 10-4 *Ressourcenproblem*

Wählt Philosoph P1 immer zuerst die Gabel und nimmt Philosoph P2 immer zunächst das Messer, so kann es bei ungünstiger zeitlicher Abfolge (nahezu zeitgleichem Zugriff auf die Besteck-Ressource) dazu kommen, dass beide niemals etwas essen, da ihnen die jeweils andere Ressource fehlt. Halten sich beide an eine identische Reihenfolge erst Gabel, dann Messer (oder umgekehrt), so kann es niemals zu einer (endlosen) gegenseitigen Blockierung kommen.

Hinweis: Vermeidung von Deadlocks

Deadlocks lassen sich vermeiden, indem man Locks und Ressourcen immer in der gleichen Reihenfolge belegt, etwa gemäß dem folgenden Schema: Lock A, Lock B, Unlock B, Unlock A. Hält man sich an diese Regel, so vermeidet man Situationen, in denen ein Thread auf einen anderen wartet und keiner von beiden ausgeführt werden kann, weil beide jeweils auf eine bereits belegte Ressource zugreifen wollen. Bei komplexeren Abläufen ist jedoch nicht immer sofort klar, wer wann welche Ressource belegt und wieder freigibt bzw. sogar, ob dies überhaupt geschieht.

Starvation

Wenn der momentane Besitzer einer Ressource diese nach getaner Arbeit nicht wieder freigibt, obwohl er sie nicht weiter benötigt, kann es zu dem ***Starvation*** genannten Phänomen kommen: Andere auf diese Ressource wartende Threads sind endlos blockiert, da sie vergeblich auf den Erhalt der benötigten Ressource warten. Verhalten sich alle Beteiligten allerdings fair, so muss zwar manchmal einer auf den anderen warten (wenn dieser gerade die benötigten Ressourcen besitzt), aber alle können ihre Aufgabe auf jeden Fall nacheinander bzw. abwechselnd ausführen.

10.2.4 Kritische Bereiche und das Interface `Lock`

Java bietet mit dem Schlüsselwort `synchronized` leider keine Unterbrechbarkeit oder Timeouts beim Zugriff auf Locks. Als Abhilfe gibt es zur Definition kritischer Bereiche ergänzend zu den impliziten Sperren über `synchronized` explizite Sperrvarianten. Die für diese Funktionalität wichtigsten Interfaces sind `Lock`, `ReadWriteLock` und `Condition` sowie die beiden Klassen `ReentrantLock` und `ReentrantReadWriteLock` aus dem Package `java.util.concurrent.locks`. Durch deren Einsatz können einige Schwächen des in die Sprache integrierten Lock- und Synchronisationsmechanismus umgangen bzw. behoben werden. Die wichtigsten Möglichkeiten sind in folgender Aufzählung genannt:

- Ein Thread kann unterbrechbar versuchen, auf einen Lock zuzugreifen.
- Ein Thread kann mit einer maximalen Wartezeit und nicht nur unbeschränkt lange auf den Zugriff eines Locks warten, wie dies bei `synchronized` geschieht.
- Es werden unterschiedliche Arten von Locks unterstützt. Beispielsweise sind dies sogenannte Read-Write-Locks, die mehrere parallele Lesezugriffe erlauben, falls der zugehörige Write-Lock nicht vergeben wurde.
- `Lock`-Objekte können nun nicht nur einen gewissen Block aufspannen, sondern beliebige Bereiche. Dadurch lassen sie sich im Gegensatz zu `synchronized` in verschiedenen Klassen über Block- und Methodengrenzen hinweg einsetzen.

Grundlagen von Locks

`Lock`-Objekte stellen eine konzeptionelle Erweiterung von Synchronisationsobjekten dar. Letztere verwendet man bekanntermaßen wie folgt:

```
synchronized (lockObj)
{
    // Kritischer Bereich
}
```

Da `Lock`-Objekte nicht auf den in die JVM integrierten Mechanismen der Monitore beruhen, muss die Definition eines kritischen Bereichs explizit über Methodenaufrufe programmiert werden. Zum Sperren dient die Methode `lock()`. Diese wird auf einem `Lock`-Objekt aufgerufen. Soll der kritische Bereich beendet werden, muss explizit ein Methodenaufruf von `unlock()` erfolgen:

```
final Lock lockObj = new ReentrantLock();
lockObj.lock();       // Explizit Lock anfordern
try
{
    // Kritischer Bereich
}
finally
{
    lockObj.unlock(); // Explizit Lock freigeben
}
```

Das Listing zeigt den Einsatz des Interface `Lock` und dessen konkreter Realisierung `ReentrantLock`, um ein zu `synchronized` kompatibles Verhalten nachzubilden, das sowohl Eintrittsinvarianz bietet als auch rekursive Methodenaufrufe erlaubt. Die konkreten Implementierungen des Interface `Lock` garantieren zudem ein bezüglich der Sichtbarkeit von Änderungen zu `synchronized` kompatibles Verhalten. Die Hintergründe beschreibt der folgende Infokasten.

Es ist wichtig, dass der Aufruf von `lock()` außerhalb des `try`-Blocks geschieht, damit beim Fehlschlagen des Sperrens durch das `unlock()` im `finally` keine ungewollte Freigabe erfolgt. Nachfolgendes Listing zeigt die falsche Variante, die Sie vermeiden sollten:

```
try
{
    lockObj.lock(); // !!!!!!!!!!!! FALSCH !!!!!!!!!!!!!
    // Kritischer Bereich
}
finally
{
    lockObj.unlock(); // Explizit Lock freigeben
}
```

Info: Konsistenz von Locks und `synchronized`

Es mag zunächst verwundern, dass Locks die Sichtbarkeit von Änderungen wie `synchronized` herstellen, da keine explizite Synchronisierung zu sehen ist. Synchronisierung sorgt neben dem gegenseitigen Ausschluss von Threads auch für Konsistenz von Wertänderungen an Attributen, wodurch Änderungen innerhalb eines `synchronized`-Blocks anschließend für andere lesend zugreifende Threads sichtbar sind. Locks nutzen `volatile`-Attribute, die ebenfalls diesen Abgleich sicherstellen und somit die Konsistenz von Änderungen erreichen.

Parallelität durch den Einsatz von Read-Write-Locks

In der Praxis erfolgen oftmals deutlich mehr Lese- als Schreibzugriffe. Dann bietet sich der Einsatz sogenannter Read-Write-Locks an, die beide Arten von Zugriffen schützen können. Mehrere Threads können Read-Locks halten, um parallele Lesezugriffe zu erlauben, solange kein Thread Schreibzugriffe ausführt bzw. genauer: das Write-Lock akquiriert hat. Das beschriebene Verhalten wird durch die Klasse `ReentrantReadWriteLock` realisiert, die das Interface `ReadWriteLock` implementiert. Dieses bietet Zugriff auf zwei spezielle Locks mit den folgenden Methoden, die `Lock`-Objekte zurückgeben, die dann wie zuvor beschrieben eingesetzt werden können:

```
public interface ReadWriteLock
{
    public Lock readLock();
    public Lock writeLock();
}
```

Während ein Thread das Write-Lock besitzt, können andere Threads das Read-Lock nicht akquirieren. Finden andererseits noch Leseaktionen statt, während ein Thread Zugriff auf das Write-Lock bekommen möchte, muss dieser warten, bis alle lesenden Threads ihre Read-Locks freigegeben haben. Das kann jedoch dazu führen, dass Schreibzugriffe stark verzögert werden, wenn es sehr viele Lesezugriffe gibt. Ist ein solches Verhalten nicht gewünscht, so kann man bei der Konstruktion der Klasse `ReentrantReadWriteLock` einen Parameter übergeben, der dafür sorgt, dass die Klasse sich »fair« verhält. In diesem Modus werden Threads daran gehindert, das Read-Lock zu akquirieren, solange es noch Threads gibt, die auf das Write-Lock warten. Dadurch sollen Schreibzugriffe selbst bei höherem Leseaufkommen möglich sein und Starvation der Schreibzugriffe vermieden werden.

10.3 Kommunikation von Threads

Bis hierher haben wir Möglichkeiten kennengelernt, Thread-sicher auf gemeinsam benutzte Daten zuzugreifen. In der Zusammenarbeit von Threads benötigt man aber häufig außerdem Möglichkeiten zur Abstimmung zwischen Threads – genauer: zu deren Kommunikation. Ein Beispiel dafür ist das sogenannte ***Producer-Consumer-Problem***. Hierbei geht es darum, dass ein Erzeuger (Producer) gewisse Daten herstellt und ein Konsument (Consumer) diese verbraucht und beide über eine gemeinsame Datenstruktur interagieren, wie es symbolisch in Abbildung 10-5 dargestellt ist.

Abbildung 10-5 *Kommunikation von Producer und Consumer*

Im Folgenden stelle ich verschiedene Formen der Kommunikation von `Producer`- und `Consumer`-Klassen vor, damit Sie dabei lauernde Probleme nachvollziehen können.

10.3.1 Kommunikation mit Synchronisation

Als Erstes betrachten wir, wie eine Kommunikation über eine gemeinsame Datenstruktur und den Einsatz von Synchronisation gelöst werden kann. Producer und Consumer implementieren wir als Spezialisierungen von `Runnable`.

Realisierung des Producers

In diesem Beispiel soll der Producer periodisch im Takt einer vorgegebenen Wartezeit `sleepTime` Daten vom Typ `Item` erzeugen und sie in der gemeinsam genutzten Datenstruktur `sharedItems` vom Typ `List<Item>` ablegen. Damit es dabei nicht zu Zugriffskonflikten kommt, muss das Ganze innerhalb eines kritischen Bereichs exklusiv erfolgen, hier über `synchronized (sharedItems)` realisiert:

```
public static class Producer implements Runnable
{
    private final List<Item> sharedItems;
    private final long       sleepTime;

    public Producer(final List<Item> items, final long sleepTime)
    {
        this.sharedItems = items;
        this.sleepTime = sleepTime;
    }

    public void run()
    {
        int counter = 0;

        while (!Thread.currentThread().isInterrupted())
        {
            // Ereugen eines Items
            final Item newItem = new Item("Item " + counter);
            System.out.println("Producing ... " + newItem);
            SleepUtils.safeSleep(sleepTime);

            // Lock akquirieren, dann exklusiv zugreifen und Item hinzufügen
            synchronized (sharedItems)
            {
                sharedItems.add(newItem);
                System.out.println("Produced " + newItem);
            } // Lock wird automatisch freigegeben

            counter++;
        }
    }
}
```

Realisierung des Consumers

Der Consumer schaut zyklisch nach, ob bereits Daten für ihn vorliegen, und liest diese dann aus der Liste `sharedItems`. Ebenso wie der Producer muss auch der Consumer seine Zugriffe exklusiv in einem kritischen Bereich erledigen. Dabei ist zu beachten, dass nur der Zugriff selbst, nicht aber das Warten mit `Thread.sleep(long)` (gekapselt durch `SleepUtils.safeSleep(long)`) im kritischen Bereich erfolgt. Würde `sleep()` im kritischen Bereich aufgerufen, so würde der Lock des `sharedItems`-Objekts nicht freigegeben. Folglich könnte der Producer niemals auf die gemeinsame Liste zugreifen, um Daten abzulegen. Der Consumer würde endlos und vergeblich warten.

```
public static class Consumer implements Runnable
{
    private final List<Item> sharedItems;
    private final long       sleepTime;

    public Consumer(final List<Item> items, final long sleepTime)
    {
        this.sharedItems = items;
        this.sleepTime = sleepTime;
    }

    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            // Status-Flag ist als lokale Variable immer Thread-sicher
            boolean noItems = true;
            while (noItems)
            {
                // Lock akquirieren, dann exklusiv zugreifen und Item auslesen
                synchronized (sharedItems)
                {
                    noItems = (sharedItems.size() == 0);
                    if (noItems)
                       System.out.println("Consumer waiting for items ...");
                    else
                       System.out.println("Consuming " + sharedItems.remove(0));
                } // Lock wird automatisch freigegeben

                // Achtung: sleep() nicht in synchronized aufrufen
                SleepUtils.safeSleep(sleepTime);
            }
        }
    }
}
```

Das hier genutzte aktive Warten wird auch ***Busy Waiting*** genannt. Das sollte man in der Regel vermeiden, weil es Rechenzeit kostet und es elegantere Alternativen gibt. Falls häufig genug `Thread.sleep(long)` (oder `SleepUtils.safeSleep(long)`) aufgerufen wird, ist der Einsatz nicht ganz so tragisch.

Beispiel der Kommunikation

Wir schauen uns nun an, wie die beiden Basisbausteine zusammenarbeiten. Zur Kommunikation wird jeweils eine Instanz eines `Producer`s und eines `Consumer`s erzeugt. Danach legt der Producer jede Sekunde etwas in die gemeinsame Liste und der Consumer schaut alle 500 ms nach, ob etwas zu konsumieren ist – das Ganze läuft 20 Sekunden:

```
public static void main(final String[] args) throws InterruptedException
{
    final List<Item> sharedItems = new LinkedList<>();

    final Thread producer = new Thread(new Producer(sharedItems, 1000));
    final Thread consumer = new Thread(new Consumer(sharedItems, 500));
    producer.start();
    consumer.start();

    TimeUnit.SECONDS.sleep(20); // Warte 20 Sekunden zur Demonstration

    // Stoppe die Verarbeitung
    producer.interrupt();
    consumer.interrupt();
}
```

Listing 10.3 *Ausführbar als* '**PRODUCERCONSUMERSYNCHRONISATIONEXAMPLE**'

Betrachten wir die gekürzte Ausgabe, um das Programm zu analysieren:

```
Producing ... [Item] Item 0
Consumer waiting for items ...
Consumer waiting for items ...
Produced [Item] Item 0
Producing ... [Item] Item 1
Consuming [Item] Item 0
Consumer waiting for items ...
Consumer waiting for items ...
Produced [Item] Item 1
Producing ... [Item] Item 2
Consuming [Item] Item 1
```

Anhand dieser Ausgabe erkennen wir folgende Dinge:

1. Producer und Consumer laufen zwar parallel, aber hier synchron ab: Der produzierte Gegenstand wird immer sofort konsumiert, sobald er verfügbar ist.
2. Das vom Consumer genutzte aktive Warten führt zu ständigen Prüfungen, kostet etwas Rechenzeit und sollte möglichst vermieden werden.
3. Es sind auch andere Wartezeiten für Producer und Consumer denkbar. Würde man etwa die Wartezeiten vertauschen, so würden in einer Sekunde zwei Elemente produziert, der Consumer würde aber nur ein Element konsumieren. Dadurch würde der Zwischenspeicher irgendwann überlaufen. Deshalb sollte die Größe beschränkt werden. Der Producer sollte die Arbeit einstellen, wenn ein maximaler Füllgrad erreicht ist, und ein Consumer nur so lange aktiv werden, wie Elemente vorhanden sind. Dies lässt sich mit aktivem Warten kaum adäquat ausdrücken.

Beginnen wir mit einer alternativen Realisierung, die das aktive Warten adressiert. Später schauen wir uns dann größenbeschränkte Datenstrukturen an, die dabei helfen, die im dritten Aufzählungspunkt erwähnten Nachteile zu behandeln.

10.3.2 Kommunikation über die Methoden `wait()`, `notify()` und `notifyAll()`

Eleganter als mit Busy Waiting wird die Kommunikation von Producer und Consumer durch die im Anschluss vorgestellten Methoden `wait()`, `notify()` und `notifyAll()`. Wie beim Schlüsselwort `synchronized` werden auch hier die Verwaltung des Locks und die Warte- und Aufweckarbeiten automatisch durch die JVM erledigt.

- `wait()` – Versetzt den aktiven Thread in den Zustand `WAITING` und der belegte Lock wird freigegeben. Um ein endloses Warten auf ein möglicherweise nicht eintretendes Ereignis zu verhindern, kann beim Aufruf von `wait()` ein Timeout angegeben werden: Der Thread wechselt dann in den Zustand `TIMED_WAITING`.
- `notify()` – Informiert einen beliebigen wartenden Thread und versetzt diesen in den Zustand `RUNNABLE`. Unschönerweise lässt sich nicht kontrollieren, welcher Thread benachrichtigt wird. Daher sollte man bei der Kommunikation mehrerer Threads bevorzugt das nachfolgend beschriebene `notifyAll()` verwenden.
- `notifyAll()` – Informiert alle auf den Lock des Objekts wartenden Threads und versetzt diese in den Zustand `RUNNABLE`.

Die genannten Methoden werden nicht auf Threads, sondern auf Objekten aufgerufen. In diesem Fall kontrolliert das Objekt über seinen Lock indirekt den Thread. Wenn ein Thread A während seiner Bearbeitung Daten benötigt oder auf das Eintreten einer speziellen Bedingung warten möchte, so kann er die Methode `wait()` eines Objekts `obj` aufrufen. Ein anderer Thread B kann dadurch bei Bedarf den benötigten Lock erhalten, Berechnungen durchführen und Ergebnisse produzieren. Danach informiert er einen oder mehrere auf den Lock dieses Objekts `obj` wartende Threads durch Aufruf der Methode `notify()` oder bevorzugt `notifyAll()` auf der Objektreferenz `obj`. Einer dieser Threads erhält dann den Lock des Objekts `obj`: Dies kann der wartende Thread A oder ein beliebiger anderer sein. Was dies im Einzelnen bedeutet, zeigen die folgenden Abschnitte.

Producer-Consumer mit `wait()`, `notify()` und `notifyAll()`

Die Methoden zur Steuerung der Kommunikation von Threads nutzen wir nun, um das Producer-Consumer-Beispiel zu vereinfachen und das Busy Waiting zu vermeiden.

Producer Damit der im Anschluss realisierte Consumer nach Aufrufen von `wait()` nicht endlos wartet, muss der `Producer` einen Aufruf der Methode `notify()` bzw. `notifyAll()` ausführen, der zum Aufwachen des `Consumer`s führt. Die `run()`-Methode im `Producer` wird somit wie folgt modifiziert:

```
public void run()
{
    int counter = 0;

    while (!Thread.currentThread().isInterrupted())
    {
        final Item newItem = new Item("Item " + counter);
        System.out.println("Producing ... " + newItem);

        SleepUtils.safeSleep(sleepTime);

        synchronized (sharedItems)
        {
            sharedItems.add(newItem);
            System.out.println("Produced " + newItem);
            // Informiere wartende Threads
            sharedItems.notifyAll();
        }
        counter++;
    }
}
```

Listing 10.4 *Ausführbar als* '**PRODUCERCONSUMEREXAMPLE**'

Consumer Die `run()`-Methode der `Consumer`-Klasse wird so modifiziert, dass vor dem Zugriff auf die Daten in `sharedItems` ein Aufruf von `wait()` in einem kritischen Bereich ausgeführt wird.

```
public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        synchronized (sharedItems)
        {
            try
            {
                System.out.println("Consumer waiting ...");
                sharedItems.wait();
                // ACHTUNG: Potenziell unsicherer Zugriff
                final Item item = sharedItems.remove(0);
                System.out.println("Consuming " + item);
            }
            catch (final InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
        }
        SleepUtils.safeSleep(sleepTime);
    }
}
```

Diese Art der Kommunikation ist klarer und erfordert keine Tricks wie Busy Waiting. Für den Einsatz der Methode `wait()` gibt es allerdings einige Dinge zu bedenken, wie wir es nun sehen werden.

Hintergrundwissen: Gleichwertige Varianten bei `notify()`

Bei der Kommunikation von Threads über die Methoden `wait()` und `notify()` bzw. `notifyAll()` sieht man beim Benachrichtigen folgende zwei Varianten:

```
Variante 1
synchronized (lock)
{
    modCount++;
    lock.notify();
}
```

```
Variante 2
synchronized (lock)
{
    lock.notify();
    modCount++;
}
```

Obwohl es zunächst nicht intuitiv ist, sind beide Lösungen aber gleichwertig. Auf den ersten Blick könnte man meinen, dass in Variante 2 das Hochzählen nach dem Benachrichtigen eines möglicherweise wartenden Threads geschieht und dieser einen falschen Wert ausliest. Tatsächlich erhält ein wartender Thread den Lock jedoch erst beim Verlassen des `synchronized`-Blocks – das vorzeitige Benachrichtigen ändert durch das Halten des Locks nichts am Ablauf. Dennoch ist Variante 2 verwirrend und sollte nicht verwendet werden, um erst gar keine Fragen oder Unsicherheiten aufkommen zu lassen.

IDIOM: WARTEN AUF EINE BEDINGUNG

Grundsätzlich sollte ein wartender Thread, nachdem er aufgeweckt wurde, prüfen, ob tatsächlich die erwartete Situation eingetreten ist. Wenn man `wait()` nutzt, so benötigt man sogar zwei Prüfungen, nämlich eine vor und eine nach dem Aufruf: Zunächst prüft man für den Fall, dass eine Benachrichtigung vor dem Warten durch Aufruf von `wait()` versendet wird. Nach dem Aufwachen prüft man erneut, um sicherzustellen, dass die erwartete Bedingung immer noch gilt, bevor man dann eine Aktion, hier die Methode `doWork()`, ausführt. Das Warten auf produzierte Elemente sollte damit folgendermaßen realisiert werden:

```
if (sharedItems.size() == 0)
    wait();

if (sharedItems.size() > 0)
    doWork();
```

Das Ganze lässt sich eleganter schreiben, doch schauen wir uns zunächst detailliertere Begründungen für die Prüfungen an.

Prüfung vor `wait()` Vor dem Aufruf von `wait()` sollte die gewünschte Bedingung geprüft werden. Das scheint zunächst unlogisch, doch nehmen wir einmal an, der Producer würde deutlich vor dem Consumer gestartet. Er produziert ein Element, legt es in der gemeinsamen Liste ab und ruft `notifyAll()` auf. Zu dem Zeitpunkt wartet aber noch kein Consumer darauf. Die Benachrichtigung geht ins Leere. Später wird dann `wait()` vom Consumer aufgerufen. Er wartet nun bis zum Sankt Nimmerleinstag auf eine Benachrichtigung, die er nie erhält. Aber auch für den Fall, dass der Producer (versehentlich) keine Benachrichtigung versendet, ist die Prüfung hilfreich. Es können trotzdem Elemente in der Datenstruktur vorhanden sein, die zu konsumieren sind.

Prüfung nach `wait()` Der Wartezustand durch Aufruf von `wait()` wird nur dann verlassen, wenn eine Benachrichtigung per `notify()` bzw. `notifyAll()` erfolgt. Es können aber auch andere Threads zwischenzeitlich geweckt worden sein und bei ihrer Ausführung eventuell Daten geändert haben. Daher gilt die erwartete Bedingung möglicherweise nicht mehr, aufgrund derer die Benachrichtigung erfolgte.

Verbesserung der Prüfung Um die genannten Probleme zu adressieren, sollte man `wait()` besser innerhalb einer `while`-Schleife ausführen und dort erneut die Bedingung prüfen. Dabei nutzt man folgendes Idiom:

```
while (!condition)
    wait();
```

Doug Lea schlägt in seinem Buch »Concurrent Programming in Java« [48] vor, die Prüfung von Bedingungen in eigene Methoden auszulagern. Dieses Vorgehen ist sehr empfehlenswert und trägt deutlich zur Lesbarkeit bei. Um beispielsweise im Consumer zu prüfen, ob Daten zur Verarbeitung bereitgestellt wurden, kann folgende Methode `waitForItemsAvailable(List<Item>)` wie folgt realisiert werden:

```
private static void waitForItemsAvailable(final List<Item> items) throws
      InterruptedException
{
    while (items.size() == 0)
        items.wait();
}
```

Neben der besseren Lesbarkeit gibt es einen weiteren triftigen Grund für diese Art der Umsetzung: Erfolgt eine Benachrichtigung mit `notify()`, bevor darauf mit `wait()` gewartet wird, so wird diese Benachrichtigung nicht wahrgenommen und ein Thread wird dann eventuell bis zum Programmende vergeblich auf das Eintreffen einer Benachrichtigung warten. Deshalb muss zusätzlich die Bedingung geprüft werden. Wie gezeigt könnte man das zwar über eine `if`-Abfrage vor dem `wait()` lösen. Eleganter als die eingangs gezeigten zwei Prüfungen über `if`-Abfragen ist aber die gerade vorgestellte Umsetzung der Prüfung in einer `while`-Schleife.

Korrektur Mit dem bis hierher erlangten Verständnis für das Warten mit `wait()` wollen wir die `run()`-Methode des Consumers korrigieren: Dazu nutzen wir die zuvor erstellte Methode `waitForItemsAvailable(List<Item>)` wie folgt:

```
System.out.println(consumerName + " waiting ...");

waitForItemsAvailable(sharedItems);
// Nachfolgende Zugriffe auf items durch waitForItemsAvailable() immer sicher
final Item item = sharedItems.remove(0);
System.out.println(consumerName + " consuming " + item);

SleepUtils.safeSleep(sleepTime);
```

Listing 10.5 *Ausführbar als* **'PRODUCERMULTICONSUMEREXAMPLE'**

Als Folge dieser Korrektur warten anfangs alle Consumer auf das Eintreffen einer Benachrichtigung. Es werden zwar alle Consumer geweckt, allerdings erhält nur einer von diesen zunächst den Lock und konsumiert die produzierten Elemente. Erhalten in der Folgezeit die anderen Consumer den Lock, prüfen diese zunächst wieder die Bedingung. Diese ist eventuell nicht mehr gültig, nämlich dann, wenn der Consumer zuvor alle Elemente konsumiert hat. Dann legen sich die anderen Consumer schlafen und warten wieder auf das Eintreten der Bedingung.

Erweiterung auf eine Größenbeschränkung

Bisher haben wir lediglich den Fall betrachtet, dass die Consumer schneller verbrauchen als der Producer Dinge herstellen kann. Für den Fall, dass der Producer allerdings wesentlich schneller ist, kämen die Consumer nicht mehr hinterher und die gemeinsame Datenstruktur würde immer voller, bildlich gesprochen: Sie würde überlaufen.

Daher bietet es sich zum Datenaustausch an, einen Zwischenspeicher begrenzter Kapazität zu nutzen. Wird dessen Kapazität erreicht, so muss der Producer mit dem Fortsetzen des Produzierens darauf warten, dass der oder die Consumer wieder Platz geschaffen haben. Umgekehrt gilt: Sind die Consumer schneller als der Producer, und gibt es keine zu verarbeitenden Dinge mehr, so sollten die Consumer so lange warten, bis wieder Dinge zu konsumieren sind.

Wir erweitern das Beispiel. Wenig objektorientiert wäre es, die Anforderungen an die Verwaltung gemeinsamer Daten in den Klassen des Producers und Consumers zu realisieren. Stattdessen könnten wir eine typsichere Containerklasse definieren – sinnvoller ist es, sich im JDK umzuschauen. Weil beim Einsatz von Multithreading gemeinsam genutzte, größenbeschränkte Datenstrukturen elementar sind, wurden mit JDK 5 entsprechende Containerklassen eingeführt. Diese werden in Abschnitt 11.1 kurz beschrieben und helfen dabei, die Kommunikation verschiedener Threads zu steuern. Nachfolgend nutzen wir aus dem Package `java.util.concurrent` das Interface `BlockingQueue<E>` sowie die konkrete Implementierung `LinkedBlockingQueue<E>`, um den Datenaustausch zu realisieren:

```
public static void main(final String[] args) throws InterruptedException
{
   final int MAX_QUEUE_SIZE = 7;
   final BlockingQueue<Item> items = new LinkedBlockingQueue<>(MAX_QUEUE_SIZE);

   final Thread producer = new Thread(new Producer(items, 100));
   producer.start();

   // Kurz warten, um Größenbeschränkung besser zu zeigen
   SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

   final Thread consumer1 = new Thread(new Consumer(items, 1000, "Consumer 1"));
   final Thread consumer2 = new Thread(new Consumer(items, 1000, "Consumer 2"));
   consumer1.start();
   consumer2.start();
   ...
}
```

Listing 10.6 *Ausführbar als* '**PRODUCERCONSUMERBLOCKINGQUEUEEXAMPLE**'

Neben der Low-Level-Kommunikation mit `wait()`, `notify()` und `notifyAll()` haben wir die Datenstruktur `BlockingQueue<E>` genutzt, um auf höherer Abstraktionsebene Daten austauschen zu können. Nun schauen wir uns weitere Varianten an.

10.3.3 Abstimmung von Threads

Bei der Zusammenarbeit mehrerer Threads müssen mitunter Aufgaben in parallele Teile aufgespalten und später an Synchronisationspunkten wieder zusammengeführt werden. Bekanntermaßen kann man mit der Methode `isAlive()` ein `Thread`-Objekt fragen, ob dieses durch einen Aufruf von `start()` als neuer Ausführungspfad gestartet wurde und die Methode `run()` abgearbeitet wird. Vor und nach der Abarbeitung von `run()` stellt ein Thread nur ein ganz normales Objekt dar. Ein Aufruf von `isAlive()` liefert dann den Wert `false` und nur während der Abarbeitung von `run()` den Wert `true`.

Startet man Threads zur Berechnung und möchte man auf deren Beendigung und die Ergebnisse warten, so ist dies durch Aufruf der Methode `join()` möglich. Repräsentiert beispielsweise die Referenzvariable `workerThread` einen Thread, so wartet der momentan aktive Thread durch den Aufruf

```
workerThread.join();
```

synchron, d. h. blockierend, auf die Beendigung des Threads `workerThread` und wird erst danach fortgesetzt. Um nicht endlos zu warten, falls der andere Thread niemals endet, kann eine Timeout-Zeit in Millisekunden per `join(long)` angegeben werden, nach der das Warten auf das Ende des Threads abgebrochen wird. Statt einer Millisekundenangabe kann die Klasse `TimeUnit` für sprechendere Wertangaben sorgen.

Die Arbeitsweise von `isAlive()` und `join()` verdeutlicht folgendes Programm:

```
public static void main(final String[] args) throws InterruptedException
{
    final List<Item> items = new LinkedList<>();
    final Thread producerThread = new Thread(new Producer(items, 1000));
    producerThread.start();

    // Aktueller Thread wird für 5 Sekunden angehalten
    producerThread.join(TimeUnit.SECONDS.toMillis(5));
    System.out.println("after join(): producer is alive? " +
                                    producerThread.isAlive());
    // 1000 ms Produktionszeit und 5000 ms Wartezeit => ca. 5 Items
    System.out.println("Item-Count after join(): " + items.size());
    // Der Producer arbeitet noch 2 Sekunden weiter ...
    SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

    // Der Producer wird aufgefordert, nun anzuhalten ...
    producerThread.interrupt();

    // 1000 ms Produktionszeit und 7 s Wartezeit => ca. 7 Items
    System.out.println("Item-Count after interrupt(): " + items.size());
    System.out.println("after interrupt(): producer is alive? " +
                                         producerThread.isAlive());
}
```

Listing 10.7 *Ausführbar als* '**PRODUCERJOINEXAMPLE**'

In diesem Beispiel wartet der aktuelle Thread (hier: der `main`-Thread) durch Aufruf von `join()` fünf Sekunden auf das Ende des `Producer`-Threads. Dieser ist dann aber noch nicht terminiert. Die Ausgabe der Anzahl der produzierten Elemente stellt daher nur einen Zwischenstand dar. Der `Producer`-Thread setzt seine Arbeit fort und wird nach weiteren zwei Sekunden über einen Aufruf von `interrupt()` zum Anhalten aufgefordert und, weil er darauf adäquat reagiert, auch beendet.

Starten wir das Programm PRODUCERJOINEXAMPLE, so kommt es zu folgender Ausgabe (hier auf das Wesentliche gekürzt), die das zuvor Gesagte bestätigt:

```
[...]
Producing ... [Item] Item 4
Produced [Item] Item 4
Producing ... [Item] Item 5
after join(): producer is alive? true
Item-Count after join(): 5
Produced [Item] Item 5
Producing ... [Item] Item 6
Produced [Item] Item 6
Item-Count after interrupt(): 7
after interrupt(): producer is alive? false
```

Achtung: Verwirrende Syntax von `join()`

Die Schreibweise beim Aufruf von `join()` ist verwirrend, aber trotzdem korrekt. Besser verständlich und objektorientierter wäre etwa folgende Schreibweise gewesen: `currentThread.waitForTermination(workerThread);`

Besonderheiten bei `join()`

Beim Aufruf von `join()` sind zwei Dinge zu beachten: Zum einen sollte dies nur erfolgen, wenn der Thread, auf den gewartet werden soll, bereits gestartet wurde. Der Grund ist folgender: Ein Aufruf von `join()` wird sofort beendet, falls der zu überprüfende Thread noch nicht gestartet oder bereits beendet ist. Der aufrufende Thread wird dann augenblicklich fortgesetzt.

Kommunikation mehrerer Threads mit `join()` Zum Teil soll ein Thread auf das Ende mehrerer Threads warten. Aufgrund der Tatsache, dass `join()` sofort zurückkehrt, wenn ein zu überprüfender Thread beendet ist, kann man einfach mehrere `join()`-Aufrufe in beliebiger Reihenfolge hintereinander ausführen. Die Reihenfolge ist unbedeutend, denn die maximale Wartezeit ist sowieso durch den Thread mit der längsten Ausführungszeit bestimmt.

Die Abstimmung der Ablaufreihenfolge mehrerer Threads lässt sich zwar über Aufrufe von `join()` realisieren, etwa indem abhängige Threads erst nach den Threads gestartet werden, auf deren Beendigung sie warten sollen. Sinnvoller ist es aber, dafür spezielle Klassen zu verwenden, etwa einen Semaphor.

10.3.4 Unerwartete `IllegalMonitorStateException`s

Beim Einsatz von Multithreading und der Kommunikation von Threads treten zum Teil unerwartet `java.lang.IllegalMonitorStateException`s auf. Manche Entwickler sind dann ratlos. Wie kann es dazu kommen?

Zu einer solchen Exception kommt es, wenn der Thread, der eine der Methoden `wait()`, `notify()` bzw. `notifyAll()` aufruft, zu der Zeit nicht den Lock des zugehörigen Objekts besitzt. Diese Tatsache kann zur Kompilierzeit nicht geprüft werden und führt erst zur Laufzeit zu der genannten Exception.

Ein kurzes Beispiel hilft dabei, diesen Sachverhalt zu verstehen. In nachfolgendem Listing wird über das Objekt `lock` synchronisiert. Auch der Aufruf von `notifyAll()` geschieht augenscheinlich auf diesem Objekt:

```
static Integer lock = new Integer(1);

public static void main(final String[] args)
{
    synchronized (lock)
    {
        System.out.println("lock is " + lock);
        lock++;
        System.out.println("lock is " + lock);
        lock.notifyAll();
    }
    System.out.println("notWorking");
}
```

Listing 10.8 *Ausführbar als* **'ILLEGALMONITORSTATEEXAMPLE'**

Beim Ausführen des Programms ILLEGALMONITORSTATEEXAMPLE kommt es zu einer `IllegalMonitorStateException` und man erhält in etwa folgende Ausgabe:

```
lock is 1
lock is 2
Exception in thread "main" java.lang.IllegalMonitorStateException
    at java.lang.Object.notifyAll(Native Method)
```

Tritt eine solche `IllegalMonitorStateException` auf, kann man folgendermaßen vorgehen, um die Ursache zu finden:

1. Prüfe, ob die Aufrufe an `wait()`, `notify()` bzw. `notifyAll()` innerhalb eines `synchronized`-Blocks ausgeführt werden.
   a. Ist dies der Fall, weiter mit Punkt 2.
   b. Erfolgt ein Aufruf ohne `synchronized`, so muss die Aufrufhierarchie der Methode verfolgt werden. Falls dort ein `synchronized` gefunden wird, geht es weiter mit Punkt 2. Ansonsten ist an geeigneter Stelle eine Synchronisierung einzufügen.
2. Prüfe, ob das korrekte Objekt zum Synchronisieren verwendet wird.

Für das Beispiel ist der erste Punkt offensichtlich gegeben. Anschließend prüft man, ob auch das korrekte Synchronisationsobjekt verwendet wird. Auf den ersten Blick scheint dies ebenfalls zuzutreffen. Doch der Operator '++' und das Auto-Boxing führen hier dazu, dass wir nicht auf derselben Instanz des `Integer`-Objekts `lock` die Anweisung `synchronized(lock)` ausführen, auf der wir auch `lock.notifyAll()` aufrufen. Somit kommt es zur `IllegalMonitorStateException`.

Tipp: Regeln zu Synchronisationsobjekten

Beim Einsatz von Synchronisationsobjekten helfen folgende Hinweise:

1. Verwende als Synchronisationsobjekt möglichst die Basisklasse `Object`.
2. Synchronisiere immer auf unveränderlichen Referenzen.
3. Vermeide den Einsatz von Objekten vom Typ `Integer`, `Long` und `String` als Synchronisationsobjekte. Das verhindert Effekte durch Auto-Boxing oder Stringmanipulationen.

`Lock`-Klassen und `IllegalMonitorStateExceptions`

Beim Einsatz der bereits vorgestellten `Lock`-Klassen kann es gleichermaßen zu `IllegalMonitorStateExceptions` kommen. Man muss dann sicherstellen, dass `await()`, `signal()` und `signalAll()` innerhalb eines durch einen Lock geschützten Bereichs aufgerufen werden. Zudem muss man prüfen, ob das korrekte `Condition`-Objekt des `Lock`-Objekts verwendet wird.

10.4 Das Java-Memory-Modell

Bis hierher haben wir bereits einige Fallstricke beim Zugriff auf gemeinsam verwendete Datenstrukturen beim Einsatz von Multithreading kennengelernt. Das Verständnis des Java-Memory-Modells (JMM) hilft, verlässlichere Multithreading-Applikationen zu schreiben. Es legt fest, wie Programme, im Speziellen Threads, Daten in den Hauptspeicher schreiben und wieder daraus lesen. Im Folgenden gehe ich auf die wichtigsten Punkte ein, die in Kapitel 17 der JLS [26] im Detail dargestellt sind. Ich versuche, diese hier möglichst anschaulich und weniger theoretisch als in der JLS zu vermitteln.

Das JMM regelt die Ausführungsreihenfolge und Unterbrechbarkeit von Operationen sowie den Zugriff auf den Speicher und bestimmt damit die Sichtbarkeit von Wertänderungen gemeinsamer Variablen verschiedener Threads. Dabei müssen drei Dinge beachtet werden:

1. **Sichtbarkeit** – Variablen können in Thread-lokalen Caches zwischengespeichert werden, wodurch ihre aktuellen Werte für andere Threads nicht sichtbar sind. Diese Speicherung erfolgt jedoch nicht immer für alle Variablen.
2. **Atomarität** – Lese- und Schreibzugriffe auf Variablen werden für die 64-Bit-Datentypen `long` und `double` ***nicht atomar*** in einem »Rutsch« ausgeführt, sondern durch Abarbeitung mehrerer Bytecode-Anweisungen, die daher unterbrochen werden können.
3. **Reorderings** – Befehle werden gegebenenfalls in einer von der statischen Reihenfolge im Sourcecode abweichenden Reihenfolge ausgeführt. Dies ist immer dann der Fall, wenn der Compiler einige Optimierungen durchgeführt hat.

10.4.1 Sichtbarkeit

Konzeptionell laufen alle Threads einer JVM parallel und greifen dabei auf gemeinsam verwendete Variablen im Hauptspeicher zu. Aufgrund der Zwischenspeicherung von Werten in einem eigenen Cache eines Threads ist nicht in jedem Fall eine konsistente Sicht aller Threads auf diese Variablen gegeben. Findet kein Abgleich der gecachten Daten mit dem Hauptspeicher statt, so sind dadurch Änderungen eines Threads an einer Variablen für andere Threads nicht sichtbar.[6]

Das JMM garantiert einen solchen Abgleich von Daten mit dem Hauptspeicher lediglich zu folgenden Zeitpunkten:

- **Thread-Start** – Beim Start eines Threads erfolgt das initiale Einlesen der verwendeten Variablen aus dem Hauptspeicher, d. h., der lokale Cache eines Threads wird mit deren Werten belegt.

[6] Dieser Abgleich ist auch unter dem Begriff ***Cache-Kohärenz*** bekannt.

- **Thread-Ende** – Erst beim Ende der Ausführung eines Threads erfolgt in jedem Fall ein Abgleich des Cache mit dem Hauptspeicher. Dieser Vorgang ist zwingend notwendig, damit modifizierte Daten für andere Threads sichtbar werden.

Während der Abarbeitung eines Threads erfolgen in der Regel keine Hauptspeicherzugriffe mehr, sondern es wird mit den Daten des Cache gearbeitet. Der Thread »lebt isoliert« in seiner eigenen Welt. Ein expliziter Abgleich von Daten kann über die Schlüsselwörter `synchronized` und `volatile` erzwungen werden. Mit `synchronized` wird ein kritischer Bereich definiert. Dagegen garantiert die Verwendung von `volatile`, dass sowohl der lesende als auch der schreibende Datenzugriff direkt auf dem Hauptspeicher erfolgen. Dadurch wird sichergestellt, dass ein `volatile`-Attribut nie einen veralteten Wert aufweist. Beim Einsatz muss man allerdings beachten, dass die im Folgenden beschriebene Atomarität eingehalten wird.

10.4.2 Atomarität

Wie bereits erwähnt, erfolgt ein Zugriff auf Variablen der 64-Bit-Datentypen `long` und `double` nicht atomar. Für Multithreading ist dies insbesondere für nicht gecachte Variablen zu beachten, da folgendes Szenario entstehen kann: Nachdem die ersten 32 Bit zugewiesen wurden, kann ein anderer Thread aktiviert werden und »sieht« dann möglicherweise einen ungültigen Zwischenzustand der Variablen, hier am Beispiel der Zuweisung eines Werts an die Variable `val` und den Zeitpunkten $t1$ und $t2$ illustriert:

```
    long val = 0x6789ABCD;

t1: long val = 0x0000ABCD;
t2: long val = 0x6789ABCD:
```

Laut JLS erreicht man mit `volatile` einerseits Konsistenz, weil alle Threads die gleichen Werte sehen, und zum anderen werden auch die 64-Bit-Typen `long` und `double` immer atomar geschrieben bzw. gelesen. Somit wäre eine Idee, statt Synchronisation bevorzugt `volatile`-Attribute zum Datenaustausch bei Multithreading einzusetzen und die `increment()`-Methode eines Zählers wie folgt zu schreiben:

```
private volatile long counter = 0;

public void increment()
{
    // Achtung: ++ ist nicht atomar
    counter++;
}
```

Das ist aber weder sinnvoll noch ausreichend: Durch `volatile` wird *kein* kritischer Abschnitt definiert, sondern lediglich *ein* Schreib- oder Lesevorgang atomar ausgeführt. Bereits das Postinkrement `counter++` besteht aus drei Einzelschritten:

```
final long temp = counter;
counter = counter + 1;
return temp;
```

Somit ist es möglich, dass andere Threads zu einem beliebigen Zeitpunkt der Ausführung der obigen Anweisungsfolge aktiviert werden. Dadurch ist keine Atomarität und keine Thread-Sicherheit mehr gegeben. Daher muss jede Folge von Anweisungen, die exklusiv durch einen Thread ausgeführt werden soll, als kritischer Abschnitt geschützt werden.

Tipp: Besonderheit Postinkrement

Dass das Postinkrement `counter++` aus drei Einzelschritten wie zuvor gezeigt besteht, können Sie am Rückgabewert der folgenden Methode nachvollziehen, die den Wert 10 und nicht den Wert 11 liefert, wie man es eventuell zunächst erwarten würde.

```
public static int postIncrementSurprise()
{
    int counter = 10;
    return counter++;
}
```

Atomare Variablen als Lösung?

Die Klassen `AtomicInteger` und `AtomicLong` ermöglichen atomare Read-Modify-Write-Sequenzen, wie `counter++`.[7] Die Atomic-Klassen nutzen eine sogenannte ***Compare-and-Swap-Operation*** (CAS). Die Besonderheit ist, dass diese zunächst einen Wert aus dem Speicher lesen und mit einem erwarteten Wert vergleichen, bevor sie eine Zuweisung mit einem übergebenen Wert durchführen. Dies geschieht allerdings nur, wenn ein erwarteter Wert gespeichert ist. Übertragen auf das vorherige Beispiel des Zählers würde dies unter Verwendung der Klasse `AtomicLong` wie folgt aussehen:

```
public long incrementAndGetUsingCAS()
{
    long oldValue = counter.get();
    // Unterbrechung möglich
    while (!atomicLongCounter.compareAndSet(oldValue, oldValue + 1))
    {
        // Unterbrechung möglich
        oldValue = atomicLongCounter.get();
        // Unterbrechung möglich
    }
    return oldValue + 1;
}
```

Wie man sieht, ist selbst das einfache Hochzählen eines Werts deutlich komplizierter als ein Einsatz von `synchronized`. Das liegt vor allem daran, dass in einer Schleife geprüft werden muss, ob das Setzen des neuen Werts korrekt erfolgt ist. Diese Komplexität entsteht dadurch, dass zwischen dem Lesen des Werts und dem Setzen ein anderer Thread den Wert verändert haben könnte. In einem solchen Fall wird die Schleife so lange wiederholt, bis eine Inkrementierung erfolgreich ist.

[7] Für Gleitkommatypen gibt es keine Unterstützung.

Auf einer solch niedrigen Abstraktionsebene möchte man in der Regel nicht arbeiten. Zur Vereinfachung werden die Details durch verschiedene Methoden der atomaren Klassen gekapselt: Ein sicheres Inkrementieren erfolgt beispielsweise mit der Methode `incrementAndGet()`. Weitere Details zu den atomaren Klassen finden Sie in den Büchern »Java Concurrency in Practice« von Brian Goetz [24] und »Java Threads« von Scott Oaks und Henry Wong [57].

Wie bei `volatile` gilt, dass sobald mehrere Attribute konsistent zueinander geändert werden sollen, eine Definition eines kritischen Abschnitts über `synchronized` oder Locks zwingend notwendig wird.

10.4.3 Reorderings

Die JVM darf gemäß der JLS zur Optimierung beliebige Anweisungen in ihrer Ausführungsreihenfolge umordnen, wenn dabei die Semantik des Programms nicht verändert wird. Bei der Entwicklung von Singlethreading-Anwendungen muss man den sogenannten ***Reorderings*** keine Beachtung schenken. Für Multithreading gibt es jedoch einige Dinge zu berücksichtigen. Betrachten wir zur Verdeutlichung eine Klasse `ReorderingExample` mit den Attributen `x1` und `x2` und folgenden Anweisungen:

```
public class ReorderingExample
{
    int x1 = 0;
    int x2 = 0;

    void method()
    {
        // Thread 1
        x1 = 1;                                                  // #1
        x2 = 2;                                                  // #2
        System.out.println("x1 = " + x1 + " / x2 = " + x2 );  // #3
    }
}
```

Werden die obigen Anweisungen ausgeführt, so kann der Compiler die Anweisungen #1 und #2 der Methode `method()` vertauschen, ohne dass dies Auswirkungen auf die nachfolgende Anweisung #3, hier die Ausgabe, hat. Anweisung #3 kann jedoch nicht mit Anweisung #1 oder #2 getauscht werden, da die Ausgabe lesend auf die zuvor geschriebenen Variablen zugreift. Reorderings sind demnach nur dann erlaubt, wenn sichergestellt werden kann, dass es keine Änderungen an den Wertzuweisungen der sequenziellen Abarbeitung der Befehle gibt. Es kommt also immer zu der folgenden Ausgabe: `x1 = 1 / x2 = 2`.

Für Multithreading wird die Situation komplizierter: Würde parallel zu den obigen Anweisungen in einem separaten Thread etwa folgende Methode `otherMethod()` dieser Klasse abgearbeitet, so kann es zu unerwarteten Ergebnissen kommen.

```
void otherMethod()
{
    // Thread 2
    int y1 = x2;                                          // #1
    int y2 = x1;                                          // #2
    System.out.println("x1 = " + x1 + " / x2 = " + x2 );  // #3
    System.out.println("y1 = " + y1 + " / y2 = " + y2 );  // #4
}
```

Betrachten wir zunächst den einfachen Fall ohne Reorderings. Wird Thread 1 vor Thread 2 ausgeführt, also `method()` komplett vor `otherMethod()` abgearbeitet, dann gilt offensichtlich $x1 = 1 = y2$ und $x2 = 2 = y1$. Dies ergibt sich daraus, dass alle Schreiboperationen in Thread 1 bereits ausgeführt wurden, bevor die Leseoperationen in Thread 2 durchgeführt werden. Werden die beiden Threads allerdings abwechselnd, beliebig ineinander verwoben, ausgeführt, so kann es zu merkwürdigen Ausgaben kommen. Denkbar ist etwa eine Situation, in der zwar wie vermutet $x1 = 1$ und $x2 = 2$ gilt, aber auch $y1 = 0$ und $y2 = 1$. Dies ist der Fall, wenn zunächst die Ausführung von Thread 1 (`method()`) begonnen und nach der Zuweisung $x1 = 1$ unterbrochen wird und dann Thread 2 ausgeführt wird, wie dies im Folgenden beispielhaft dargestellt ist:

```
int x1 = 0;
int x2 = 0;

// Thread 1
                                    // Thread 2
x1 = 1;
                                    y1 = x2 = 0;
                                    y2 = x1 = 1;
                                    // System.out: x1 = 1 / x2 = 0
                                    // System.out: y1 = 0 / y2 = 1
x2 = 2;
// System.out: x1 = 1 / x2 = 2
```

Wie man leicht sieht, kann es bereits beim abwechselnden Ausführen (***Interleaving***) von Threads zu Inkonsistenzen kommen. War obige Ausgabe noch mit etwas Nachdenken intuitiv verständlich, ist jedoch auch folgende, unerwartete Ausführungsreihenfolge möglich, wenn die Anweisungen umgeordnet werden:

```
int x1 = 0;
int x2 = 0;

// Thread 1
                                    // Thread 2
                                    y1 = x2 = 0;
x2 = 2;
                                    y2 = x1 = 0;
                                    // System.out: x1 = 0 / x2 = 2
                                    // System.out: y1 = 0 / y2 = 0
x1 = 1;
// System.out: x1 = 1 / x2 = 2
```

Man erkennt einige Besonderheiten beim Zusammenspiel von Threads und gemeinsamen Variablen: Es kann zu Inkonsistenzen durch Reorderings und Interleaving kommen. Für Singlethreading analysiert und ordnet die JVM die Folge von Schreib- und Lesezugriffen automatisch, sodass diese Probleme gar nicht erst entstehen können. Bei Multithreading ist die zeitliche Reihenfolge der Abarbeitung nicht im Vorhinein bekannt, sodass keine definierte Reihenfolge bezüglich des Schreibens und Lesens durch verschiedene Threads gegeben ist und die JVM keine Konsistenz sicherstellen kann. ***Die Folge ist, dass ein Programm zufällig (häufig sogar korrekte) Resultate liefert, aber im schlimmsten Fall unbrauchbar wird***. Bei Multithreading ist es daher Aufgabe des Entwicklers, der JVM Hinweise zu geben, in welcher Reihenfolge die Abarbeitungen erfolgen sollen bzw. welche Bereiche kritisch sind und zu welchen Zeitpunkten keine Reorderings stattfinden dürfen.

Für Multithreading ist dazu eine spezielle Ordnung ***Happens-before*** (hb) definiert, die für zwei Anweisungen A und B beliebiger Threads besagt, dass für B alle Änderungen von Variablen einer Anweisung A sichtbar sind, wenn $hb(A, B)$ gilt. Durch diese Ordnung wird indirekt festgelegt, in welchem Rahmen Reorderings stattfinden dürfen, da durch $hb(A, B)$ »Abstimmungspunkte« oder »Synchronisationspunkte« von Multithreading-Applikationen definiert werden. Zwischen diesen Abstimmungspunkten sind Reorderings allerdings möglich. An solchen Abstimmungspunkten im Programm weiß man dann jedoch sicher, dass alle Änderungen stattgefunden haben und diese für andere Threads sichtbar sind. Beispielsweise kann über Synchronisation eine gewisse Abarbeitungsreihenfolge erreicht werden: Zwei über dasselbe Lock synchronisierte kritische Abschnitte schließen sich gegenseitig aus. Laut $hb(A, B)$ gilt, dass nachdem einer von beiden abgearbeitet wurde, alle Modifikationen für den nachfolgenden sichtbar sind. Über die Reihenfolge der Ausführung innerhalb eines `synchronized`-Blocks kann allerdings keine Aussage getroffen werden. ***Besteht die Happens-before-Ordnung zwischen zwei Anweisungen jedoch nicht, so darf die JVM den Ablauf von Befehlen beliebig umordnen***.

Hintergrundinformationen zur Ordnung hb

Zum besseren Verständnis des Ablaufs bei Multithreading ist es hilfreich zu wissen, für welche Anweisungen eine Happens-before-Ordnung gilt:

- Zwei Anweisungen A und B erfüllen $hb(A, B)$, wenn B *nach* A im selben Thread in der sogenannten Program Order steht. Vereinfacht bedeutet dies, dass jeder Schreibzugriff für folgende Lesezugriffe auf ein Attribut sichtbar ist. Erfolgen Lese- und Schreibzugriffe nicht auf gleiche Attribute, stehen Anweisungen also nicht in einer Lese-Schreib-Beziehung, dürfen diese vom Compiler beliebig umgeordnet werden. Bei Singlethreading macht sich dies nicht bemerkbar, da trotz Reorderings die Bedeutung nicht verändert wird.

- Es gilt $hb(A, B)$, wenn zwei Anweisungen A und B über denselben Lock synchronisiert sind.
- Für einen Schreibzugriff A auf ein `volatile`-Attribut und einen späteren Lesezugriff B gilt $hb(A, B)$.
- Für den Aufruf von `start()` (A) eines Threads und alle Anweisungen B, die in diesem Thread ausgeführt werden, gilt $hb(A, B)$.
- Für alle Anweisungen A eines Threads vor dessen Ende B gilt $hb(A, B)$, d. h., alle nachfolgenden Threads können diese Änderungen sehen – aber nur, wenn diese über den gleichen Lock synchronisiert sind oder `volatile`-Attribute verwenden.

In Multithreading-Programmen muss man daher als Programmierer dafür sorgen, eine Happens-before-Ordnung herzustellen, um mögliche Fehler durch Reorderings zu verhindern. Auch ohne sämtliche Details dieser Ordnung zu verstehen, kann man Thread-sichere Programme schreiben, wenn man sich an folgende Grundregeln hält:

1. **Korrekte, minimale, aber vollständige Synchronisierung** – Greifen mehrere Threads auf ein gemeinsam benutztes Attribut zu, so müssen immer *alle* Zugriffe über *dasselbe* Lock-Objekt synchronisiert werden. Für semantische Einheiten von Attributen kann auch ein und dasselbe Lock-Objekt verwendet werden.
2. **Beachtung von Atomarität** – Zuweisungen erfolgen in der Regel atomar. Für die 64-Bit-Datentypen muss dies explizit über das Schlüsselwort `volatile` sichergestellt werden. Mehrschrittoperationen (beispielsweise `i++`) können so nicht geschützt werden und müssen zwingend synchronisiert werden.

Tipp: Eigenschaften der Ordnung hb

Die Ordnung hb besitzt für beliebige Anweisungen A und B folgende Eigenschaften:

- **Irreflexiv** – $!hb(A, A)$ – Keine Anweisung A »sieht« ihre eigenen Änderungen.
- **Transitiv** – Wenn $hb(A, B)$ und $hb(B, C) \Rightarrow hb(A, C)$ – Die Transitivität sorgt für die Sichtbarkeit gemäß der Reihenfolge der Anweisungen im Sourcecode.
- **Antisymmetrisch** – Wenn $hb(A, B)$ und $hb(B, A) \Rightarrow A = B$ oder hier intuitiver: Wenn $hb(A, B)$ und $A \neq B \Rightarrow !hb(B, A)$ – Die Antisymmetrie stellt sicher, dass eine Anweisung A nicht die Modifikationen einer Nachfolgeanweisung B »sieht«. Tatsächlich ist dies schon durch die Irreflexivität ausgeschlossen.

10.5 Besonderheiten bei Threads

Nachdem wir recht ausführlich den Lebenszyklus und die Zusammenarbeit von Threads kennengelernt haben, geht dieser Abschnitt auf Besonderheiten ein. Dazu stelle ich zunächst verschiedene Arten von Threads vor. Danach betrachten wir, wie sich Exceptions in Threads auswirken. Anschließend greife ich das Beenden von Threads erneut auf und zeige, wie dies sicher realisiert werden kann. Abschließend stelle ich Möglichkeiten zur zeitgesteuerten Ausführung von Aufgaben vor.

10.5.1 Verschiedene Arten von Threads

Bei der Arbeit mit Threads gibt es noch ein bisher nur am Rande erwähntes Detail zu beachten: In Java unterscheidet man zwischen User- und Daemon-Threads.

`main`-Thread und User-Threads

Wie bereits bekannt, erzeugt die JVM beim Start einen speziellen Thread, den man `main`-Thread nennt, weil dieser statt der `run()`-Methode die `main()`-Methode des Programms ausführt. Nehmen wir an, eine Applikation würde mehrere Threads aus dem `main`-Thread erzeugen und starten. Diese Threads nennt man ***User-Threads***. Die JVM bleibt selbst nach Ausführung des `main`-Threads bzw. der letzten Anweisung der `main()`-Methode aktiv, solange noch vom `main`-Thread abgespaltene User-Threads existieren und deren `run()`-Methode ausgeführt wird.

Daemon-Threads

Manchmal sollen Aufgaben im Hintergrund ablaufen und die Terminierung der JVM von solchen Threads unabhängig sein. Werden aber lediglich User-Threads gestartet, so wird die JVM nicht beendet, solange noch einer von diesen läuft. Als Abhilfe gibt es sogenannte ***Daemon-Threads***. Einen solchen erhält man, wenn man einen beliebigen Thread, d. h. ein `Thread`-Objekt, durch Aufruf der Methode `setDaemon(boolean)` *vor* seiner Ausführung zu einem Daemon-Thread umwandelt. Daemon-Threads sind praktisch, wenn die Hintergrundaufgaben nicht wirklich zur eigentlichen Programmfunktionalität beitragen. Das gängigste Beispiel ist der Garbage Collector, der im Hintergrund Speicher aufräumt (vgl. Abschnitt 12.4).

Den Unterschied zwischen User- und Daemon-Threads erkennt man, wenn ein Programm bzw. die zugehörige JVM terminieren soll. Bekanntermaßen geschieht dies nur, nachdem alle User-Threads beendet sind. Dann noch aktive Daemon-Threads werden abrupt beendet, d. h. irgendwo in der Abarbeitung ihrer `run()`-Methode. Daher muss man sorgsam sein, wenn Daemon-Threads Ressourcen belegen. Zu deren Freigabe kann man in einer eigenen, von `Thread`-abgeleiteten Klasse die Methode `finalize()` implementieren. Weil deren Abarbeitung laut JLS jedoch nicht garantiert ist, bietet sich ein Shut-down-Hook (vgl. Abschnitt 16.1.12) an.

10.5.2 Exceptions in Threads

In diesem Abschnitt wollen wir uns damit beschäftigen, was passiert, wenn während der Abarbeitung eines Threads eine Exceptions ausgelöst wird. Nehmen wir dazu an, eine Applikation würde mehrere Threads aus dem `main`-Thread starten und in irgendeinem dieser Threads würde eine Ausnahmesituation auftreten, die zu einer Exception führt. Aus Abschnitt 5.6.1 wissen wir, dass eine Exception mit einem `catch`-Block behandelt werden muss oder automatisch entlang der Methodenaufrufkette weitergereicht wird, bis ein passender `catch`-Block gefunden wird oder man schließlich die `run()`- bzw. `main()`-Methode erreicht.

Findet sich auch dort kein passender `catch`-Block, so bricht die Ausführung ab und dies führt zu einer Beendigung des ausführenden Threads. Abschließend wird die vom Programm unbehandelte Exception über den Error-Stream `System.err` inklusive des kompletten Stacktrace ausgegeben. Zum Nachvollziehen des zuvor beschriebenen Verhaltens können Sie folgendes Programm EXCEPTIONINTHREADSEXAMPLE ausführen:

```
// Achtung: Nur zur Demonstration des Exception Handlings
public static void main(final String[] args) throws InterruptedException
{
    exceptionInMethod();
}

static void exceptionInMethod() throws InterruptedException
{
    final Thread exceptional = new Thread()
    {
        public void run()
        {
            throw new IllegalStateException("run() failed");
        }
    };

    exceptional.start();
    Thread.sleep(1000);
}
```

Listing 10.9 *Ausführbar als* **'EXCEPTIONINTHREADSEXAMPLE'**

Es erscheint folgende Ausgabe (gekürzt) auf der Konsole:

```
Exception in thread "Thread-0" java.lang.IllegalStateException: run() failed
```

Wird das abrupte Ende eines Threads lediglich auf der Konsole ausgegeben, erschwert dies eine spätere Fehlersuche. Zum Nachvollziehen ist es nützlicher, Exceptions in eine Log-Datei zu schreiben. Dazu wollen wir eine kleine Utility-Klasse erstellen und greifen auf das mit Java 5 eingeführte innere Interface `UncaughtExceptionHandler` der Klasse `Thread` zurück. Dessen Implementierung erlaubt es, von `catch`-Blöcken unbehandelte Exceptions behandeln zu können, indem man in der Methode `uncaughtException(Thread, Throwable)` das gewünschte Verhalten realisiert.

Die folgende Klasse `LoggingUncaughtExceptionHandler` implementiert beispielsweise eine Ausgabe in eine Log-Datei. Im Listing ist eine `main()`-Methode gezeigt, die zu Demonstrationszwecken Exceptions provoziert:

```
public final class LoggingUncaughtExceptionHandler implements Thread.
      UncaughtExceptionHandler
{
   private static final Logger log = LogManager.getLogger("UncaughtExceptions");

   @Override
   public void uncaughtException(final Thread thread, final Throwable throwable)
   {
       log.error("Unexpected exception occured: ", throwable);
   }
}
```

In diesem Beispiel wird ein mögliches Problem zwar nicht weiter behandelt, aber zumindest in einer Log-Datei protokolliert, was eine spätere Analyse erleichtert.

Ein solcher `UncaughtExceptionHandler` kann für alle Threads durch Aufruf von `Thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler)` global gesetzt werden. Bei Bedarf kann dies für jeden Thread einzeln durch Aufruf von `setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler)` erfolgen.

10.5.3 Sicheres Beenden von Threads

Wie bereits erwähnt, lassen sich Threads leider nicht so einfach beenden wie starten. Die Methode `stop()` der Klasse `Thread` ist als `@deprecated` markiert. In verschiedenen Quellen wird als Grund dafür genannt, dass beim Beenden eines Threads nicht alle Locks freigegeben werden. Das ist ein verbreiteter Irrtum. Definitiv werden beim Auftreten von Exceptions alle gehaltenen Locks durch die JVM zurückgegeben. Wäre dem nicht so, wäre kein sinnvolles Programmverhalten mehr möglich: Alle Aufrufe von `synchronized`-Methoden, die über diese Locks geschützt werden, wären für die restliche Laufzeit der JVM blockiert.

Der Grund für die Markierung als `@deprecated` ist vielmehr folgender: Wenn ein Thread durch Aufruf von `stop()` beendet wird, können dadurch die Daten, auf denen der Thread gerade gearbeitet hat, in einen inkonsistenten Zustand gebracht werden. Insbesondere gilt dies, wenn der Thread innerhalb einer eigentlich atomaren Anweisungsfolge unterbrochen wird: Wird `stop()` durch einen anderen Thread mitten während der Ausführung eines über `synchronized` definierten kritischen Bereichs aufgerufen, so wird dieser irgendwo unterbrochen und nicht mehr atomar ausgeführt. Eine mögliche Inkonsistenz im Objektzustand ist die Folge.

Um Konsistenz zu wahren, müssen wir andere Wege finden, einen Thread korrekt zu beenden. Zwei Möglichkeiten, dies sauber zu lösen, sind die folgenden:

1. Einführen einer Hilfsklasse, die die Funktionalität bereitstellt
2. Beenden durch Aufruf der Methode `interrupt()`

Hilfsklasse zum Beenden

Eine mögliche Lösung zum Beenden von Threads besteht darin, eine Hilfsklasse mit einem Flag-Attribut `shouldStop` sowie korrespondierende Zugriffsmethoden zu implementieren und periodisch das Flag-Attribut abzufragen.

Betrachten wir zunächst eine denkbare, aber falsche Umsetzung, wie man sie in der Praxis und (leider auch) in manchem Buch findet:

```
// Achtung: Nicht Thread-sicher
public class BaseStoppableThread extends Thread
{
    private boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public void shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // Kein Aufruf von requestStop() und
            // keine Schreibzugriffe auf shouldStop
        }
    }
}
```

In der Praxis wird das Flag häufig `stopped` genannt. Zudem heißen die Zugriffsmethoden auf das Flag etwa `setStopped(boolean)` und `isStopped()`. Dies entspricht zwar der Intention des Beendens, allerdings wird hier eher ein Stoppwunsch geäußert. Daher werden die Methoden von mir `requestStop()` und `shouldStop()` genannt.

Auf den ersten Blick ist im Listing – abgesehen von der ungeschickten Ableitung von der Utility-Klasse `Thread` (vgl. folgenden Hinweis »Ableitung von der Klasse Thread«) – kein Fehler zu erkennen. Wieso ist diese Umsetzung trotzdem problematisch? Nach Lektüre von Abschnitt 10.4 über das Java-Memory-Modell sind wir bereits etwas sensibilisiert: Die JVM darf zur Optimierung Reorderings durchführen, sofern die Happens-before-Ordnung eingehalten wird. Ohne diese Ordnung beachtet der Compiler bei der Optimierung keine Multithreading-Aspekte: Er kann beispielsweise wiederholte Lesezugriffe auf sich nicht ändernde Variablen zu vermeiden versuchen.

Innerhalb der `run()`-Methode finden wir keinen Aufruf von `requestStop()` oder einen sonstigen Schreibzugriff auf das Attribut `shouldStop`. Für Singlethreading ergibt sich daraus, dass sich das Attribut `shouldStop` in der Schleife nicht mehr ändert. Damit ist das Ergebnis der Bedingung konstant. Um den Methodenaufruf und die wiederholte Auswertung der Bedingung `!shouldStop()` einzusparen, kann der Sourcecode durch den Compiler und die JVM wie folgt optimiert und umgewandelt werden:

```
public void run()
{
    if (!shouldStop())
    {
        while (true)
        {
            // ...
```

Beim Einsatz von Multithreading und Reorderings ist zum korrekten Ablauf dieses Beispiels die Happens-before-Ordnung sicherzustellen. Das kann in diesem Fall entweder über die Deklaration des Attributs `shouldStop` als `volatile` oder einen synchronisierten Zugriff geschehen. Bei der nachfolgenden Korrektur vermeiden wir außerdem, von der Klasse `Thread` abzuleiten, und implementieren eine Lösung, die auf dem Interface `Runnable` basiert und die Happens-before-Ordnung sicherstellt, wodurch keine Reorderings und keine Optimierung der Schleifenabfrage erfolgen:

```
abstract class AbstractStoppableRunnable implements Runnable
{
    private volatile boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public boolean shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // ...
        }
    }
}
```

Hinweis: Ableitung von der Klasse `Thread`

Ableitungen von Utility-Klassen sind in der Regel ungünstig. ***Leider sieht man aber genau dies häufig beim Einsatz der Klasse `Thread`***. Schauen wir uns an, wieso es zu Problemen kommen kann. Nehmen wir dazu an, eine Klasse `AbstractStoppableThread` sei durch Vererbung realisiert und die restliche Realisierung entspräche der zuvor vorgestellten Klasse `AbstractStoppableRunnable`:

```
abstract class AbstractStoppableThread extends Thread
{
    private volatile boolean shouldStop = false;

    // ...
}
```

Durch diese Implementierung wird das Substitutionsprinzip und demzufolge auch die »is-a«-Eigenschaft (vgl. Kapitel 3) verletzt. Der Grund ist folgender: Mit dieser von `Thread` abgeleiteten Klasse `AbstractStoppableThread` können keine Implementierungen von `Runnable` ausgeführt werden. Es entsteht eine Inkompatibilität: Die `run()`-Methode der Basisklasse `Thread` wird durch eine eigene Implementierung überschrieben, die keine Ausführung von `Runnable`s erlaubt. Die obige Umsetzung ist somit OO-technisch unsauber, weil sich die eigene Klasse nicht so verwenden lässt wie die Klasse `Thread`.

Beenden mit `interrupt()`

Statt einen Thread durch die Verwendung eigener Mechanismen, etwa den Einsatz eines Stop-Flags, zu beenden, kann man über die Methode `interrupt()` der Klasse `Thread` die Beendigung der Ausführung eines anderen Threads anregen. Wie bereits bekannt, ist ein Aufruf der Methode `interrupt()` jedoch nur als Aufforderung zu sehen, sie besitzt keine unterbrechende Wirkung: Es wird lediglich ein Flag gesetzt. Die Bearbeitung und Auswertung dieses Flags mithilfe der Methode `isInterrupted()` ist Aufgabe des Entwicklers der `run()`-Methode und kann in etwa so geschehen:

```
public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        // ...
    }
}
```

Achtung: `Thread.interrupted()` vs. `isInterrupted()`

Bei der Abfrage des Flags muss man etwas Vorsicht walten lassen. Die Klasse `Thread` bietet die statische Methode `interrupted()`, die zwar das Flag prüft, dieses allerdings auch zurücksetzt. In der Regel soll eine Prüfung ohne Seiteneffekt erfolgen. Dazu ist immer die Objektmethode `isInterrupted()` zu verwenden.

Fazit

Beide Lösungen zum Beenden von Threads sind funktional nahezu gleichwertig. Damit länger andauernde Aktionen in der `run()`-Methode tatsächlich abgebrochen werden können, müssen dort gegebenenfalls weitere Abfragen und Aufrufe von `shouldStop()` bzw. `isInterrupted()` erfolgen, um auch zwischen den ausgeführten Arbeitsschritten eine Reaktion auf Stoppwünsche zu ermöglichen. Das haben wir bereits in Abschnitt 10.1.4 diskutiert.

10.6 Weiterführende Literatur

Da die Themen Multithreading und Nebenläufigkeit recht komplex und vielschichtig sind, empfiehlt es sich, weitere Quellen zu konsultieren. Weiterführende Informationen finden Sie in folgenden Büchern:

- »**Parallele und verteilte Anwendungen in Java**« von Rainer Oechsle [58]
  Dieses Buch gibt einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5.
- »**Java Threads**« von Scott Oaks und Henry Wong [57]
  Ebenso wie das Buch von Rainer Oechsle bietet dieses Buch einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5. Durch den Fokus auf Threads erklärt es einige Dinge noch gründlicher.
- »**Concurrent Programming in Java**« von Doug Lea [48]
  Dieser Klassiker, der bereits einige Jahre auf dem Buckel hat, stammt vom Entwickler der Concurrency Utilities selbst. Viele der im Buch beschriebenen Klassen haben später Einzug in die Java-Klassenbibliothek gehalten. Es ist allerdings keine leichte Lektüre.
- »**Taming Java Threads**« von Allen Holub [30]
  Dieses Buch gibt einen fundierten Einstieg und beschreibt sehr genau die Details von Multithreading inklusive diverser Fallstricke. Es werden auch fortgeschrittenere Themen wie Multithreading und Swing, Thread-Pools und blockierende Warteschlangen behandelt. Da dieses Buch noch auf Java 1.4 basiert, werden hier ähnliche Ideen und Klassen entwickelt wie bei Doug Lea.

11 Modern Concurrency

Die bis einschließlich JDK 1.4 vorhandenen Sprachmittel, wie die Schlüsselwörter `volatile` und `synchronized` sowie die Methoden `wait()`, `notify()` und `notifyAll()`, erlauben es zwar, ein Programm auf mehrere Threads aufzuteilen und zu synchronisieren, allerdings lassen sich viele Aufgabenstellungen so nur relativ umständlich realisieren. Außerdem möchte kaum jemand heute noch auf Low-Level-Konzepten mit `Thread` und `Runnable` direkt programmieren. Im vorherigen Kapitel wurde deutlich, welche Fallstricke lauern und dass es anspruchsvoll ist, das korrekt umzusetzen. Zur besseren Auslastung von Multicore-Prozessoren ist es für Concurrency erstrebenswert, sich von den feingranularen und recht primitiven Ansätzen mit Threads zu lösen. Vielmehr bieten sich High-Level-Frameworks an. Doug Lea hat hier einen enormen Beitrag geleistet: Nicht nur das Executor- und das Fork-Join-Framework, sondern auch die von ihren Möglichkeiten her großartige Klasse `CompletableFuture<T>` sowie die mit JDK 9 bereitgestellten Reactive Streams gehen auf sein Konto.

Das gilt ebenfalls für die mit JDK 5 eingeführten Concurrency Utilities. Diese erleichtern die Entwicklung von Multithreading-Anwendungen, da sie für viele zuvor nur mühsam zu lösende Probleme bereits fertige Bausteine bereitstellen. Durch deren Einsatz wird Komplexität aus der Anwendung in das Framework verlagert, was die Lesbarkeit und Verständlichkeit des Applikationscodes deutlich erhöht: Für Locks und atomare Variablen haben wir bereits gesehen, dass sich damit Ideen und Konzepte klarer ausdrücken lassen.

Einführend werden wir in Abschnitt 11.1 die Concurrent Collections anschauen und ihre Vorteile in Multithreading-Umgebungen gegenüber normalen Collections beleuchten. Danach folgt in Abschnitt 11.2 eine Einführung in das Executor-Framework, das die Ausführung von Tasks mithilfe von Thread-Pools vereinfacht. Im Anschluss daran schauen wir in Abschnitt 11.3 kurz auf das Fork-Join-Framework, mit dessen Hilfe man rekursive Berechnungen nach dem Teile-und-herrsche-Prinzip parallel abarbeiten kann. Darüber hinaus wird die mit JDK 8 eingeführte Klasse `CompletableFuture<T>` in Abschnitt 11.4 vorgestellt. Damit lassen sich Verarbeitungsabläufe sehr elegant beschreiben und parallelisieren. Diese Klasse sollte jedem ambitionierten Java-Entwickler geläufig sein. Den Abschluss des Kapitels bildet eine Beschreibung zu Reactive Streams in Abschnitt 11.5. Reactive Streams ermöglichen Reactive Programming und die nicht blockierende Verarbeitung von Daten.

11.1 Concurrent Collections

Die Concurrent Collections enthalten auf Parallelität spezialisierte Implementierungen der Interfaces `List<E>`, `Set<E>`, `Map<K,V>`, `Queue<E>` und `Deque<E>`, wodurch ein einfaches Ersetzen »normaler« Datenstrukturen durch deren Concurrent-Pendants wesentlich erleichtert wird, falls Nebenläufigkeit erforderlich ist.

Wenn mehrere Threads parallel verändernd auf eine Datenstruktur zugreifen, kann es leicht zu Inkonsistenzen kommen. Dies gilt insbesondere, da die meisten Containerklassen des Collections-Frameworks nicht Thread-sicher sind. Die Zusammenarbeit und Kommunikation von Threads haben wir recht ausführlich in den Abschnitten 10.2 und 10.3 kennengelernt. Wir rekapitulieren in diesem Abschnitt zunächst kurz typische Probleme in Multithreading-Anwendungen beim Einsatz »normaler« Collections. Anschließend werden wir zur Lösung dieser Probleme die Concurrent Collections nutzen. Diese können bei Bedarf nach Parallelität stellvertretend für die »Original«-Container eingesetzt werden. Die Grundlage für diese Austauschbarkeit bilden gemeinsame Interfaces (z. B. `List<E>`, `Set<E>` und `Map<K,V>`).

Wenn tatsächlich Nebenläufigkeit und viele parallele Zugriffe unterstützt werden müssen, können die Concurrent Collections ihre Stärken ausspielen. In Singlethreading-Umgebungen oder bei sehr wenigen konkurrierenden Zugriffen ist ihr Einsatz gut abzuwägen, da in den Containern selbst einiges an Aufwand betrieben wird, um sowohl Thread-Sicherheit als auch Parallelität zu gewährleisten.

11.1.1 Thread-Sicherheit und Parallelität mit »normalen« Collections

Die `synchronized`-Wrapper des Collections-Frameworks ermöglichen einen Thread-sicheren Zugriff durch Synchronisierung aller Methoden (vgl. Abschnitt 6.3.2). Betrachten wir folgende synchronisierte Map als Ausgangsbasis unserer Diskussion:

```
final Map<String, Person> syncMap = Collections.synchronizedMap(personsMap);
```

Eine derartige Ummantelung kann zu einer (stark) eingeschränkten Parallelität führen, weil die Synchronisierung die Zugriffe serialisiert. Zudem schützt die Ummantelung nicht vor möglichen Inkonsistenzen, wenn man mehrere für sich Thread-sichere Methoden hintereinander aufruft. Das habe ich bereits bei der Beschreibung der `synchronized`-Wrapper in Abschnitt 6.3.2 diskutiert und greife es hier kurz auf, um die Arbeitsweise und Vorteile der Concurrent Collections zu motivieren.

Datenzugriff Jeder einzelne Methodenaufruf einer synchronisierten Collection ist für sich gesehen Thread-sicher. Für eine benutzende Komponente sind solche feingranularen Sperren aber häufig uninteressant. Vielmehr sollen Operationen mit mehreren Schritten atomar und Thread-sicher ausgeführt werden. Solche Mehrschrittoperationen sind etwa »`testAndGet()`« oder »`putIfAbsent()`«, die zunächst prüfen, ob

ein gewisses Element enthalten ist, und nur dann einen Zugriff bzw. eine Modifikation ausführen. Funktional würde man Folgendes schreiben:

```
// ACHTUNG: nicht Thread-sicher
public Person putIfAbsent(final String key, final Person newPerson)
{
    if (!syncMap.containsKey(key))
    {
        return syncMap.put(key, newPerson);
    }
    return syncMap.get(key);
}
```

Abschnitt 10.2.2 beschreibt, dass für sich Thread-sichere Methoden in ihrer Kombination nicht Thread-sicher sind. Abhilfe schafft eine Synchronisierung der gesamten Aktionen. Mithilfe eines Synchronisationsobjekts implementiert man eine korrekt synchronisierte Version der `putIfAbsent(String, Person)`-Methode etwa wie folgt:

```
public Person putIfAbsent(final String key, final Person newPerson)
{
    synchronized (syncMap)  // Kritischer Bereich für Mehrschrittoperationen
    {
        if (!syncMap.containsKey(key))
        {
            return syncMap.put(key, newPerson);
        }
        return syncMap.get(key);
    }
}
```

Durch eine solche Umsetzung wird allerdings keine gute Nebenläufigkeit erreicht, da andere Zugriffe blockiert werden.

Iteration Eine weitere sehr gebräuchliche Mehrschrittoperation ist das Iterieren über eine Datenstruktur. Die Iteratoren der »normalen« Container sind »fail-fast«, d. h., sie prüfen sehr streng, ob möglicherweise während einer Iteration eine Veränderung an der Datenstruktur vorgenommen wurde, und reagieren darauf mit einer `ConcurrentModificationException` (vgl. Abschnitt 6.1.4).

Zur Thread-sicheren Iteration über die Einträge einer synchronisierten Liste ist dieser Vorgang zusätzlich durch einen `synchronized`-Block zu schützen:

```
// Blockiert andere Zugriffe auf syncPersons während der Iteration
synchronized (syncPersons)
{
    for (final Person person : syncPersons)
    {
        person.doSomething();
    }
}
```

Auf diese Weise ist zwar eine korrekt synchronisierte Iteration möglich, allerdings auf Kosten von Nebenläufigkeit: Durch den `synchronized`-Block wird ein kritischer Bereich definiert und ***die Liste `syncPersons` ist während des Iterierens für mögliche andere Zugriffe blockiert. Nebenläufigkeit wird dadurch stark behindert***.

11.1.2 Parallelität mit den Concurrent Collections

Wenn mehrere Threads gemeinsam lesend auf einer Collection arbeiten, muss nicht immer die gesamte Datenstruktur gesperrt und dadurch der Zugriff für andere Threads blockiert werden. Mehr noch: Lesezugriffe sollten sich gegenseitig nicht beeinflussen und nicht blockieren, wohingegen Lese- und Schreibzugriffe aufeinander abgestimmt werden sollten. Dazu existieren verschiedene Verfahren. In den Concurrent Collections werden die zwei aufgelisteten Techniken eingesetzt, um neben Thread-Sicherheit auch für mehr Parallelität als bei den `synchronized`-Wrappern zu sorgen:

- **Kopieren beim Schreiben** – Die dahinterliegende Idee ist, vor jedem Schreibzugriff die Datenstruktur zu kopieren und dann das Element hinzuzufügen, zu verändern oder zu entfernen. Andere Threads können dadurch lesend zugreifen, ohne durch die Modifikation gestört zu werden. Diese Variante realisieren die Klassen `CopyOnWriteArrayList<E>` und `CopyOnWriteArraySet<E>` für Listen sowie Sets.
- **Lock-Striping / Lock-Splitting** – Hierbei werden verschiedene Teile eines Objekts oder einer Datenstruktur mithilfe mehrerer Locks geschützt. Dadurch sinkt die Wahrscheinlichkeit für gleichzeitige Zugriffe auf jeden einzelnen Lock und Threads werden seltener durch das Warten auf Locks am Weiterarbeiten gehindert. Als Folge davon steigt die Möglichkeit zur Parallelisierung. Zur Realisierung der parallelen `ConcurrentHashMap<K,V>` wird genau dieses Verfahren angewendet, das auf die spezielle Arbeitsweise von Hashcontainern mit Buckets abgestimmt ist und statt der gesamten `HashMap<K,V>` nur jeweils Teile davon schützt.

Datenzugriff in den Klassen `CopyOnWriteArrayList<E>/-Set<E>` Die Implementierung der Klasse `CopyOnWriteArraySet<E>` nutzt die Klasse `CopyOnWriteArrayList<E>`. Diese wiederum verwendet ein Array zur Speicherung von Elementen und passt dieses bei Schreibvorgängen ähnlich der in Abschnitt 6.1.2 beschriebenen Größenänderung von Arrays an. Jede Änderung der Daten erzeugt eine Kopie des zugrunde liegenden Arrays. Dadurch können Threads ungestört parallel lesen, sehen eventuell jedoch nicht die aktuellen Änderungen durch andere Threads.

Allerdings sollte man folgende zwei Dinge beim Einsatz bedenken: Für kleinere Datenmengen (< 1.000 Elemente) ist das Kopieren und die mehrfache Datenhaltung in der Regel vernachlässigbar. Der negative Einfluss steigt mit der Anzahl zu speichernder Elemente linear an. Aufgrund der gewählten Strategie des Kopierens bieten sich diese Datenstrukturen daher vor allem dann an, wenn deutlich mehr Lese- als Schreibzugriffe erfolgen und die zu speichernden Datenvolumina nicht zu groß sind.

Datenzugriff in der Klasse `ConcurrentHashMap<K,V>` Betrachten wir die Klasse `ConcurrentHashMap<K,V>`, die eine Realisierung einer `Map<K,V>` ist und mehreren Threads paralleles Lesen und Schreiben ermöglicht. Die Realisierung garantiert, dass sich Lesezugriffe nicht gegenseitig blockieren. Für Schreibzugriffe kann man zum Konstruktionszeitpunkt bestimmen, wie viel Nebenläufigkeit unterstützt werden soll, indem man die Anzahl der Schreibsperren festlegt.

Streng genommen ist die Klasse `ConcurrentHashMap<K,V>` lediglich ein Ersatz für die Klasse `Hashtable<K,V>` und nicht für die Klasse `HashMap<K,V>`, wie es der Name andeutet. Das liegt daran, dass die Klasse `ConcurrentHashMap<K,V>` im Gegensatz zur Klasse `HashMap<K,V>` keine `null`-Werte für Schlüssel und Werte unterstützt. Der Wert `null` drückt stattdessen aus, dass ein gesuchter Eintrag fehlt. In vielen Fällen wird diese Unterstützung für `null` nicht benötigt. Dann kann man die Klasse `ConcurrentHashMap<K,V>` problemlos anstelle einer `HashMap<K,V>` verwenden.

Das Interface `ConcurrentMap<K,V>` deklariert vier Mehrschrittoperationen. Das Besondere daran ist, dass diese von den konkreten Realisierungen `ConcurrentHashMap<K,V>` und `ConcurrentSkipListMap<K,V>` atomar ausgeführt werden müssen. Die folgende Aufzählung nennt die Methoden und zeigt zur Verdeutlichung der Funktionalität eine schematische Pseudoimplementierung. Die tatsächliche Implementierung ist wesentlich komplizierter, da sie Parallelität ohne Blockierung gewährleistet.

- `V putIfAbsent(K key, V value)` – Erzeugt einen neuen Eintrag zu diesem Schlüssel und dem übergebenen Wert, sofern kein Eintrag zu dem Schlüssel existiert. Liefert den zuvor gespeicherten Wert zurück, falls bereits ein Eintrag zu dem Schlüssel vorhanden ist, oder `null`, wenn dies nicht der Fall ist.

```
if (!map.containsKey(key))
{
    return map.put(key, value);
}
return map.get(key);
```

- `boolean remove(Object key, Object value)` – Entfernt den Eintrag zu diesem Schlüssel, falls der gespeicherte Wert mit dem übergebenen Wert übereinstimmt. Liefert `true`, falls es einen solchen Eintrag gab, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(value))
    {
        map.remove(key);
        return true;
    }
}
return false;
```

Tipp: Wieso der Typ `Object` statt generischer Typparameter?

Vielleicht fragen Sie sich, wieso hier `Object` statt der generischen Typen `K` und `V` genutzt wird. Das liegt zum einen daran, dass die Methoden in den Concurrent Collections diejenigen aus den Originalcontainerklassen überschreiben und somit die gleichen Typparameter besitzen müssen. Im Collections-Framework sind die `remove()`-Methoden mit `Object` definiert, weil die Prüfungen mit `equals()` auch mit `Object` arbeiten und man somit mehr Flexibilität beim Löschen hat.

- `V replace(K key, V value)` – Ersetzt zu diesem Schlüssel den gespeicherten durch den übergebenen Wert. Liefert den zuvor mit dem Schlüssel assoziierten Wert zurück oder `null`, wenn es keinen Eintrag zu dem Schlüssel gab:

```
if (map.containsKey(key))
{
    return map.put(key, value);
}
return null;
```

- `boolean replace(K key, V oldValue, V newValue)` – Ersetzt den Eintrag zu diesem Schlüssel mit dem neuen Wert `newValue`, falls der gespeicherte Wert mit dem alten Wert `oldValue` übereinstimmt. Liefert `true`, falls dem so ist, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(oldValue))
    {
        map.put(key, newValue);
        return true;
    }
}
return false;
```

Die gezeigten Aufrufe von `map.containsKey(key)` sind notwendig, um zu verhindern, dass bei einem fehlenden Eintrag eine `NullPointerException` ausgelöst wird.

Iteration Um eine parallele Verarbeitung zu unterstützen, wurde das Verhalten der Iteratoren so angepasst, dass diese weder `ConcurrentModificationException`s auslösen noch eine Synchronisierung benötigen. Allerdings sind die Iteratoren auch nur »schwach« konsistent oder »weakly consistent«, d. h., eine Iteration liefert nicht immer die aktuelle Zusammensetzung der Datenstruktur, aber zumindest den Zustand zum Zeitpunkt der Erzeugung des Iterators. Nachfolgende Änderungen an der Zusammensetzung können bei der Iteration berücksichtigt werden, müssen es aber nicht. Das erlaubt es, Iterationsvorgänge parallel zu Veränderungen durchzuführen.

11.1.3 Blockierende Warteschlangen und das Interface `BlockingQueue<E>`

Zum Austausch von Daten zwischen Programmkomponenten können Implementierungen des Interface `Queue<E>` (vgl. Abschnitt 6.6) verwendet werden. Das Interface `BlockingQueue<E>` erweitert das Basisinterface `Queue<E>` um folgende Methoden:

- `boolean offer(E element, long time, TimeUnit unit)` – Fügt ein Element in die Queue ein, falls keine Größenbeschränkung existiert oder die Queue nicht voll ist. Ansonsten wartet der Aufruf blockierend maximal die angegebene Zeitspanne, bis eine andere Programmkomponente ein Element entfernt, sodass als Folge ein Einfügen möglich wird. Liefert `true`, wenn das Einfügen erfolgreich war, ansonsten `false`.
- `E poll(long time, TimeUnit unit)` – Gibt das erste Element zurück und entfernt es aus der Queue. Wenn die Queue leer ist, wird maximal die angegebene Zeitspanne auf das Einfügen eines Elements durch eine andere Programmkomponente gewartet und nach einem erfolglosen Warten `null` zurückgegeben.
- `void put(E element)` – Fügt ein Element in die Queue ein. Dieser Aufruf erfolgt blockierend. Das bedeutet, dass gewartet werden muss, wenn die Queue eine Größenbeschränkung besitzt und zum Zeitpunkt des Einfügens voll ist.
- `E take()` – Gibt das erste Element zurück und entfernt es aus der Queue. Der Aufruf blockiert, solange die Queue leer ist.

Wie schon angedeutet, blockieren diese Methoden beim Schreiben in eine volle Queue bzw. beim Lesen aus einer leeren Queue.[1] Das erleichtert die Kommunikation zwischen Threads, da auf eine fehleranfällige Synchronisierung und Benachrichtigung über die Methoden `wait()` und `notify()` bzw. `notifyAll()` verzichtet werden kann.

Die durch das Interface `BlockingQueue<E>` beschriebene Schnittstelle wurde bereits in Abschnitt 10.3 zur Abstimmung von Producer und Consumer genutzt. Ähnliche Funktionalität wird durch folgende Klassen der Concurrent Collections realisiert:

- `ArrayBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff, besitzt eine Größenbeschränkung und verwendet ein Array zur Speicherung.
- `LinkedBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff und nutzt eine `LinkedList<E>`, wodurch keine Größenbeschränkung gegeben ist.
- `PriorityBlockingQueue<E>` – Diese Realisierung nutzt ein Sortierkriterium, um die Reihenfolge der Elemente innerhalb der Queue zu bestimmen. Elemente mit der höchsten Priorität stehen am Anfang und werden somit bei Lesezugriffen zuerst zurückgeliefert.
- `SynchronousQueue<E>` – Diese Queue ist ein Spezialfall einer größenbeschränkten Queue, allerdings mit der Größe 0. Zunächst klingt dies unsinnig, ist aber für solche Anwendungsfälle geeignet, die erfordern, dass zwei Threads unmittelbar

[1] Durch das Interface kann dies nur gefordert, nicht aber sichergestellt werden.

aufeinander warten. Bezogen auf das Producer-Consumer-Beispiel bedeutet dies, dass ein Producer erst sein Produkt »speichern« kann, wenn ein Consumer dieses direkt abholt. Andersherum gilt: Möchte ein Consumer Daten aus der Queue entnehmen, muss er warten, bis ein Producer Daten ablegt.[2]

- `DelayQueue<E>` – Bei dieser Art von Queue beschreiben zu speichernde Elemente ihre Sortierung, indem sie das Interface `java.util.concurrent.Delayed` implementieren. Das Interface `Delayed` erweitert `Comparable<Delayed>` und beschreibt somit eine Ordnung: In einer solchen Queue sind die Elemente nach ihrem »Verfallsdatum« geordnet. Dazu kann ein Ablaufzeitpunkt über die Methode `getDelay(java.util.concurrent.TimeUnit unit)` angegeben werden.

Beispiel

Entwickeln wir ein minimales Beispiel mit einer auf vier Einträge größenbeschränkten `ArrayBlockingQueue<Integer>`. Zunächst produzieren wir ein paar mehr als die dort vorgesehenen Einträge, wodurch der Producer nach Erreichen des Limits ein klein wenig warten muss, bis der Consumer die Daten abholt. Danach verlangsamt der Producer seine Arbeit und der Consumer muss dann immer 2 Sekunden warten, bis der nächste Wert produziert ist und abgeholt werden kann. Die Erleichterung im Vergleich zu den Beispielen aus dem vorherigen Kapitel besteht darin, dass einfacher auf die Queue zugegriffen werden kann, ohne dies speziell synchronisieren zu müssen:

```
public class BlockingQueueExample
{
    public static void main(final String[] args) throws Exception
    {
        final BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(4);

        new Thread(new Producer(queue)).start();
        final Thread consumerThread = new Thread(new Consumer(queue));
        consumerThread.start();

        Thread.sleep(7000);
        consumerThread.interrupt();
    }

    static class Producer implements Runnable
    {
        private final BlockingQueue<Integer> queue = null;

        public Producer(final BlockingQueue<Integer> queue)
        {
            this.queue = queue;
        }

        public void run()
        {
            // erste 4 gehen direkt, dann warten auf Consumer
            for (int i = 0; i < 8; i++)
            {
```

[2] Man spricht auch von Rendezvous, was Treffen in Französisch bedeutet. Für Multithreading meint dies, dass sich zwei Threads zum Datenaustausch »treffen«.

```
                try
                {
                    System.out.println("putting " + i);
                    queue.add(i);
                }
                catch (Exception e1)
                {
                    e1.printStackTrace();
                }
            }
            // Consumer muss immer 2 Sekunden warten, bis Erzeugung fertig
            try
            {
                queue.put(1111);
                Thread.sleep(2000);
                queue.put(2222);
                Thread.sleep(2000);
                queue.put(3333);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }

    static class Consumer implements Runnable
    {
        private final BlockingQueue<Integer> queue;

        public Consumer(final BlockingQueue<Integer> queue)
        {
            this.queue = queue;
        }

        public void run()
        {
            while (!Thread.currentThread().isInterrupted())
            {
                try
                {
                    System.out.println("Consuming " + queue.take());
                    Thread.sleep(100);
                }
                catch (InterruptedException e)
                {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
}
```

Nutzt man im Producer `add()` statt `put()`, so kann man die Größenbeschränkung sichtbar machen: Bei Überschreiten der Größe wird eine Exception ausgelöst:

```
putting 0
putting 1
putting 2
putting 3
putting 4
java.lang.IllegalStateException: Queue full
```

11.2 Das Executor-Framework

Das Executor-Framework unterstützt bei der Ausführung asynchroner Aufgaben durch Bereitstellung von Thread-Pools und ermöglicht es, verschiedene Abarbeitungsstrategien zu verwenden sowie die Bearbeitung abzubrechen und Ergebnisse abzufragen.

Das Executor-Framework vereinfacht den Umgang mit Threads und die Verarbeitung von Ergebnissen asynchroner Aufgaben, sogenannter ***Tasks***. Es erfolgt eine Trennung der Beschreibung eines Tasks und dessen tatsächlicher Ausführung. Dabei wird von den Details abstrahiert, etwa wie und wann eine Aufgabe von welchem Thread abgearbeitet wird. Die Kernidee ist, Tasks an sogenannte Executoren zur (späteren) Ausführung zu übergeben.

11.2.1 Einführung

Ohne das Executor-Framework können Tasks (Aufgaben bzw. Arbeitsblöcke), die das Interface `Runnable` implementieren, mithilfe der Klasse `Thread` ausgeführt werden. Bekanntermaßen schreibt man dazu Folgendes:

```
new Thread(runnableTask).start();
```

Das Interface `Executor`

Statt explizit mit Threads zu arbeiten, kann man alternativ eine Realisierung des Interface `java.util.concurrent.Executor` nutzen, um einen Task auszuführen. Dadurch wird von den konkreten Aktionen »Erzeugen« und »Starten« abstrahiert. Das Interface `Executor` ist im JDK wie folgt definiert:

```
public interface Executor
{
    void execute(final Runnable runnableTask);
}
```

Damit lässt sich die obige Ausführung folgendermaßen schreiben:

```
executor.execute(runnableTask);
```

Sollen nur einige wenige Tasks abgearbeitet werden, so ergibt sich kaum ein Vorteil durch den Einsatz eines Executors gegenüber dem Einsatz von Threads. Für Multithreading-Anwendungen, die aus diversen (unabhängigen) Tasks bestehen, kann eine derartige Abstraktion allerdings sehr vorteilhaft sein, wie wir dies nun beleuchten.

Implementierung des Interface `Executor` Die zuvor gezeigte Abstraktion von Threads und Implementierungsdetails wirkt zunächst wenig spektakulär. Warum das Ganze dennoch hilfreich ist, werden wir sukzessive kennenlernen. Ein erster Vorteil besteht darin, dass das Interface `Executor` keine konkrete Art der Ausführung vorgibt.

Die Details werden durch die jeweilige `Executor`-Implementierung spezifiziert, beispielsweise wie viele Aufgaben parallel ausgeführt werden und wann bzw. in welcher Reihenfolge dies erfolgt.

Betrachten wir zunächst zwei extreme Arten der Ausführung: alle Tasks synchron hintereinander und alle Tasks vollständig asynchron. Für Ersteres könnte man eine Klasse `SynchronousExecutor` wie folgt implementieren:

```
public class SynchronousExecutor implements Executor
{
    public void execute(final Runnable runnable)
    {
        runnable.run();
    }
}
```

In der Regel sollen Tasks nicht synchron im aufrufenden Thread, sondern parallel dazu ausgeführt werden. Eine Klasse `AsyncExecutor`, die für jeden Task einen eigenen Thread startet, könnte folgendermaßen realisiert werden:

```
public class AsyncExecutor implements Executor
{
    public void execute(final Runnable runnableTask)
    {
        new Thread(runnable).start();
    }
}
```

Hinweis: `Executor` selbst implementieren?

Wie bereits gesehen, ist es zwar möglich, eigene Implementierungen des Interface `Executor` zu schreiben, normalerweise empfiehlt es sich jedoch, die vordefinierten Klassen des Executor-Frameworks zu verwenden, die wir im Verlauf dieses Abschnitts kennenlernen werden.

Motivation für Thread-Pools

Nehmen wir an, es seien viele (möglichst voneinander unabhängige Aufgaben zu erledigen (konsultieren Sie bitte den folgenden Praxistipp für Hintergrundinformationen dazu). Die zuvor vorgestellten Extreme bei der Umsetzung, d. h. die streng sequenzielle Ausführung aller Aufgaben durch einen einzigen Thread bzw. die maximale Parallelisierung durch Abspalten eines eigenen Threads pro Aufgabe, besitzen beide unterschiedliche Nachteile. Der erste Ansatz führt durch die Hintereinanderausführung zu langen Wartezeiten und schlechten Antwortzeiten. Es kommt zu einem geringeren Durchsatz. Der zweite Ansatz parallelisiert die Ausführung und erreicht auf diese Weise einen höherem Durchsatz, sodass man etwas naiv auf die Idee kommen könnte, einfach so viele Threads zu erzeugen, wie Aufgaben existieren, um dadurch die Parallelität der Abarbeitung zu maximieren und für gute Antwortzeiten zu sorgen. Allerdings skaliert dieses Vorgehen nur begrenzt: Zunächst einmal gibt es doch mitunter Abhängigkeiten

zwischen den Threads. Vor allem gilt aber: Steigt die Anzahl der genutzten Threads deutlich über die Anzahl der verfügbaren CPUs bzw. Rechenkerne, so sinkt der Leistungszuwachs. Dies liegt daran, dass maximal so viele Threads gleichzeitig aktiv sein können, wie es CPUs bzw. Rechenkerne gibt. Zudem steigt der Aufwand zur Verwaltung und Abstimmung der Threads und für Thread-Wechsel.

Weiterhin ist die Ausführung von Aufgaben durch Threads mit einem gewissen Overhead verbunden: Zum einen kostet das Erzeugen und Starten eines Threads deutlich mehr Rechenzeit als ein Methodenaufruf. Zum anderen belegen Threads sowohl Betriebssystemressourcen als auch Speicher innerhalb sowie außerhalb der JVM. Sie können daher nur in begrenzter Anzahl[3] erzeugt werden.

Einen sinnvollen Kompromiss erreicht man durch den Einsatz sogenannter ***Thread-Pools***. Die Idee ist dabei, eine bestimmte Anzahl lauffähiger, aber pausierter Threads vorrätig zu halten und bei Bedarf zur Ausführung von Tasks zu aktivieren. Dadurch spart man sich die aufwendige Konstruktion neuer Threads. Thread-Pools ermöglichen zudem ein klares Programmdesign, da Verwaltungsaufgaben, d. h. das gesamte Thread-Management (Erzeugung, Zuteilung, Fehlerbehandlung usw.), aus der Applikation in den Thread-Pool verlagert wird. Ein solcher Thread-Pool kann ebenfalls dazu dienen, bei sehr hoher Last momentan nicht verarbeitbare Anfragen zu speichern und im Nachhinein zu bearbeiten. Alternativ kann man mit einem Thread-Pool beim Auftreten von Belastungsspitzen dynamisch versuchen, den Durchsatz zu erhöhen, indem temporär die Anzahl der Bearbeitungs-Threads (auch Worker-Threads genannt) erhöht wird.

Achtung: Nachteile von Thread-Pools

Thread-Pools bieten sich dann an, wenn die auszuführenden Tasks möglichst unabhängig voneinander sind. Doch selbst dabei kann eine ungünstige Mischung von langlaufenden und kurzen Tasks zu Problemen führen, wenn die »Langläufer« sich vor den »Kurzläufern« registriert haben und letztere so für längere Zeit und damit eventuell unerwartet stark verzögert werden.

Wenn übergebene Tasks voneinander abhängig sind, kann der Einsatz eines Thread-Pools ungeeignet sein, da keine Abarbeitungsreihenfolge garantiert ist. Im Extremfall würde ein auszuführender Task vor einem logisch davor auszuführenden Task einsortiert werden. Damit würde der Nachfolger vergeblich auf das Ergebnis eines noch nicht durch den Thread-Pool ausgeführten Vorgänger-Tasks warten. Dadurch ist keine weitere Abarbeitung dieses Tasks mehr möglich und der Thread-Pool würde »verstopft«.

Für Aufgaben, die sich gut per Rekursion in Teilaufgaben untergliedern lassen, kann man statt auf Thread-Pools des Executor-Frameworks auf das Fork-Join-Framework (vgl. Abschnitt 11.3) zurückgreifen. Sind eher Verarbeitungsketten mit einigen mitunter parallel, teilweise auch unabhängig auszuführenden Berechnungen erforderlich, so bietet sich dafür die Klasse `CompletableFuture<T>` an (vgl. Abschnitt 11.4).

[3] Diese ist abhängig von der verwendeten Stackgröße und dem zur Verfügung stehenden Hauptspeicher.

11.2.2 Definition von Aufgaben

Bevor wir uns mit der Ausführung von Aufgaben mithilfe von Thread-Pools des Executor-Frameworks beschäftigen, schauen wir uns die Definition von Aufgaben und ein paar Herausforderungen dabei an: Bei der Abarbeitung nebenläufiger Tasks gibt es gewisse Anforderungen, die mithilfe von `Runnable`s nur schwierig oder umständlich zu realisieren sind. Nachdem die Beschränkungen vorgestellt wurden, betrachten wir als Abhilfe das Interface `java.util.concurrent.Callable<V>`.

Definition von Aufgaben mit `Runnable`

Um die Ergebnisse nebenläufiger Berechnungen wieder an den Thread kommunizieren zu können, der die Ausführung gestartet hat, muss man sich beim Einsatz von `Runnable`s gewisser Tricks bedienen. Ursache dafür ist, dass die `run()`-Methode weder einen Ergebniswert zurückgeben noch eine Checked Exception auslösen kann. Ein Austausch von Ergebnissen zwischen einem Aufrufer und dem Bearbeitungs-Thread ist daher lediglich dadurch möglich, dass diese in gemeinsam benutzten Datenstrukturen abgelegt werden. Dazu kann man in eigenen Realisierungen von `Runnable`s spezielle Attribute zur Speicherung von Rückgabewerten definieren, die nach Abschluss der Berechnung von Aufrufern ausgelesen werden können.

IDIOM: EIN- UND AUSGABE MIT RUNNABLE Die Methode `run()` aus dem Interface `Runnable` ist für Berechnungen ohne Abhängigkeiten recht gut geeignet. Des Öfteren benötigt man allerdings Ein- und Ausgaben und eine Möglichkeit, wie man das `Runnable` mit Informationen versorgen bzw. parametrieren und nach der Berechnung auch wieder auf ein berechnetes Ergebnis zugreifen kann. Das Ganze wird nun an einem Beispiel verdeutlicht. Zunächst implementieren wir das Interface `Runnable` in Form der Klasse `InputOutputRunnableExample` und bieten einen Konstruktor mit Eingabeparametern, um Informationen für die Berechnung zu übergeben. Ein Aufrufer kann auf das Berechnungsergebnis über die Methode `getResult()` zugreifen. Ob ein solches bereits vorliegt, ermittelt man durch Abfrage von `isCalculationFinished()`. Falls dieses Flag noch `false` ist, liegt noch kein Berechnungsergebnis vor und `getResult()` liefert `null` als Rückgabe:

```
public class InputOutputRunnableExample<T1, T2, V> implements Runnable
{
    private final T1 input1;
    private final T2 input2;

    // Zuweisungen sind zwar redundant, helfen aber,
    // die Wertebelegung klar darzulegen
    private volatile V resultValue = null;
    private volatile boolean calculationFinished = false;

    public InputOutputRunnableExample(final T1 input1, final T2 input2)
    {
        this.input1 = input1;
        this.input2 = input2;
    }
```

```
    public void run()
    {
        // Rücksetzen, falls mehrmals run() aufgerufen wird
        calculationFinished = false;

        resultValue = calculateResult(input1, input2);

        calculationFinished = true;
    }

    public V getResult()
    {
        return resultValue;
    }

    public boolean isCalculationFinished()
    {
        return calculationFinished;
    }

    private V calculateResult(final T1 input1, final T2 input2)
    {
        // nur zur Demonstration, in der Praxis basierend auf
        // input1 und input2 berechnetes Ergebnis
        return null;
    }
}
```

Der hier erforderliche recht hohe Implementierungsaufwand ist durch die Beschränkungen des Interface `Runnable` begründet. Es ist für Aufrufer darüber hinaus nicht immer einfach, ohne Polling-Aufruf von `isCalculationFinished()` das Ende einer Berechnung zu erkennen.

Definition von Aufgaben mit `Callable<V>`

Aufgrund dieser Einschränkungen bei der Ausführung nebenläufiger Aktivitäten und deren Überwachung wurde das Interface `Callable<V>` eingeführt, womit man einem Aufrufer einen Rückgabewert übermitteln und außergewöhnliche Situationen über Exceptions kommunizieren kann. Dieses Interface ist im JDK folgendermaßen definiert:

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

Betrachten wir nun, wie sich das zuvor dargestellte Idiom zum Datenaustausch mit einem Aufrufer durch den Einsatz eines `Callable<V>` klarer gestalten lässt. Auch hier haben wir zwei Eingabeparameter und berechnen zu Demonstrationszwecken lediglich eine Wartezeit in Millisekunden, für die wir dann auch die Ausführung pausieren lassen. Das generische Interface `Callable<V>` erlaubt beliebige nicht primitive Rückgabetypen. Für dieses einfache Beispiel nutzen wir den Typ `Long`, wodurch diese Realisierung kürzer als diejenige mit `Runnable` ist und keine Hilfsvariable `result` zur Zwischenspeicherung der Berechnungsergebnisse benötigt:

```
public class CalcDurationInMs implements Callable<Long>
{
    private final TimeUnit timeUnit;
    private final long     duration;

    CalcDurationInMs(final TimeUnit timeUnit, final long duration)
    {
        this.timeUnit = timeUnit;
        this.duration = duration;
    }

    @Override
    public Long call() throws Exception
    {
        timeUnit.sleep(duration);
        return timeUnit.toMillis(duration);
    }
}
```

Das Interface `ExecutorService`: Ausführen von Tasks

Nachdem die auszuführenden Tasks in Form von Implementierungen der Interfaces `Runnable` oder `Callable<V>` definiert sind, müssen diese geeignet an eine Ausführungseinheit übergeben werden. Eine Form davon sind verschiedene Arten von Thread-Pools, die alle jeweils das Interface `java.util.concurrent.ExecutorService` erfüllen. Dieses erweitert das Interface `Executor` unter anderem um die Möglichkeit, bereits laufende Aufgaben abbrechen zu können.

Ein `ExecutorService` bietet folgende Methoden zum Ausführen von Tasks:

- `<V> Future<V> submit(Callable<V> task)` – Es erfolgt eine Abarbeitung des `Callable<V>` mit anschließender Möglichkeit, das Ergebnis auszuwerten.
- `Future<?> submit(Runnable task)` – Das übergebene `Runnable` wird von einem Thread des `ExecutorService` abgearbeitet. Aufgrund der zuvor beschriebenen Einschränkungen von `Runnable`s ist es im Gegensatz zur Ausführung von `Callable<V>` nur möglich, abzufragen, ob die Abarbeitung bereits beendet ist. Es wird jedoch kein Rückgabewert über `get()` geliefert, sondern immer `null`.
- `<T> Future<T> submit(Runnable task, T result)` – Erweitert den vorherigen Aufruf um die Möglichkeit, mit `get()` ein Ergebnis abfragen zu können. Nach der Abarbeitung des Tasks wird der als Referenz übergebene Wert `result` vom Typ `T` zurückgeliefert. Das ist jedoch nur dann sinnvoll, wenn dieser Typ veränderlich ist und der Task den Inhalt als Folge seiner Berechnungen ändert. Das ist zwar ein Seiteneffekt, dieser ist aber lokal begrenzt und daher vertretbar.

Die Methode `submit()` nimmt zu einem Zeitpunkt genau ein `Callable<V>` an und führt es aus. Häufig möchte man jedoch mehrere Tasks parallel abarbeiten lassen. Dazu kann man sukzessiv `submit()` aufrufen. Eine Alternative stellen die folgenden Methoden des `ExecutorService` dar:

- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)` – Führt alle Aufgaben aus und liefert eine Liste von `Future<T>`-Objekten zum Zugriff auf die Ergebnisse.
- `T invokeAny(Collection<Callable<T>> tasks)` – Hierdurch werden zwar alle Aufgaben ausgeführt, es wird allerdings nur das Ergebnis des Tasks geliefert, der als Erster fertig ist. Die anderen Berechnungen werden abgebrochen.

Für beide Methoden existiert eine überladene Version, der man eine maximale Ausführungszeit mitgeben kann. Das berechnete Ergebnis wird nur geliefert, wenn es zu keiner Zeitüberschreitung kommt.

Das `Future<V>`-Interface: Rückgaben von Tasks

Zum Verfolgen des Ausführungsfortschritts eines übergebenen Tasks erfolgt die Rückgabe eines `java.util.concurrent.Future<V>`-Interface, das sowohl das Ergebnis einer Berechnung als auch deren Asynchronität kapselt. Das Interface `Future<V>` ist im JDK folgendermaßen definiert:

```
public interface Future<V>
{
    V get() throws InterruptedException, ExecutionException;
    V get(long Timeout, TimeUnit unit) throws
        InterruptedException, ExecutionException, TimeoutException;
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Mithilfe dieses Interface kann man unter anderem den Lebenszyklus von Tasks abfragen. Überladene `get()`-Methoden ermitteln den Ergebniswert und blockieren so lange, bis das Ergebnis verfügbar ist oder eine angegebene Timeout-Zeit überschritten wurde.

Ausprägungen vom Interface `ExecutorService`

Spezielle Realisierungen des Interface `ExecutorService` können mit folgenden Methoden der Utility-Klasse `java.util.concurrent.Executors` erzeugt werden:

- `newFixedThreadPool(int poolSize)` – Erzeugt einen Thread-Pool fester Größe.
- `newCachedThreadPool()` – Erzeugt einen Thread-Pool unbegrenzter Größe, der nach Bedarf wachsen und schrumpfen kann. Auf diese Weise kann dynamisch auf Belastungsspitzen reagiert werden.
- `newSingleThreadExecutor()` – Erzeugt einen Thread-Pool, der lediglich einen einzigen Thread verwaltet und folglich übergebene Aufgaben sequenziell abarbeitet (dies jedoch parallel zur eigentlichen Applikation).
- `newScheduledThreadPool(int poolSize)` – Erzeugt einen Thread-Pool der angegebenen Größe, der eine zeitgesteuerte Ausführung unterstützt und als Ersatz für die Klassen `Timer` und `TimerTask` dienen kann.

Wenn gegen das Interface `ExecutorService` programmiert wird, können die verwendeten, konkreten Realisierungen ausgetauscht und auf den jeweiligen Einsatz abgestimmt gewählt werden. Ein weiterer Vorteil der eben genannten Thread-Pool-Implementierungen ist, dass diese eine Fehlerbehandlung durchführen: Kommt es bei der Abarbeitung eines Tasks zu einer Exception, so führt dies zum Ende des ausführenden Worker-Threads. Durch den Thread-Pool werden »sterbende« Worker-Threads automatisch ersetzt, sodass für weitere Tasks immer eine Abarbeitung garantiert wird. Dies ist ein entscheidender Vorteil gegenüber vielen handgeschriebenen Varianten von Thread-Pools und auch gegenüber einem `java.util.Timer`. Dort führt eine Exception zum Abbruch aller angemeldeten `java.util.TimerTask`s.

Die erzeugten Thread-Pools stellen spezielle Ausprägungen der folgenden beiden Klassen dar:

- `ThreadPoolExecutor` – Diese Klasse realisiert einen Thread-Pool. Neu eintreffende Aufgaben werden von bereitstehenden Threads bearbeitet, d. h. von solchen, die momentan kein `Runnable` ausführen. Gibt es keinen freien Thread, so werden die Tasks in einer Warteliste zur Abarbeitung vorgemerkt.
- `ScheduledThreadPoolExecutor` – Diese Klasse erlaubt es, Tasks zeitgesteuert und insbesondere auch periodisch auszuführen und ähnelt damit der Klasse `Timer`.

11.2.3 Parallele Abarbeitung im `ExecutorService`

Zuvor haben wir die Grundlagen zu Tasks und deren Ausführung kennengelernt, nun wollen wir das Thema vertiefen und dazu eine Verarbeitung, angestoßen über `submit()`-Methoden, anschauen. Bei Bedarf könnten parallel weitere Aufgaben gestartet werden. Jede einzelne lässt sich über das zurückgelieferte `Future<V>`-Interface mit der Methode `isDone()` befragen, ob deren Berechnung beendet ist. Folgendes Listing zeigt dies exemplarisch für zwei Berechnungen mit dem bereits bekannten Task `CalcDurationInMs`. Der erste Task wartet 5, der zweite 10 Sekunden:

```
public static void main(final String[] args)
{
    final int POOL_SIZE = 3;
    final ExecutorService executorService =
                          Executors.newFixedThreadPool(POOL_SIZE);

    // Definition und Start zweier Tasks
    final Future<Long> future1 = executorService.submit(
                                 new CalcDurationInMs(TimeUnit.SECONDS, 5));
    final Future<Long> future2 = executorService.submit(
                                 new CalcDurationInMs(TimeUnit.SECONDS, 10));

    System.out.println("Start: " + LocalTime.now());
    try
    {
        // synchron auf das Ende von Task 1 warten
        final Long result1 = future1.get();
        System.out.println("After Job 1: " + LocalTime.now());
        System.out.println(result1);
```

```
        // Zugriff nach 5 Sekunden sollte false liefern
        System.out.println("isDone? Job 2: " + future2.isDone());

        // synchron auf das Ende von Task 2 warten
        final Long result2 = future2.get();
        System.out.println("After Job 2: " + LocalTime.now());
        System.out.println(result2);
    }
    catch (final InterruptedException | ExecutionException ex)
    {
        // Kann in diesem Beispiel nicht auftreten, s. o.
    }

    // Aufruf, um Thread-Pool zu beenden und JVM-Terminierung zu ermöglichen
    executorService.shutdown();
}
```

Listing 11.1 *Ausführbar als* '**EXECUTORSERVICEEXAMPLE**'

An den Ausgaben – insbesondere dem Start 5 bzw. 10 Sekunden vor dem jeweiligen Ende – sieht man, dass die Abarbeitung parallel ausgeführt wird:

```
Start: 14:05:30.691937
After Job 1: 14:05:35.655458
5000
isDone? Job 2: false
After Job 2: 14:05:40.656019
10000
```

Auf ein Detail möchte ich noch eingehen: Im Listing sehen Sie als letzte Anweisung den Aufruf von `executorService.shutdown()`. Dies dient dazu, den Executor Service geordnet zu beenden und belegte Ressourcen freizugeben. Insbesondere werden keine neuen Tasks mehr akzeptiert, die laufenden aber noch abgearbeitet. Möchte man auf deren Ende warten, so sollte noch ein Aufruf von `awaitTermination()` erfolgen, was in diesem Beispiel nicht nötig ist, da bereits alle Tasks durch die blockierende Abfrage von `get()` abgearbeitet sind. Zudem gibt es noch die Methode `shutdownNow()`, die den momentan noch im Thread-Pool abgearbeiteten Threads einen Interrupt sendet.

Zeitgesteuerte Ausführung mit `ScheduledExecutorService`

Das Interface `ScheduledExecutorService` bietet Methoden, um Aufgaben zu bestimmten Zeiten bzw. wiederholt auszuführen, und wird von der Klasse `ScheduledThreadPoolExecutor` implementiert.

Zur Demonstration der Arbeitsweise der Klassen wird hier folgende simple Implementierung eines `Runnable`s zur Ausgabe eines Textes auf der Konsole verwendet:

```
public class SampleMessageTask implements Runnable
{
    private final String message;

    SampleMessageTask(final String message)
    {
        this.message = message;
    }
```

```
    public void run()
    {
        System.out.println(message);
    }
}
```

Einmalige Ausführungen Diese gerade gezeigte Klasse wird in der nachfolgenden `main()`-Methode genutzt, um die zeitgesteuerte Verarbeitung zu demonstrieren, nämlich eine sofortige Ausführung, eine nach 5 Sekunden und eine weitere nach 1 Minute. Diese werden mit der Methode `schedule()` eingeplant:

```
public static void main(final String[] args) throws InterruptedException
{
    final int POOL_SIZE = 3;
    final ScheduledExecutorService executorService =
                                   Executors.newScheduledThreadPool(POOL_SIZE);

    // Sofortige Ausführung
    executorService.schedule(new SampleMessageTask("OnceImmediately"), 0,
                             TimeUnit.SECONDS);

    // Ausführung nach fünf Sekunden
    executorService.schedule(new SampleMessageTask("OnceAfter5s"), 5,
                             TimeUnit.SECONDS);

    // Ausführung nach einer Minute
    executorService.schedule(new SampleMessageTask("OnceAfter1min"), 1,
                             TimeUnit.MINUTES);

    executorService.shutdown();
    executorService.awaitTermination(65, TimeUnit.SECONDS);
}
```

Listing 11.2 *Ausführbar als* '**SCHEDULEDEXECUTOREXAMPLE**'

Durch die Verwendung der Klasse `TimeUnit` lassen sich Zeitangaben gut lesbar notieren. Auch kann man das Warten mit `awaitTermination()` sauber wie im Listing gezeigt realisieren. Dieses ist für das Beispiel nötig, weil so der einzige existierende Vordergrund-Thread (der `main()`-Thread) am Laufen gehalten wird, damit all die anderen (Daemon-)Threads überhaupt zum Zuge kommen können. Genauer: Die Verarbeitung der im `ExecutorService` eingeplanten Aufgaben abgearbeitet werden können.

Mehrmalige Ausführungen Zudem besitzen die Methoden zur Ausführung mit festem Takt und festgelegter Verzögerung zwischen den Ausführungen sprechende Namen: `scheduleAtFixedRate()` und `scheduleWithFixedDelay()`. Folgendes Beispiel zeigt beide Varianten der wiederholten Ausführung. Zum einen eine periodische Ausführung mit einem Abstand von untereinander 30 Sekunden, die einen zyklischen Refresh nachahmt. Zum anderen erfolgt die Ausgabe mit fixem Takt, hier von 10 Sekunden.

```
public static void main(final String[] args) throws InterruptedException
{
    final int POOL_SIZE = 3;
    final ScheduledExecutorService executorService =
                                   Executors.newScheduledThreadPool(POOL_SIZE);

    // Eingeplante Ausführung mit INITIAL_DELAY und Verzögerung DELAY
    final long INITIAL_DELAY_ONE_SEC = 1;
    final long DELAY_30_SECS = 30;
    executorService.scheduleWithFixedDelay(new SampleMessageTask(
                "PeriodicRefreshing"), INITIAL_DELAY_ONE_SEC, DELAY_30_SECS,
                TimeUnit.SECONDS);

    // Eingeplante Ausführung mit INITIAL_DELAY und Takt RATE
    final long RATE_10_SECS = 10;
    executorService.scheduleAtFixedRate(new SampleMessageTask(
                "10s FixedRateExecution"), INITIAL_DELAY_ONE_SEC, RATE_10_SECS,
                TimeUnit.SECONDS);

    // Shutdown in der Praxis nicht sinnvoll, solange noch eine
    // periodische Ausführung gewünscht ist
    executorService.shutdown();
    executorService.awaitTermination(65, TimeUnit.SECONDS);
}
```

Listing 11.3 *Ausführbar als* '**SCHEDULEDEXECUTORPERIODICEXAMPLE**'

Fazit

Dieser Abschnitt hat einen Einstieg in das Executor-Framework gegeben. Ein großer Vorteil besteht darin, dass man von vielen Details der Thread-Verwaltung abstrahieren kann. Dadurch gibt es nahezu keinen Grund mehr, direkt mit Threads zu arbeiten. Wenn viele Tasks auszuführen sind, gelten diese Aussagen umso mehr. ***In neuen Projekten sollten `Executor`-Realisierungen bevorzugt werden***. Dies empfiehlt auch Joshua Bloch in seinem Buch »Effective Java« [6] in Item 68.

11.3 Das Fork-Join-Framework

Mit JDK 7 wurde das Executor-Framework um das sogenannte Fork-Join-Framework erweitert, das die Parallelverarbeitung von Aufgaben unterstützt. Die dazu eingeführten `ForkJoinTask`s bilden eine plattformunabhängige Möglichkeit, rechenintensive Operationen tatsächlich parallel auf Mehrprozessormaschinen auszuführen. Diese Art der Verarbeitung lässt sich auf viele Algorithmen der Kategorie »teile und herrsche« (***divide and conquer***) anwenden. Die gewählte Lösungsstrategie besteht darin, ein größeres Problem in diverse kleinere zu lösende Probleme aufzuteilen und diesen Vorgang so lange rekursiv fortzusetzen, bis eine einfache (häufig nur aus wenigen Anweisungen bestehende) Lösung möglich ist. Man spricht hier vom rekursiven Abstieg mit Abbruchkriterium. Das Besondere an den Problemstellungen ist, dass diese voneinander unabhängig sind und daher parallel gelöst werden können.

11.3.1 Einführendes Beispiel

Zum Einstieg schauen wir uns die parallele Berechnung der Summe der kubierten Elemente (x^3) einer Liste an. Die Potenzierung nutzen wir exemplarisch für eine komplexere Aktion, da eine reine Summenbildung linear schneller wäre. Betrachten wir die zu parallelisierende Implementierung:

```
long cubicSum = 0;
for (int i = 0; i < values.size(); i++)
{
    cubicSum += Math.pow(values.get(i), 3);
}
```

Zur Parallelisierung wollen wir einen rekursiven Algorithmus nutzen. Dieser untergliedert eine Liste in jeweils zwei Teile, indem die Originalliste in deren Mitte aufgespalten wird. Das wiederholt man, bis die Größe der jeweiligen Listen 0 oder 1 Element ist.

Klassen aus dem Fork-Join-Framework Die Grundlage für die Implementierung und für die rekursive Berechnung bildet die abstrakte Klasse `RecursiveTask<V>` als Erweiterung der Klasse `ForkJoinTask<V>`. Für eigene Implementierungen leitet man von `RecursiveTask<V>` ab und überschreibt die abstrakte Methode `compute()` mit dem Rückgabetyp `V`.

```
public static class CubicSumTask extends RecursiveTask<Long>
{
    private final List<Long> values;

    public CubicSumTask(final List<Long> values)
    {
        this.values = values;
    }

    @Override
    protected Long compute()
    {
        // TODO
    }
}
```

Parallele Ausführung mit dem Fork-Join-Framework Kommen wir nun zur Umsetzung der rekursiven Funktionalität. Zunächst definieren wir den rekursiven Abbruch für eine Liste mit nur keinem oder einem Element. Dort ist die Rückgabe demnach entweder 0 oder der kubische Wert (x^3). Wie zuvor beschrieben teilen wir die Ausgangsdaten immer in der Mitte. Dies erfolgt mithilfe von `subList()`. Bis hierher merken wir eigentlich noch gar nichts vom Fork-Join-Framework oder von Parallelisierung. Um die beiden Hälften nun parallel abzuarbeiten, erzeugen wir jeweils einen weiteren `CubicSumTask`. Damit dieser mit den Worker-Threads des Fork-Join-Frameworks ausgeführt wird, übergeben wir diese an `invokeAll()`. Schließlich müssen wir noch die Summe der Ergebnisse bilden. Dazu ermitteln wir mithilfe von `join()` die Summe der

Tasks und addieren beide Werte. Nutzen wir das gewonnene Wissen zur Implementierung einer `compute()`-Methode wie folgt:

```
@Override
protected Long compute()
{
    // rekursiver Abbruch
    if (values.size() == 0)
        return 0L;

    if (values.size() == 1)
        return Math.pow(values.get(0), 3);

    // rekursiver Abstieg
    final int midPos = values.size() / 2;
    final List<Long> lowerPart = values.subList(0, midPos);
    final List<Long> upperPart = values.subList(midPos, values.size());

    final CubicSumTask left = new CubicSumTask(lowerPart);
    final CubicSumTask right = new CubicSumTask(upperPart);
    invokeAll(left, right);

    return left.join() + right.join();
}
```

Zur Ausführung derartiger Tasks muss man sich einen Pool an Worker-Threads in Form der Klasse `ForkJoinPool` anlegen bzw. den bereits vordefinierten CommonPool mit `commonPool()` verwenden. Anschließend wird durch den Aufruf der Methode `invoke(ForkJoinTask<T>)` die Berechnung gestartet:

```
private static long cubicSumWithFj(final List<Long> values)
{
    return ForkJoinPool.commonPool().invoke(new CubicSumTask(values));
}
```

Führen Sie dies als CUBICSUMMERFJ_V1 aus, so wird deutlich, dass erst bei 100 Millionen Einträgen die Variante mit Fork-Join mit rund 2.500 ms minimal schneller als die normale Schleife mit 2.800 ms ist. Dabei geschehen die Messungen auf Basis folgender Anweisungen:

```
var values = new ArrayList<Long>();
for (int i = 0; i < 100_000_000; i++)
    values.add(i, (long) (Math.random() * 7));

long start = System.currentTimeMillis();
long cubicSum = 0;
for (int i = 0; i < values.size(); i++)
{
    cubicSum += Math.pow(values.get(i), 3);
}
long end = System.currentTimeMillis();
System.out.println("result " + cubicSum + " / loop ms: " + (end - start));

long start2 = System.currentTimeMillis();
long cubicSum2 = cubicSumWithFj(values);
long end2 = System.currentTimeMillis();
System.out.println("result " + cubicSum2 + " / fj ms:" + (end2 - start2));
```

Wissenswertes bei der parallelen Ausführung Bei den `ForkJoinTasks` sind jedoch zwei Aspekte zu beachten: Zum einen ist dies die Komplexität des zu lösenden Problems und zum anderen der Verwaltungsoverhead, der durch das Aufteilen in kleinere Teilaufgaben entsteht. Dieser führt dazu, dass ab einer gewissen Granularität (Anzahl an Anweisungen) die sequenzielle Berechnung schneller ist als die parallele. Es bedarf allerdings einiges an Erfahrung und Fingerspitzengefühl sowie Messungen, um diese Grenze für die Aufspaltung passend wählen zu können. Weiterführende Informationen finden Sie in Doug Leas Buch »Concurrent Programming in Java« [48].

Nachfolgend gehen wir vereinfachend davon aus, dass ab 7 Elementen die weitere Unterteilung in Fork-Join-Tasks keine Vorteile mehr bringt. Damit wandeln wir die Berechnung folgendermaßen ab:

```
public static class CubicSumTask extends RecursiveTask<Long>
{
    private static final int THRESHOLD = 7;

    private final List<Long> values;

    public CubicSumTask(List<Long> values)
    {
        this.values = values;
    }

    @Override
    protected Long compute()
    {
        // rekursiver Abbruch
        if (values.size() < THRESHOLD)
        {
            long sum = 0;
            for (int i = 0; i < values.size(); i++)
                sum += Math.pow(values.get(i), 3);
            return sum;
        }

        // rekursiver Abstieg
        final int midPos = values.size() / 2;
        final List<Long> lowerPart = values.subList(0, midPos);
        final List<Long> upperPart = values.subList(midPos, values.size());

        final CubicSumTask left = new CubicSumTask(lowerPart);
        final CubicSumTask right = new CubicSumTask(upperPart);
        invokeAll(left, right);

        return left.join() + right.join();
    }
}
```

Führen Sie dies als CUBICSUMMERFJ_V2 aus, so wird erkennbar, dass diese Optimierung deutlich positivere Auswirkungen besitzt: Die Laufzeit der Variante mit Fork-Join ist für 100 Millionen `long`-Werte mit 2.000 ms nun deutlich schneller als die der normalen Schleife mit 2.800 ms. Zu erklären ist dies dadurch, dass viel weniger `CubicSumTasks` erzeugt werden und auch weniger davon mit Worker-Threads verarbeitet werden müssen. Hier zeigt sich, dass die »gesunde« Mischung aus sequenziellen und parallelen Anteilen die beste Performance liefert.

11.3.2 Real-World-Example: Merge Sort

Das einführende Beispiel war vereinfacht, um die prinzipiellen Abläufe klar und deutlich darstellen zu können. Ein praxisorientiertes Beispiel für die rekursive Aufteilung mit einer Problemlösungskomponente bei nur wenigen verbliebenen Elementen ist das Sortieren eines Arrays mit Merge Sort in Kombination mit Insertion Sort.

Die algorithmischen Details sind hier nicht von gesteigertem Interesse. Für eine ausführlichere Beschreibung der Grundlagen zur Sortierung und der Arbeitsweise von Merge Sort und Insertion Sort verweise ich Sie auf mein Buch »Java Challenge« [42].

Ausgangsbasis: Rekursiver Merge Sort

Merge Sort basiert auf einem Divide-and-Conquer-Ansatz und zerteilt rekursiv den zu sortierenden Datenbestand in immer kleiner werdende Teile von etwa der Hälfte der ursprünglichen Größe, bis diese nur noch aus einem oder gegebenenfalls keinem Element bestehen. Danach werden die Teile wieder vereint. In diesem Merge-Schritt erfolgt das Sortieren durch die passende, auf den jeweiligen Werten basierende Zusammenführung. Der Algorithmus lässt sich rekursiv und gut verständlich implementieren. Zum Zusammenführen der Teilergebnisse kommt die Realisierung der Hilfsmethode `merge()` zum Einsatz, die später der Vollständigkeit halber gezeigt wird:

```
static List<Long> mergesort(final List<Long> values)
{
    // Rekursiver Abbruch: Länge 0 (nur wenn initial leeres Array) oder 1
    if (values.size() <= 1)
        return values;

    // Rekursiver Abstieg: teile in zwei Hälften
    final int midPos = values.size() / 2;
    final List<Long> lowerPart = values.subList(0, midPos);
    final List<Long> upperPart = values.subList(midPos, values.size());

    final List<Long> sortedLeft = mergesort(lowerPart);
    final List<Long> sortedRight = mergesort(upperPart);

    return merge(sortedLeft, sortedRight);
}
```

Analogie aus dem realen Leben führt zu Optimierung Für Merge Sort lässt sich eine Analogie zum Sortieren von Karten eines Kartenspiels nutzen. Wenn man einen ziemlich großen Stapel an Karten sortieren muss, kann man diesen in viele, deutlich kleinere Stapel aufteilen, diese für sich sortieren und danach sukzessive vereinigen. Statt jedoch die Stapel bis auf eine Karte zu reduzieren, bietet es sich an, die kleineren Stapel mit einem anderen Verfahren zu sortieren, oftmals mit Insertion Sort, das für kleine, idealerweise fast geordnete Arrays bzw. Listen eine Laufzeit von $O(n)$ besitzt. Dieses Sortierverfahren kann man zum Feintunen verwenden. Genialerweise ist dies bei Merge Sort kinderleicht:

```
static List<Long> mergesortWithInsertionsort(final int[] toSort)
{
    // Rekursiver Abbruch inklusive Mini-Optimierung
    if (toSort.length < 7)
    {
        InsertionSortExample.insertionSort(toSort);
        return toSort;
    }

    // Rekursiver Abstieg: Teile in zwei Hälften
    final int midPos = toSort.length / 2;
    final List<Long> lowerPart = values.subList(0, midPos);
    final List<Long> upperPart = values.subList(midPos, values.size());

    final List<Long> sortedLeft = mergesortWithInsertionsort(lowerPart);
    final List<Long> sortedRight = mergesortWithInsertionsort(upperPart);

    return merge(sortedLeft, sortedRight);
}
```

Abschließend möchte ich noch darauf hinweisen, dass die Grenze, ab wann man auf Insertion Sort wechseln sollte, hier recht willkürlich auf den Wert 7 festgesetzt wurde. Vermutlich sind Werte zwischen 10 und 20 Elementen ziemlich praxistauglich, jedoch sollte man dann auf das Wissen von Algorithmen-Profis zurückgreifen.

Tipp: Zusammenführen von zwei Listen

Das Zusammenführen von zwei Listen wird wie folgt implementiert:

```
static List<Long> merge(final List<Long> values1, final List<Long> values2)
{
    final List<Long> result = new ArrayList<>();
    int idx1 = 0;
    int idx2 = 0;

    while (idx1 < values1.size() && idx2 < values2.size())
    {
        final long value1 = values1.get(idx1);
        final long value2 = values2.get(idx2);

        if (value1 < value2)
        {
            result.add(value1);
            idx1++;
        }
        else
        {
            result.add(value2);
            idx2++;
        }
    }
    for (int i = idx1; i < values1.size(); i++)
        result.add(values1.get(i));
    for (int i = idx2; i < values2.size(); i++)
        result.add(values2.get(i));

    return result;
}
```

Merge Sort mit Fork-Join

Schauen wir uns nun an, wie wir die vorherige Lösung unter Zuhilfenahme des Fork-Join-Frameworks parallelisieren können. Beim Merge Sort unterscheidet man zunächst das Aufsplitten und dann das Zusammenfügen. Jetzt müssen wir dies auf die jeweiligen Bestandteile des Fork-Join-Frameworks abbilden: Recht offensichtlich ist, dass das Untergliedern bzw. Aufspalten dem Fork und das Zusammenführen dem Join entspricht. Wie schon angedeutet, führen wir die Unterteilung nur bis zu einem Grenzwert fort. Zum einen wird das Ganze dadurch praxisrelevanter und zum anderen lohnt sich eine weitere Parallelisierung nicht mehr, da eine sequenzielle Abarbeitung, hier mit Insertion Sort, performanter wird.

Machen wir es konkret: Wir nutzen `RecursiveTask<T>` als Basis und überschreiben die `compute()`-Methode. Diese entspricht abgesehen von Kleinigkeiten der ursprünglichen Methode `mergesort()`, hier mit anderem Rückgabewert in Form einzelner `MergesortTask`s sowie dem expliziten Aufruf der Verarbeitung mit `invokeAll()`:

```
public static class MergesortTask extends RecursiveTask<List<Long>>
{
    private static final int THRESHOLD = 7;
    private final List<Long> values;

    public MergesortTask(final List<Long> values)
    {
        this.values = values;
    }

    @Override
    protected List<Long> compute()
    {
        if (values.size() < THRESHOLD)
        {
            InserionSortExample.insertionSort(values);
            return values;
        }

        final int midPos = values.size() / 2;
        final List<Long> lowerPart = values.subList(0, midPos);
        final List<Long> upperPart = values.subList(midPos, values.size());

        final MergesortTask left = new MergesortTask(lowerPart);
        final MergesortTask right = new MergesortTask(upperPart);
        invokeAll(left, right);

        return merge(left.join(), right.join());
    }
}
```

Beim Zusammenführen greifen wir auf die bereits bestehende Funktionalität zurück. Um an die Resultate vorheriger Berechnungen zu kommen, rufen wir `join()` auf den gestarteten einzelnen `MergesortTask`s auf.

11.4 Die Klasse CompletableFuture<T>

Die Klasse java.util.concurrent.CompletableFuture<T> implementiert das Interface Future<V> (vgl. Abschnitt 11.2.2) und bietet eine Erleichterung bei der Programmierung asynchroner Abläufe. Diese lassen sich bezüglich Lesbarkeit und Komplexität deutlich besser als die Lösungen mit Threads gestalten. Das liegt daran, dass bei asynchronen Berechnungen diese miteinander zu synchronisieren oder Ergebnisse auszutauschen sind, was über Runnable oder Callable<V> schnell unhandlich wird. Die Klasse CompletableFuture<T> führt zu einer deutlichen Vereinfachung, weil die Details der Thread-Abstimmung für den Benutzer unsichtbar ablaufen.

11.4.1 Einführung

Weil hier ein neues Konzept eingeführt wird, beginne ich mit einem stark vereinfachten Beispiel. Normalerweise nutzt man CompletableFuture<T> zur Ausführung länger dauernder Aktionen. Diese sind hier durch einfache Berechnungen stilisiert:

1. Initial erfolgt das Berechnen eines Ergebnisses als simple Rückgabe eines Strings (longRunningAction).
2. Symbolisch für komplexe Berechnungen werden zunächst eine Wandlung eines Strings in ein Integer-Objekt (complexConverter) und
3. dann eine Berechnung, hier eine einfache Multiplikation (complexCalculation), ausgeführt.
4. Schließlich wird das Ergebnis per get() ausgelesen.

```
public static void main(final String[] args) throws InterruptedException,
                                                    ExecutionException
{
    // Auszuführende Aktionen
    final Supplier<String> longRunningAction = () ->
    {
        System.out.println("Current thread: " + Thread.currentThread());
        return "101";
    };
    final Function<String, Integer> complexConverter = Integer::parseInt;
    final Function<Integer, Double> complexCalculation = value -> .75 * value;

    // Schritt 1: Aufwendige Berechnung, hier nur Rückgabe eines Strings
    final CompletableFuture<String> step1 =
                          CompletableFuture.supplyAsync(longRunningAction);

    // Schritt 2: Konvertierung, hier nur Abbildung von String auf Integer
    final CompletableFuture<Integer> step2 = step1.thenApply(complexConverter);

    // Schritt 3: Konvertierung, hier nur Multiplikation mit .75
    final CompletableFuture<Double> step3 = step2.thenApply(complexCalculation);

    // Schritt 4: Auslesen per get() wartet blockierend auf Fertigstellung
    System.out.println(step3.get());
}
```

Listing 11.4 *Ausführbar als* **'FIRSTCOMPLETABLEFUTUREEXAMPLE'**

Im Listing sehen wir, wie man die beschriebenen zu parallelisierenden Schritte mithilfe von `CompletableFuture<T>` verarbeiten kann. Dabei kommen folgende Methoden zum Einsatz:

- `supplyAsync()` – Konstruiert ein neues `CompletableFuture<T>`-Objekt.
- `thenApply()` – Führt eine Aktion in Form einer `Function<T,R>` aus.
- `get()` – Greift auf das Ergebnis zu, wartet blockierend, falls dieses noch nicht vorliegt.

Die Aufrufe erfolgen in dem Thread des Thread-Pools (blockierend), der die Aktion ausführt. Darüber hinaus existiert für `thenApply()` eine asynchrone Variante, deren Methodenname mit `Aysnc` endet. Damit kann man Aufrufketten parallelisieren.

Das Beispiel zeigt, dass man mit `CompletableFuture<T>` auf einfache Weise asynchrone Abläufe erzeugen und an definierten Punkten wieder zusammenlaufen lassen kann. Führt man das Programm FIRSTCOMPLETABLEFUTUREEXAMPLE aus, so wird dies anhand der Protokollierung des aktuellen Threads und folgender Konsolenausgabe deutlich:

```
Current thread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
75.75
```

Die Komplexität im Vergleich zu einer Lösung mittels Threads ist erheblich geringer. Man muss sich weder um die Erzeugung von Threads noch um deren Interaktion selbst kümmern. Stattdessen erfolgen die Berechnungen automatisch durch Threads, die aus dem `commonPool` des Fork-Join-Frameworks stammen.

Verarbeitungsketten mit `CompletableFuture` aufbauen

Das folgende Beispiel zeigt, wie man Verarbeitungsketten aufbauen kann, etwa um Aktionen parallel zum Aufrufer oder nach Abschluss der Abarbeitung auszuführen. Dafür nutzen wir folgende Methoden:

1. `supplyAsync()` – Als Ausgangspunkt einer Verarbeitung wird mit `supplyAsync()` asynchron eine Aktion abgearbeitet.
2. `thenAccept()` – Als Nächstes wird mit `thenAccept()` eine Aktion nach Abschluss der potenziell aufwendigen Verarbeitung ausgeführt. Dies kann auch asynchron durch `thenAcceptAsync()` geschehen.
3. `exceptionally()` – Schließlich lässt sich durch `exceptionally()` elegant auf das Auftreten von Exceptions reagieren.

Im kommenden Listing nutze ich initial mit `thenAcceptAsync()` die asynchrone und für die Verarbeitung des Ergebnisses beim Exception Handling mit `thenAccept()` die synchrone Ausführung.

Mit diesem Wissen implementieren wir verschiedene asynchrone Abläufe als Beispiel zu `CompletableFuture<T>` in der Methode `perform()` wie folgt:

```
public static void main(final String[] args) throws InterruptedException,
                                                    ExecutionException
{
    new CompletableFutureProcessingChainExample().perform();
}

public void perform() throws InterruptedException, ExecutionException
{
    // 1) supplyAsync(): eine einzelne Aktion parallel zum Aufrufer ausführen
    final CompletableFuture<String> supplyAsync =
          CompletableFuture.supplyAsync(() -> longRunningCreateMsg(2));

    // 2) thenAcceptAsync(): Aktion nach Abschluss der Ausführung
    supplyAsync.thenAcceptAsync(this::notifySubscribers);

    // 3) exceptionally(): Exception-Mapping
    CompletableFuture.supplyAsync(this::failingMsg)
                     .exceptionally(ex -> "FAILED IN BETWEEN")
                     .thenAccept(this::notifySubscribers);

    Thread.sleep(5_000);
}

public String longRunningCreateMsg(final int durationInSecs)
{
    System.out.println(getCurrentThread() + " >>> longRunningCreateMsg");

    sleepInSeconds(durationInSecs);
    System.out.println(getCurrentThread() + " <<< longRunningCreateMsg");

    return "longRunningCreateMsg";
}

public String getCurrentThread()
{
    return Thread.currentThread().getName();
}

public void notifySubscribers(final String msg)
{
    System.out.println(getCurrentThread() + " notifySubscribers: " + msg);
}

public String failingMsg()
{
    throw new IllegalStateException("ISE");
}

private void sleepInSeconds(final int durationInSeconds)
{
    try
    {
        TimeUnit.SECONDS.sleep(durationInSeconds);
    }
    catch (InterruptedException e)
    {  /* not possible here */ }
}
```

Listing 11.5 *Ausführbar als* '**COMPLETABLEFUTUREPROCESSINGCHAINEXAMPLE**'

Zunächst wird eine Nachricht per `longRunningCreateMsg()` mit einer Abarbeitungszeit von 2 Sekunden erzeugt und mit `supplyAsync()` asynchron (parallel zum aufrufenden Thread, in diesem Fall dem `main()`-Thread) ausgeführt. Als darauffolgende Aktion wird `notifySubscribers()` durch Aufruf von `thenAcceptAsync()` registriert und nach Abschluss der ersten Aktion abgearbeitet. Zum Abschluss sehen wir ein Beispiel für das Exception Handling. Da die Aktionen parallel zum `main`-Thread laufen, wird das Ergebnis schon vor Abarbeitung von `longRunningCreateMsg()` ausgegeben.

Die Ausgaben des Programms COMPLETABLEFUTUREPROCESSINGCHAINEXAMPLE erleichtern das Nachvollziehen der beschriebenen Abläufe, insbesondere auch der jeweils ausführenden Threads:

```
ForkJoinPool.commonPool-worker-3 >>> longRunningCreateMsg
main notifySubscribers: FAILED IN BETWEEN
ForkJoinPool.commonPool-worker-3 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-5 notifySubscribers: longRunningCreateMsg
```

11.4.2 Beispiel: Parallele Verarbeitung von Dateiinhalten

Als praktisches Anwendungsbeispiel soll der Inhalt einer Datei eingelesen und analysiert werden. Weil man beim (professionellen) Programmieren immer auch ein Augenmerk auf Wartbarkeit und Testbarkeit legen sollte, entwickeln wir die Funktionalität als kleine Bausteine. Diese realisieren wir in Form von Methoden mit folgenden Aktionen: Das Einlesen der Datei geschieht zeilenweise. Die Zeilen werden dann als einzelne Worte aufbereitet. Diese Funktionalität ist Schritt 1 und wird durch die Methode `extractWordsFromFile(Path)` und den Aufruf von `supplyAsync()` realisiert. Im Anschluss daran sollen in einem zweiten Schritt zwei Filterungen parallel auf den bereitgestellten Daten durchgeführt werden: Zum einen werden zu ignorierende Wörter herausgefiltert, zum anderen werden Wörter mit weniger als vier Buchstaben aus dem Ergebnis entfernt. Beide Verarbeitungen werden durch `thenApplyAsync()` angestoßen und laufen parallel zueinander ab. In Schritt 3 werden nun die beiden parallel berechneten Ergebnisse miteinander per `thenCombine()` verbunden. Dass dies erst nach Abschluss beider Berechnungen geschieht, darum kümmert sich die Logik aus der Klasse `CompletableFuture<T>`.

Das folgende Listing zeigt, dass man durch den Einsatz der Klasse `CompletableFuture<T>` und ihrer Methoden die Abläufe gut nachvollziehbar gestalten kann:

```
public static void main(final String[] args) throws Exception
{
    final Path exampleFile = Paths.get("src/main/resources/"+
                             "ch11_modern_concurrency/Example.txt");

    // Schritt 1: Möglicherweise längerdauernde Aktion
    final CompletableFuture<List<String>> contents =
               CompletableFuture.supplyAsync(extractWordsFromFile(exampleFile));

    contents.thenAccept(text -> System.out.println("Initial: " + text));
```

```
    // Schritt 2: Filterungen parallel ausführen
    final CompletableFuture<List<String>> filtered1 =
                            contents.thenApplyAsync(removeIgnorableWords());

    final CompletableFuture<List<String>> filtered2 =
                            contents.thenApplyAsync(removeShortWords());

    // Schritt 3: Verbinde die Ergebnisse
    final CompletableFuture<List<String>> result =
                            filtered1.thenCombine(filtered2, calcIntersection());

    System.out.println("result: " + result.get());
}
```

Listing 11.6 *Ausführbar als* '**COMPLETABLEFUTUREFILEPROCESSINGEXAMPLE**'

Im Listing sind die Verarbeitungsschritte und deren Kombination gut erkennbar, auch ohne die jeweiligen Berechnungen im Detail zu verstehen. Die zur Realisierung eingesetzten einzelnen Bausteine nutzen `Predicate<T>`-Objekte, mit denen wir Filterbedingungen formulieren (vgl. Abschnitt 4.5.1). Außerdem lesen wir Daten aus einer Datei mithilfe von `Files.readAllLines()` (vgl. Abschnitt 5.5.5), das die Daten als `Stream<String>` liefert. Auf diesem operieren wir mit den aus Kapitel 7 bekannten Methoden des Stream-APIs, `flatMap()`, `map()`, `sorted()` usw., um den Inhalt der Datei als eine Menge von Wörtern zurückzuliefern, wobei zuvor Satzzeichen von den Wörtern abgeschnitten werden. Nachfolgendes Listing zeigt die zuvor aufgerufene Hilfsmethode zum Einlesen aus einer Datei:

```
private static Supplier<List<String>> extractWordsFromFile(final Path inputFile)
{
    return () ->
    {
        try
        {
            final List<String> lines = Files.readAllLines(inputFile);

            final Stream<String> words = lines.stream().flatMap(line ->
                                                Stream.of(line.split(" ")));

            final Stream<String> mapped = words.map(removePunctationMarks());
            final Stream<String> sorted = mapped.sorted(
                                                  String.CASE_INSENSITIVE_ORDER);

            return sorted.collect(Collectors.toList());
        }
        catch (final Exception e)
        {
            return Collections.emptyList();
        }
    };
}
```

Um die Lesbarkeit und Verständlichkeit hoch zu halten wurden folgende Hilfsmethoden extrahiert:

```
private static Function<List<String>, List<String>> removeIgnorableWords()
{
    final List<String> wordsToIgnore = Arrays.asList("this", "This", "text");
    final Predicate<String> isIgnorable = word -> wordsToIgnore.contains(word);

    return input -> { return input.stream().filter(isIgnorable.negate())
                                           .collect(Collectors.toList()); };
}

private static Function<List<String>, List<String>> removeShortWords()
{
    final Predicate<String> isShortWord = word -> word.length() <= 3;
    final Predicate<String> notIsShortWord = isShortWord.negate();

    return input -> { return input.stream().filter(notIsShortWord)
                                           .collect(Collectors.toList()); };
}

private static BiFunction<? super List<String>, ? super List<String>,
                          ? extends List<String>> calcIntersection()
{
    return (list1, list2) -> { list1.retainAll(list2); return list1; };
}

private static Function<String, String> removePunctationMarks()
{
    final Function<String, String> removePunctationMarks = str ->
    {
        if (str.endsWith(".") || str.endsWith(":") || str.endsWith("!"))
        {
            return str.substring(0, str.length()-1);
        }
        return str;
    };
    return removePunctationMarks;
}
```

Startet man das zuvor gezeigte Programm COMPLETABLEFUTUREFILEPROCESSING-EXAMPLE, so kommt es bei folgender Eingabedatei

```
This is the first line.
The second line of this text.
Third line contains some text.
Last line and goodbye:
End of text!
```

zu diesen Konsolenausgaben:

```
Initial: [and, contains, End, first, goodbye, is, Last, line, line, line, line,
      of, of, second, some, text, text, text, the, The, Third, This, this]
result: [contains, first, goodbye, Last, line, line, line, line, second, some,
      Third]
```

Man erkennt die beiden parallel ablaufenden Filterungen und das Zusammenführen der jeweiligen Zwischenergebnisse.

11.4.3 Erweiterungen in JDK 9

In der Klasse CompletableFuture<T> wurden in Java 9 diverse Methoden ergänzt, unter anderem folgende:

- completeAsync(Supplier<? extends T>) und completeAsync(Supplier<? extends T>, Executor) – Gibt ein CompletableFuture<T> mit dem vom übergebenen Supplier<T> bereitgestellten Wert zurück. Das Ganze wird asynchron von einem Task als Ergebnis geliefert, der entweder vom Default Executor oder dem übergebenen ausgeführt wird.
- orTimeout(long, TimeUnit) – Sofern das CompletableFuture<T> nicht zuvor erfolgreich ausgeführt wurde, wird es mit einer java.util.concurrent.TimeoutException beendet, wenn die angegebene Timeout-Zeit erreicht ist.
- completeOnTimeout(T, long, TimeUnit) – Das CompletableFuture<T> wird mit dem übergebenen Wert erfüllt, falls die Berechnungen nicht innerhalb der gegebenen Timeout-Zeit zum Ergebnis führen.
- failedFuture(Throwable) – Gibt ein CompletableFuture<T> zurück, das bereits durch die übergebene Exception erfüllt wurde. Dies kann man zum Signalisieren von Fehlerzuständen während einer asynchronen Berechnung nutzen.

Für die aufgelisteten Methoden wollen wir ein Beispiel erstellen. Dabei kommen die gleichen Hilfsmethoden wie zuvor zum Einsatz und werden nicht nochmals abgebildet. Zunächst wird asynchron eine rund fünf Sekunden dauernde Methode longRunningCreateMsg() ausgeführt. Deren Verarbeitung kann man durch completeAsync() vorzeitig als beendet markieren. Hier wird als Ergebnis COMPLETE, wie schon im Beispiel in Listing 11.5, mithilfe der Aufrufe von thenAccept() und notifySubscribers() ausgegeben. Die zweite Variante führt wieder die lang dauernde Methode asynchron aus, diesmal kommt es aber nach drei Sekunden zu einem Timeout. Das führt zur Ausgabe von exception occurred: java.util.concurrent.TimeoutException. Die dritte Variante zeigt, wie man bei einem Timeout einen Wert zurückliefern kann. Dies geschieht im Falle eines Timeouts, für den hier zwei Sekunden gewählt wurden. Verbleibt noch die Methode failedFuture(). Hiermit lässt sich das CompletableFuture<T> mit einem Fehler komplettieren. In diesem Fall wird dadurch ALWAYS FAILING ausgegeben.

```
public static void main(final String[] args) throws ExecutionException
{
    CompletableFutureJdk9Example example = new CompletableFutureJdk9Example();

    example.perform();
}

public void perform() throws ExecutionException
{
   CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
                    .completeAsync(() -> "COMPLETE")
                    .thenAccept(this::notifySubscribers);

   CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
                    .orTimeout(3, TimeUnit.SECONDS)
                    .exceptionally(ex -> "exception occurred: " + ex)
                    .thenAccept(this::notifySubscribers);

   CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
                    .completeOnTimeout("TIMEOUT-FALLBACK", 2, TimeUnit.SECONDS)
                    .thenAccept(this::notifySubscribers);

   CompletableFuture.failedFuture(new IllegalStateException())
                    .exceptionally(ex -> {
                           System.out.println("ALWAYS FAILING");
                           return -1;
                     });

   sleepInSeconds(10);  // Auf die Terminierung des CompletableFutures warten
}
```

Listing 11.7 *Ausführbar als* '**COMPLETABLEFUTUREJDK9EXAMPLE**'

Startet man das Programm COMPLETABLEFUTUREJDK9EXAMPLE, so kommt es zu folgenden Ausgaben, wobei die Reihenfolge durch die Nebenläufigkeit abweichen kann:

```
ForkJoinPool.commonPool-worker-9 >>> longRunningCreateMsg
main notifySubscribers: COMPLETE
ForkJoinPool.commonPool-worker-2 >>> longRunningCreateMsg
ForkJoinPool.commonPool-worker-11 >>> longRunningCreateMsg
ALWAYS FAILING
CompletableFutureDelayScheduler notifySubscribers: TIMEOUT-FALLBACK
CompletableFutureDelayScheduler notifySubscribers: exception occurred: java.util
      .concurrent.TimeoutException
ForkJoinPool.commonPool-worker-9 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-2 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-11 <<< longRunningCreateMsg
```

Hier erscheint die Ausgabe des Ergebnisses der letzten Aktion `ALWAYS FAILING` noch vor der Protokollierung der zweiten und dritten Aktion. Insbesondere wird auch deutlich, dass alle länger laufenden Aktionen vollständig abgearbeitet werden – nur der Ergebniswert im `CompletableFuture<T>` wird durch die Aktionen anders als der Rückgabewert der Methode `longRunningCreateMsg` ausgewertet.

11.4.4 Beispiel: Von synchron zu mutlithreaded

Zum Abschluss der Vorstellung der Klasse CompletableFuture<T> möchte ich nochmal genauer auf deren Vorteile eingehen, weil diese Klasse meiner Meinung nach eine Perle des JDKs darstellt.

Anhand des nachfolgenden Beispiels soll gezeigt werden, wie einfach sich eine bestehende synchrone Funktionalität parallelisieren lässt. Dabei hilft es ungemein, wenn die bestehende Funktionalität bereits in Methoden geschnitten ist.

Unabhängig von Parallelisierung sollte man zunächst die Business-Funktionalität sauber in für sich testbare Methoden kapseln. Generell ist es schwieriger, Funktionalität auf Low-Level-Ebene zu parallelisieren als für Building-Blocks in Form von Methoden. Wieso ist das so? Auf der Ebene einzelner oder weniger Anweisungen schleichen sich schnell Fehler bei der Thread-Synchronisierung und Abstimmung der Aufgaben ein. Glücklicherweise bietet CompletableFuture<T> hier Abhilfe, weil auf viel höherer Abstraktionsebene gearbeitet wird. Man muss dazu lediglich die ursprünglichen Aufrufe mit geeigneten Methoden der Klasse CompletableFuture<T> ummanteln, um eine parallele Ausführung zu erzielen. Praktischerweise kümmert sich CompletableFuture<T> um die ansonsten notwendigen und schon angedeuteten lästigen Details der Thread-Kommunikation und -Synchronisierung. Beachten Sie bitte noch als Voraussetzung, dass die Verarbeitungsmethoden nicht mit Seiteneffekten auf gemeinsamen Daten arbeiten dürfen, sondern einem funktionalen Ansatz von Eingabe, Verarbeitung und Ausgabe folgen sollten, wie ich es gleich exemplarisch zeige.

Ausgangslage

Nehmen wir an, wir hätten die Aufgabe, eine bestehende Applikation zu parallelisieren, bei der bereits die folgende Aktionen in Form von Methoden realisiert sind:

- Daten vom Server lesen => retrieveData()
- Auswertung 1 berechnen => processData1()
- Auswertung 2 berechnen => processData2()
- Auswertung 3 berechnen => processData3()
- Ergebnisse in Form eines Dashboards zusammenführen => calcResult()

Um das Beispiel überschaubar zu halten, nehmen wir im Vergleich zu realen Projekten folgende Vereinfachungen vor: Die Daten werden als Liste von Strings modelliert und die Berechnungen liefern lediglich ein Ergebnis vom Typ long. Schließlich wird das Dashboard als String repräsentiert. Das ergibt folgende Realisierung:

```
public void execute()
{
    final List<String> data = retrieveData();
    final Long value1 = processData1(data);
    final Long value2 = processData2(data);
    final Long value3 = processData3(data);

    System.out.println("result: " + calcResult(value1, value2, value3);
}
```

Betrachten wir den Status quo dieser Ausgangsbasis: Die Berechnungen sind vereinfacht und nutzen keine Parallelität und erfordern durch die synchrone Ausführung keine Abstimmung der Aufgaben. Darüber hinaus findet man kein Exception Handling.

Wollten wir das Ganze parallelisieren, so ersparen wir uns besser sowohl die Mühen als auch kaum verständliche und unwartbare Varianten mit `Runnable`, `Callable<T>`, Thread-Pools, `ExecutorService` usw., weil das oft doch ziemlich kompliziert und fehlerträchtig ist.

Variante mit Multithreading

Fragen wir uns kurz: Was muss man ändern, damit eine parallele Verarbeitung mit `CompletableFuture<T>` erfolgt? Basierend auf dem bisher Gelesenen, kommen Sie hoffentlich auf folgende Umsetzung:

```
public void execute() throws InterruptedException, ExecutionException
{
    final CompletableFuture<List<String>> cFData =
                            CompletableFuture.supplyAsync(()->retrieveData());

    // Auswertungen parallel ausführen
    final CompletableFuture<Long> cF1 =
                            cFData.thenApplyAsync(data -> processData1(data));
    final CompletableFuture<Long> cF2 =
                            cFData.thenApplyAsync(data -> processData2(data));
    final CompletableFuture<Long> cF3 =
                            cFData.thenApplyAsync(data -> processData3(data));

    // Einzelergebnisse als Result zusammenführen
    System.out.println("result: " + calcResult(cF1.get(), cF2.get(), cF3.get());
}
```

Wie bildet man Fehler der realen Welt ab?

In der Praxis kommt es bei Verarbeitungen und Berechnungen mitunter zu Exceptions, etwa weil Fehler ab und an auftreten, durch Netzwerkprobleme oder durch Probleme beim Zugriff aufs Dateisystem. Nehmen wir für das Beispiel an, es würde während der Datenermittlung eine `IllegalStateException` ausgelöst: Folglich würde die gesamte Verarbeitung unterbrochen und gestört! Schlimmer wäre das Ganze, wenn wir auf Low-Level-Ebene arbeiten würden, weil sich dort ein Exception Handling sehr mühsam gestaltet. Wünschenswert ist eine einfache Integration des Exception Handlings in den Ablauf.

Praktischerweise bietet die Klasse `CompletableFuture<T>` die Methode `exceptionally()`, die wir zuvor schon in Aktion erlebt haben. Mit dieser lässt sich ein Fallback-Wert bereitstellen. Im Beispiel nutzen wir das, wenn die Daten nicht ermittelt werden können. Ähnliches könnte man einsetzen, wenn eine Berechnung fehlschlägt:

```
final CompletableFuture<List<String>> cfData =
                        CompletableFuture.supplyAsync(()->retrieveData()).
                                            exceptionally(throwable -> List.of());
```

```
final CompletableFuture<Long> cF2 =
                              cfData.thenApplyAsync(data -> processData2(data)).
                                     exceptionally(throwable -> 0L);
```

Dadurch lässt sich die gesamte Berechnung selbst dann noch fortsetzen, wenn der eine oder andere Schritt fehlschlagen sollte.

Wie bildet man Verzögerungen der realen Welt ab?

In der realen Welt kommt es bei Zugriffen auf externe Ressourcen manchmal zu Verzögerungen. Im schlimmsten Fall antwortet ein externer Partner auch gar nicht und man würde gegebenenfalls unendlich warten, wenn ein Aufruf blockierend erfolgt. Wünschenswert ist es in solchen Fällen, Berechnungen mit Timeout abzubrechen. Für unser Beispiel nehmen wir an, dass die Datenermittlung mit `retrieveData()` mitunter einige Sekunden benötigt. Folglich würde die gesamte Verarbeitung gestört!

Erinnern wir uns kurz: Seit JDK 9 bietet die Klasse `CompletableFuture<T>` die folgenden beiden Methoden:

- `orTimeout(long, TimeUnit)` – Die Verarbeitung wird mit einer Exception abgebrochen, falls das Ergebnis nicht innerhalb des Timeouts berechnet wurde.
- `completeOnTimeout(T, long, TimeUnit)` – Falls das Ergebnis nicht innerhalb der angegebenen Zeitspanne berechnet wurde, wird ein Defaultwert zurückgegeben.

Mit deren Hilfe können wir selbst für den Fall, dass die Datenermittlung mitunter mehrere Sekunden dauert, dafür sorgen, dass die Verarbeitung nach spätestens 1 Sekunde abgebrochen wird. Auch für Berechnungen lässt sich ein Fallback-Wert angeben:

```
CompletableFuture<List<String>> cfData =
                CompletableFuture.supplyAsync(()->retrieveData()).
                                  orTimeout(1, TimeUnit.SECONDS).
                                  exceptionally((throwable) -> List.of());

CompletableFuture<Long> cF3 =
                        cfData.thenApplyAsync(data -> processData3(data)).
                               completeOnTimeout(7L, 2, TimeUnit.SECONDS);
```

Dadurch kann die Verarbeitung zeitnah (ohne viel Verzögerung oder gar Blockierung) selbst dann fortgesetzt werden, wenn die Datenermittlung länger dauern sollte. Zudem kann die Berechnung mit einen Fallback-Wert weitergeführt werden, ohne das Ende der Berechnung abzuwarten, selbst wenn der eine oder andere Schritt länger dauern sollte.

Fazit

Anhand eines einfachen, aber praxisnahen Beispiels wurde eine synchrone Verarbeitung in eine asynchrone überführt und dabei die jeweils passenden Methoden aus der Klasse `CompletableFuture<T>` eingesetzt. Im Praxiseinsatz sind für Multithreading

auch komplexere Themen wie Fehlerbehandlung sowie das Beachten von Timeouts wichtig. Diese wurden passend in das Beispiel integriert.

11.5 Reactive Streams und die Klasse `Flow`

Seit JDK 9 werden Reactive Streams[4] durch die Klasse `java.util.concurrent.Flow` und deren innere Interfaces unterstützt. Das wollen wir uns nun anschauen.

11.5.1 Schnelleinstieg Reactive Streams

Themen wie Performance, Skalierbarkeit und Ausfallsicherheit gewinnen heutzutage zunehmend an Bedeutung. Das adressieren Frameworks wie Akka, RxJava, Vert.x und Reactor, indem sie eine asynchrone, eventgetriebene, nicht blockierende Verarbeitung unterstützen – man spricht von Reactive Programming.[5]

Das Essenzielle ist dabei, dass voneinander unabhängig ausgeführte Verarbeitungseinheiten über Events bzw. Messages miteinander kommunizieren. Dazu registrieren sich Subscriber (Empfänger) bei einem Publisher (Sender). Bei der Kommunikation kann es zu Engpässen kommen, wenn ein Publisher zu langsam oder zu schnell arbeitet. Im ersten Fall werden Subscriber eventuell nur teilweise ausgelastet, im zweiten Fall überlastet. Früher hat man als Abhilfe oftmals einen Eingangspuffer aufseiten des Empfängers genutzt. Um Speicherprobleme zu vermeiden, sollte der Eingangspuffer jedoch größenbeschränkt sein. Alternativ kann der Sender auch nur eine gewisse Maximalmenge an Daten verschicken. Dadurch verschwendet man aber potenziell Performance, wenn der Empfänger eine schnellere Abarbeitung bewerkstelligen könnte.

Reactive Streams adressieren die beschriebenen Engpässe im Datenfluss. Dabei hilft das Konzept der ***Backpressure*** (Gegendruck oder Rückstau). Die Idee ist recht simpel: Der Empfänger kann dem Sender mitteilen, wie viele Daten er verarbeiten kann. Der Subscriber fordert aktiv Daten an und der Publisher schickt maximal so viele Daten wie angefordert. Dadurch ermöglichen Reactive Streams eine ***Selbstregulierung***, bei der die Datenverarbeitung an einen möglicherweise temporär langsamen Verarbeiter angepasst wird. Betrachten wir im Anschluss die mit JDK 9 eingeführte Umsetzung.

Abbildung 11-1 *Schematische Zusammenarbeit zwischen Publisher und Subscriber*

[4] `http://www.reactive-streams.org/`

[5] Details dazu finden Sie unter `http://www.reactivemanifesto.org/`.

11.5.2 Reactive Streams im JDK

Java 9 unterstützt Reactive Streams in Form der Klasse `Flow`. Das geschieht relativ leichtgewichtig und basiert auf den vier Interfaces `Publisher<T>`, `Subscriber<T>`, `Subscription` und `Processor<T,R>`, die innerhalb von `Flow` definiert sind. `Publisher<T>` und `Subscriber<T>` implementiert man gewöhnlich unter Zuhilfenahme der Klasse `SubmissionPublisher` selbst und eine `Subscription` hilft beim Austausch von Daten. Schließlich kombiniert ein `Processor<T,R>` sowohl `Publisher<T>` als auch `Subscriber<T>` in einem Interface.

Das Interface `Publisher<T>`

Ein `Publisher<T>` ist ein Veröffentlicher (mitunter auch Erzeuger) von Dingen, die von einem oder mehreren `Subscriber<T>`-Objekten verarbeitet werden. Dazu können sich diese bei dem `Publisher<T>` mit folgender Methode registrieren:

```
@FunctionalInterface
public static interface Publisher<T>
{
    public void subscribe(Subscriber<? super T> subscriber);
}
```

Das Interface `Subscriber<T>`

Ein `Subscriber<T>` ist ein Empfänger von Nachrichten bzw. Daten. Initial wird vom `Publisher<T>` eine Registrierung in Form einer `Subscription` an einen `Subscriber<T>` gesendet und in der Folge werden Daten mit `onNext(T)` übertragen. Zudem lassen sich Fehlersituationen oder das Ende einer Datenübertragung durch spezifische Methoden abbilden. Das Interface `Subscriber<T>` ist wie folgt definiert:

```
public static interface Subscriber<T>
{
    public void onSubscribe(Subscription subscription);
    public void onNext(T item);
    public void onError(Throwable throwable);
    public void onComplete();
}
```

Wie im Listing gezeigt, besitzt der `Subscriber<T>` vier Methoden:

- `onSubscribe(Subscription subscription)` – Dient zur Registrierung und wird vor der eigentlichen Kommunikation aufgerufen.
- `onNext(T item)` – Wird aufgerufen, wenn ein neues Item verfügbar ist.
- `onError(Throwable throwable)` – Für den Fall, dass ein Fehler auftritt, wird diese Methode durch den `Publisher<T>` aufgerufen, um dies dem jeweiligen `Subscriber<T>` mitzuteilen.
- `onComplete()` – Falls die Datenübertragung beendet werden soll, kann ein Aufruf dieser Methode durch den `Publisher<T>` erfolgen.

Das Interface `Subscription`

Eine `Subscription` dient zum Verknüpfen von `Publisher<T>` und `Subscriber<T>` und wird in der Regel durch das Framework erzeugt. Der Basistyp ist wie folgt definiert:

```
public static interface Subscription
{
    public void request(long n);
    public void cancel();
}
```

Subscriber erhalten allerdings nur dann Items, wenn diese explizit angefordert wurden. Dazu dient die `request(long)`-Methode. Mit dem Parameter wird festgelegt, wie viele Daten vom `Producer<T>` angefordert und als Reaktion auf diesen Aufruf maximal geliefert werden dürfen. Weitere Anforderungen können ergänzend per `request(long)` erfolgen. Die Gesamtzahl der per `onNext(T)` gelieferten Items wird kumuliert. Weil der Parameter die maximale Anzahl an Daten, die der `Subscriber<T>` verarbeiten kann oder will, bestimmt, lassen sich der Datenfluss steuern und Überlastsituationen vermeiden. Es ist auch eine unlimitierte Datenübertragung möglich. Dazu muss der Wert `Long.MAX_VALUE` übergeben werden. Bei Bedarf kann die Verarbeitung vom `Subscriber<T>` durch Aufruf von `cancel()` abgebrochen werden.

Das Interface `Processor<T,R>`

Der `Processor<T,R>` kombiniert die beiden Interfaces `Subscriber<T>` sowie `Publisher<R>` und ist folgendermaßen definiert:

```
public static interface Processor<T,R> extends Subscriber<T>, Publisher<R>
{
}
```

Diese Kombination erlaubt es, dass sich damit Verarbeitungsketten nach folgendem Muster zusammenfügen lassen:

Abbildung 11-2 *Zusammenarbeit zwischen Publisher, Processor und Subscriber*

Die Klassen `SubmissionPublisher` und `BufferedSubscription`

Die zuvor beschriebenen Interfaces bilden das Grundgerüst, das implementiert werden muss. Dazu bietet das JDK die Klasse `java.util.concurrent.SubmissionPublisher`. Diese implementiert gemäß ihrem Namen das Interface `Publisher<T>` und hilft dabei, `Subscriber<T>` zu verwalten, insbesondere, um asynchron Daten an registrierte `Subscriber<T>` zu publizieren. Die private statische innere Klasse `BufferedSubscription` implementiert das Interface `Subscription` und ermög-

licht es, Subscriptions zu verarbeiten. Zudem können registrierte Subscriber<T> mithilfe einer Subscription neue Daten anfordern oder die Verarbeitung abbrechen.

Die Klassen und Interfaces im Überblick

Abbildung 11-3 zeigt, wie die zuvor beschriebenen Klassen und Interfaces miteinander in Verbindung stehen. Dieses Klassendiagramm sollte das Verständnis für die Zusammenhänge und die im Anschluss beschriebenen Abläufe bei der Kommunikation erleichtern.

Abbildung 11-3 *Die* Flow*-Interfaces und die Klasse* SubmissionPublisher

Verarbeitungen mit der Klasse Flow

Die vorgestellten Interfaces ermöglichen eine flexible Kommunikation. Eine Variante davon ist in Abbildung 11-4 dargestellt und zeigt einen nummerierten Ablauf:

- Zunächst wird ein Subscriber<T> erzeugt und bei einem Publisher<T> registriert. Zur Komplettierung ruft der Publisher<T> die Methode onSubscribe() aus dem Interface Subscriber<T> auf und übergibt dabei ein Subscription-Objekt. (Aktionsnummer 1, 2 und 2.1)
- Die Kommunikation beginnt, wenn ein Subscriber<T> Daten anfordert, indem er die Methode request(long) aufruft. (Aktionsnummer 2.1.1)
- Die Datenübergabe erfolgt durch Aufruf von onNext(T). Diese Methode von Subscriber<T> darf jedoch vom Publisher<T> nur maximal so oft aufgerufen werden, wie zuvor in request(long) als Wert übergeben wurde – bzw. bei mehrfachen Aufrufen als Summe der Anzahl. (Aktionsnummern 3, 4 und 5)
- Schließlich kann ein Subscriber<T> die Kommunikation durch Aufruf von cancel() beenden. Alternativ kann dies durch den Publisher<T> geschehen, indem für jeden registrierten Subscriber<T> dessen Methode onComplete() aufgerufen wird. Nutzt man die Klasse SubmissionPublisher lässt sich dies einfacher durch einen Aufruf von close() erledigen. (Aktionsnummer 6)

Abbildung 11-4 *Darstellung der Abläufe mit der Klasse* `Flow`

Es sei darauf hingewiesen, dass ein `Subscriber<T>` jederzeit erneut die Methode `request(long)` aufrufen kann, um weitere Daten anzufordern. Das ist selbst dann möglich, wenn der `Publisher<T>` noch gar nicht alle aus dem letzten Aufruf von `request(long)` gewünschten Daten geliefert hat. Somit ist der `Subscriber<T>` nicht daran gebunden, zu warten, bis er alle Datensätze aus dem letzten Request erhalten oder verarbeitet hat. Dies ist ein Mittel, um dafür zu sorgen, dass es für ihn immer genügend Arbeit gibt. Wichtig dabei ist, dass sich die Anzahl der angeforderten Datensätze addiert. Werden beim ersten Mal zehn und danach fünf Datensätze angefordert, so liefert der `Publisher<T>` insgesamt 15 Datensätze, sofern er nicht selbst per `onComplete()` signalisiert, dass er keine Daten mehr an den `Subscriber<T>` liefern kann. Ein anderer Grund, weniger Daten zu liefern als angefordert, ist das Auftreten eines Fehlers, der über `onError(Throwable)` kommuniziert wird. Eine weitere Zusammenarbeit ist dann nicht mehr sinnvoll. Schließlich ist es dem `Subscriber<T>` möglich, beim `Publisher<T>` indirekt über `subscription.cancel()` das Ende der Kommunikation anzufordern.

Im obigen Diagramm habe ich bewusst zwei dickere Pfeile von der `Subscription` auf den `Publisher<T>` eingezeichnet. Tatsächlich ist anhand der Interfaces nicht nachvollziehbar, wie ein `Publisher<T>` darüber informiert wird, dass ein `Subscriber<T>` bei der `Subscription` eine der Methoden `request(long)` oder `cancel()` aufge-

rufen hat. Dies geschieht nur indirekt und muss durch die jeweilige Implementierung sichergestellt werden. Dies wird durch Einsatz der Klasse `SubmissionPublisher` aus dem JDK erleichtert.

11.5.3 Beispiel zur Klasse Flow

Nachdem wir grob die Abläufe kennengelernt haben, wollen wir Reactive Streams zur parallelisierten Ausführung des Zählens von Wörtern in mehreren Dateien einsetzen. Diese Funktionalität habe ich bereits zur Beschreibung einiger Neuerungen in JDK 8 für verschiedene Klassen in meinem Buch »Java 8 – Die Neuerungen« [38] genutzt. Für die Klasse `Flow` haben es Hettel und Tran in ihrem Buch »Nebenläufige Programmierung mit Java« [28] als Beispiel gewählt. Das greife ich hier auf, verändere aber die Implementierung an diversen Stellen.

Die Applikation Für ein besseres Verständnis des Zusammenspiels der einzelnen Klassen beginnen wir mit dem Hauptprogramm. Hier sehen wir sowohl das Erzeugen und den Einsatz von `Publisher<T>` als auch von `Subscriber<T>`. Der `WordPublisher` durchsucht eine übergebene Liste von Dateien nach einem speziellen Wort, hier »`private`«. Die Suche wird durch Aufruf von `performSearch()` gestartet. Bei Auffinden eines passenden Worts wird der zuvor mit `subscribe()` registrierte `WordSubscriber` informiert:

```
public static void main(final String[] args) throws Exception
{
   final WordPublisher finder = new WordPublisher("private", getInputFiles());
   finder.subscribe(new WordSubscriber());
   finder.performSearch();

   Thread.sleep(2_000);  // auf das Ende der Verarbeitung warten
   finder.terminate();
}

private static List<Path> getInputFiles()
{
   final String srcPath = "src/main/resources/";
   return List.of(Paths.get(srcPath + "reactivestreams/WordPublisher.java"),
                  Paths.get(srcPath + "reactivestreams/WordFinderClient.java"));
}
```

Listing 11.8 *Ausführbar als* **'WORDFINDERCLIENT'**

Der Publisher Der `Publisher<T>` wird so implementiert, dass er in einer Menge von Dateien nach einem übergebenen Wort suchen kann. Diese Suche erfolgt parallel mithilfe eines Thread-Pools. Das wichtigste Detail ist aber die im JDK definierte Klasse `SubmissionPublisher`. Mit deren Hilfe kann man Implementierungen des Interface `Publisher<T>` einfach umsetzen: Zum einen bietet diese Klasse die Möglichkeit zur Verwaltung von mehreren `Subscriber<T>`-Objekten und zum anderen kann man damit registrierte `Subscriber<T>` benachrichtigen:

```
public class WordPublisher implements Flow.Publisher<String>
{
    private final String word;
    private final List<Path> paths;
    private final SubmissionPublisher<String> publisher;

    private final ExecutorService executor = Executors.newFixedThreadPool(4);

    public WordPublisher(final String word, final List<Path> paths)
    {
        this.word = word;
        this.paths = Collections.unmodifiableList(paths);
        this.publisher = new SubmissionPublisher<>();
    }

    @Override
    public void subscribe(final Subscriber<? super String> subscriber)
    {
        publisher.subscribe(subscriber);
    }

    public void performSearch() throws InterruptedException
    {
        for (final Path path : paths)
        {
            // Parallele Ausführung
            executor.execute(() ->
            {
               final Stream<String> occurences = findWord(word, path);
               occurences.forEach(line -> publisher.submit("file: " + path +
                                                            " : " + line));
           });
        }
    }

    public void terminate() throws InterruptedException
    {
        // führt zum Aufruf von onComplete()
        publisher.close();

        executor.shutdown();
    }

    private Stream<String> findWord(final String wordToSearch,
                                    final Path path) throws IOException
    {
        try
        {
            final Charset utf8 = StandardCharsets.UTF_8;
            final List<String> lines = Files.readAllLines(path, utf8);

            return lines.stream().filter(line -> line.contains(wordToSearch));
        }
        catch (final IOException e)
        {
            return Stream.of();
        }
    }
}
```

Der Subscriber Der im Listing realisierte `Subscriber<T>` protokolliert alle Methodenaufrufe auf der Konsole. Für diese einfache Funktionalität benötigt man keinen Zugriff auf die `Subscription`. Normalerweise würde man diese in einem Attribut speichern, damit man beliebig Daten anfordern oder die Verarbeitung abbrechen kann. Das lassen wir hier zunächst noch aus und heben uns dies für eine Erweiterung auf:

```
public class WordSubscriber implements Subscriber<String>
{
    @Override
    public void onSubscribe(final Subscription subscription)
    {
        System.out.println(LocalDateTime.now() + " onSubscribe()");

        subscription.request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(final String item)
    {
        System.out.println(LocalDateTime.now() + " onNext(): " + item);
    }

    @Override
    public void onComplete()
    {
        System.out.println(LocalDateTime.now() + " onComplete()");
    }

    @Override
    public void onError(final Throwable throwable)
    {
        throwable.printStackTrace();
    }
}
```

Programmausführung Beim Start des Programms WORDFINDERCLIENT kommt es in etwa zu folgenden Ausgaben, die die Ergebnisse der Suche protokollieren:

```
2020-08-07T14:24:19.214172 onSubscribe()
2020-08-07T14:24:19.216685 onNext(): file: src/main/resources/reactivestreams/
      WordPublisher.java : private final String word;
2020-08-07T14:24:19.216745 onNext(): file: src/main/resources/reactivestreams/
      WordFinderClient.java :     final WordPublisher finder = new WordPublisher(
      "private", getInputFiles());
2020-08-07T14:24:19.216794 onNext(): file: src/main/resources/reactivestreams/
      WordPublisher.java :     private final List<Path> paths;
2020-08-07T14:24:19.216850 onNext(): file: src/main/resources/reactivestreams/
      WordPublisher.java : private final SubmissionPublisher<String> publisher;
22020-08-07T14:24:19.216965 onNext(): file: src/main/resources/reactivestreams/
      WordPublisher.java :     private final ExecutorService executor = Executors
      .newFixedThreadPool(4);
2020-08-07T14:24:19.217016 onNext(): file: src/main/resources/reactivestreams/
      WordFinderClient.java : private static List<Path> getInputFiles()
2020-08-07T14:24:19.217070  onNext(): file: src/main/resources/reactivestreams/
      WordPublisher.java : private Stream<String> findWord(final String
      wordToSearch, final Path path) throws IOException
2020-08-07T14:24:21.184807 onComplete
```

Man erkennt die Start- und Registrierungsphase, auf die die Verarbeitung mit `onNext(T)` folgt. Zum Abschluss wird `onComplete()` aufgerufen.

Ein weiterer Subscriber Nun schauen wir uns eine Variante von `Subscriber<T>` an, die nur die ersten fünf Items konsumiert. Als Abwandlung der sehr einfachen ersten `Subscriber<T>`-Realisierung wird hier die `Subscription` als Attribut definiert, und es lässt sich ein `Consumer<String>` mitgeben, der die Aktionen auf den Items beschreibt:

```
public class First5Subscriber implements Subscriber<String>
{
    private final Consumer<String> consumer;
    private Subscription subscription;
    private int count = 1;

    First5Subscriber(final Consumer<String> consumer)
    {
        this.consumer = consumer;
    }

    @Override
    public void onSubscribe(final Subscription subscription)
    {
        System.out.println("First 5 Subscription: " + subscription);

        this.subscription = subscription;
        this.subscription.request(10);
    }

    @Override
    public void onNext(final String item)
    {
        System.err.println("First 5 " + count + "x onNext(): " + item);

        consumer.accept(item);
        count++;
        if (count > 5)
        {
            subscription.cancel();
        }
    }

    @Override
    public void onComplete()
    {
        System.out.println("onComplete()");
    }

    @Override
    public void onError(final Throwable throwable)
    {
        throwable.printStackTrace();
    }
}
```

Anhand dieses Beispiels lernt man zwei Dinge: Zum einen, wie sich die Logik zur Verarbeitung in Form eines `Consumer<String>` an den `Subscriber<String>` übergeben lässt, und zum anderen, wie mithilfe von `onNext(String)` eine etwas kom-

plexere und flexiblere Verarbeitung realisiert werden kann: Hier wird das Zählen der Einträge sowie der Abbruch durch den `Subscriber<String>` mit einem Aufruf von `subscription.cancel()` bei fünf erhaltenen Einträgen demonstriert.

Dieser `Subscriber<String>` lässt sich problemlos parallel zu dem anderen registrieren. Dazu wandelt man die bereits gezeigte `main()`-Methode wie folgt leicht ab:

```
public static void main(final String[] args) throws IOException,
                                                    InterruptedException
{
    final WordPublisher finder = new WordPublisher("private", getInputFiles());
    finder.subscribe(new WordSubscriber());
    finder.subscribe(new First5Subscriber(System.err::println));
    finder.performSearch();

    TimeUnit.SECONDS.sleep(2); // auf das Ende der Verarbeitung warten

    finder.terminate();
}
```

Listing 11.9 *Ausführbar als* **'WORDFINDERTWOSUBSCRIBERSEXAMPLE'**

Starten Sie das Programm WORDFINDERTWOSUBSCRIBERSEXAMPLE, so produziert dieses folgende zur besseren Nachvollziehbarkeit auf das Wesentliche gekürzte Ausgaben, an denen man die Beschränkung des einen Subscribers auf 5 Callbacks sieht:

```
First 5 Subscription: java.util.concurrent.
      SubmissionPublisher$BufferedSubscription@38a17ecf
First 5 1x onNext(): file: src/main/resources/reactivestreams/WordPublisher
First 5 2x onNext(): file: src/main/resources/reactivestreams/WordFinderClient
First 5 3x onNext(): file: src/main/resources/reactivestreams/WordPublisher
First 5 4x onNext(): file: src/main/resources/reactivestreams/WordPublisher
First 5 5x onNext(): file: src/main/resources/reactivestreams/WordFinderClient
```

11.5.4 Fazit

Dieses Unterkapitel hat einen Einstieg in Reactive Streams gegeben und die grundlegenden Konzepte kurz vorgestellt. Sie sollten mit diesem Wissen in der Lage sein, erste eigene Experimente zu starten. Richtig spannend wird das Ganze, wenn man größere Verarbeitungsketten mithilfe mehrerer `Publisher<T>`, `Processor<T,R>` und `Subscriber<T>` beschreibt. Leider bietet die Implementierung der Reactive Streams im JDK dazu kein API. Unter anderem deshalb sind Frameworks wie RxJava (`http://reactivex.io/`) oder Spring Reactor (`https://projectreactor.io/`) die bessere Alternative.

11.6 Weiterführende Literatur

Da die Themen Multithreading und Nebenläufigkeit recht komplex und vielschichtig sind, empfiehlt es sich, weitere Quellen zu konsultieren. Weiterführende Informationen finden Sie in folgenden Büchern:

- »**Parallele und verteilte Anwendungen in Java**« von Rainer Oechsle [59]
  Dieses Buch gibt einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java. Es wird regelmäßig aktualisiert und behandelt in der 5. Auflage somit auch aktuellere Themengebiete.
- »**Java Concurrency in Practice**« von Brian Goetz et al. [24]
  Dieses Buch bietet eine sehr umfassende und fundierte Beschreibung zum Thema Multithreading. Dort werden auch die Erweiterungen der Concurrent Collections in JDK 6 behandelt.
- »**Nebenläufige Programmierung mit Java**« von Jörg Hettel und Manh Tien Tran [28]
  Dieses aktuelle Buch gibt einen fundierten Einstieg in die nebenläufige Programmierung mit Java und behandelt insbesondere auch die mit Java 8 neu hinzugekommenen Sprachfeatures. Zudem gibt es einen kurzen Ausblick auf die Reactive Streams in Java 9.

12 Fortgeschrittene Java-Themen

In diesem Kapitel stelle ich einige fortgeschrittene Java-Themen vor. Abschnitt 12.1 beginnt mit einem in die Sprache integrierten, ***Reflection*** genannten Mechanismus. Dieser ermöglicht es, zur Laufzeit Informationen zu Klassen, Objekten usw. zu ermitteln. Es ist sogar möglich, neue Instanzen von zum Kompilierzeitpunkt unbekannten Klassen zu erzeugen. Daran schließt sich eine Darstellung von ***Annotations*** zur Angabe von Zusatzinformationen im Sourcecode an. Abschnitt 12.2 gibt sowohl eine Einführung in die Annotations des JDKs als auch eine Beschreibung, wie man eigene Annotations definieren und auswerten kann. Abschnitt 12.3 zeigt die Möglichkeiten zur Konvertierung von Objekten in eine Folge von Bytes und zurück. Dies wird ***Serialisierung*** genannt und über das Interface `java.io.Serializable` als Funktionalität aktiviert. In Java muss man sich als Entwickler gewöhnlich kaum oder gar nicht um das Speichermanagement und insbesondere die Freigabe von Objekten kümmern. Abschnitt 12.4 erläutert das in die JVM integrierte, automatische Speichermanagement – die ***Garbage Collection***. Vervollständigt wird dieses Kapitel durch die Beschreibung von dynamischen Proxys in Abschnitt 12.5. Damit lassen sich Klassen zur Laufzeit (daher dynamisch) anhand der Angabe einer Menge zu erfüllender Interfaces erstellen. Den krönenden Abschluss bildet der Abschnitt 12.6 mit der Vorstellung des HTTP/2-API und insbesondere einem Real-World-Beispiel, in dem diverse Techniken und APIs gewinnbringend kombiniert werden, um per REST Wechselkurse abzufragen.

12.1 Crashkurs Reflection

Die Technik ***Reflection*** erlaubt es einem Programm, verschiedene Informationen über sich selbst herauszufinden und zur Laufzeit Instanzen von Klassen zu erzeugen.

Bei Reflection handelt es sich um eine fortgeschrittene Technik, die man bei der normalen Anwendungsentwicklung seltener verwendet. Es gibt jedoch einige Anwendungsfälle, die nur durch den Einsatz von Reflection umzusetzen sind. Es existieren unter anderem folgende Einsatzgebiete:

- **Erweiterbarkeit** – Eine Applikation kann mit Reflection sogar solche Klassen instanziieren, die zum Zeitpunkt der Kompilierung unbekannt sind. Zur Laufzeit wird nur ihr voll qualifizierter Name benötigt. Ausgehend davon können alle möglichen Bestandteile von Klassen, also Methoden, Attribute usw., abgefragt werden.

- **Class-Browser, Debugger und Tools zum Testen** – Mithilfe von Reflection erhält ein Programm Zugriff auf alle Bestandteile einer Klasse. Selbst der Zugriff auf private Attribute und Methoden ist möglich.

Neben den genannten Einsatzgebieten sollte Reflection nur in speziellen Situationen und für den Fall, dass es wirklich notwendig und sinnvoll ist, verwendet werden, beispielsweise wenn zur Laufzeit Informationen über das Programm selbst benötigt werden oder dessen Verhalten geändert werden soll. Denken Sie an folgenden Hinweis: ***Gibt es eine Lösung ohne Reflection, so wähle diese***, weil sie in der Regel klarer und verständlicher ist.

Hinweise zum Einsatz

Vor einem möglichen Einsatz sollte man die im Folgenden genannten Besonderheiten beachten, die sich nachteilig auswirken können:

- Bei der Verwendung von Reflection werden Informationen erst zur Laufzeit ermittelt. Das verursacht etwas Overhead, wodurch Methodenaufrufe, Attributzugriffe usw. minimal langsamer als »normale« Aufrufe ausgeführt werden.
- Aufrufe per Reflection werden möglicherweise durch nicht ausreichende Berechtigungen verhindert. Grundsätzlich ist es korrekt, eine nicht berechtigte Ausführung zu unterbinden. Die dazu notwendigen Berechtigungen werden über eine Instanz der Klasse `java.lang.SecurityManager`[1] kontrolliert. Zur Laufzeit können unterschiedliche Instanzen der Klasse `SecurityManager` aktiv sein und somit das Programmverhalten unerwartet verändern.
- Ein Nachteil beim Einsatz von Reflection ist, dass man sowohl Kapselung als auch einiges von der durch den Compiler garantierten Sicherheit aufgibt. Insbesondere kann der Sichtbarkeitsschutz privater Bestandteile von Klassen ausgehebelt werden, wodurch Interna sichtbar werden. Außerdem sind Abhängigkeiten zwischen Klassen schwieriger zu ermitteln, da diese textuell als Klassen-, Methoden- oder Attributnamen beschrieben sind.

Mit diesen Hinweisen im Hinterkopf wollen wir Reflection zumindest so weit kennenlernen, dass ein grundlegendes Verständnis vorhanden ist. Es gibt wenig Literatur, die sich explizit mit diesem Thema auseinandersetzt. Eine Ausnahme bildet das Buch »Java Reflection in Action« von Ira R. und Nate Forman [19].

[1] Mithilfe einer Spezialisierung der abstrakten Klasse `SecurityManager` können verschiedene Berechtigungen erteilt oder verweigert werden, etwa Zugriffe auf Dateien. Existiert keine Berechtigung, so wird eine `java.lang.SecurityException` ausgelöst.

12.1.1 Grundlagen

Damit eine Untersuchung eines Programms überhaupt möglich ist, müssen spezielle Informationen über Klassen und Objekte existieren, die ***Metadaten*** oder ***Metainformationen*** genannt werden. Diese sind im JDK durch Klassen modelliert, z. B. folgende:

- `java.lang.Class<T>` – Metadaten für Klassen, etwa Klassenname, implementierte Interfaces, Methoden, Attribute usw.
- `java.lang.reflect.Field` – Metadaten für Attribute, etwa Typ, Name, Sichtbarkeit usw.
- `java.lang.reflect.Method` – Metadaten für Methoden, etwa Name, Parameter, Sichtbarkeit usw.

Den Startpunkt für die Programmierung mit Reflection bildet immer eine Instanz der Klasse `java.lang.Class<T>`, über die man andere Metadaten beziehen kann. Beschäftigen wir uns also zunächst mit folgender Frage: Wie erhält man Zugriff auf die Metainformationen einer Klasse, also das zugehörige `Class<T>`-Objekt? Es gibt drei verschiedene Möglichkeiten, die Metadaten einer Klasse zu ermitteln:

1. Besitzt man einen voll qualifizierten Klassennamen, so erhält man über einen Aufruf der statischen Methode `Class.forName(String)` ein `Class<T>`-Objekt:

   ```
   // kein Import und kein Class-Loading vor dem ersten Zugriff
   final Class<?> clazz = Class.forName("packagePath.ClassName");
   ```

   Der Name der gewünschten Klasse wird dabei in Form eines Strings übergeben. Dies kann man beispielsweise nutzen, wenn der konkrete Typ zur Kompilierzeit noch unbekannt ist. Dadurch lassen sich zur Laufzeit Klassennamen etwa aus einer Konfigurationsdatei einlesen und verwenden. Im einsetzenden Sourcecode existieren keine `import`-Anweisungen und keine direkten Verweise auf die entsprechenden Klassen. Dies vermeidet wiederum ein Laden der Klassenbeschreibung, wie dies bei einem `import` implizit ausgelöst wird. ***Negativ an der Übergabe eines Klassennamens als String ist, dass es leicht zu Fehlern kommt***. Dies können Tippfehler im Programm selbst oder aber in den Konfigurationsdaten sein.

2. Durch Aufruf der Methode `getClass()` kann die Klasseninformation zu einem bestehenden Objekt wie folgt ermittelt werden:

```
// Zugriff auf ein Class-Objekt über ein Objekt
final Class<?> clazz = obj.getClass();
```

3. Wenn man den konkreten Typ kennt, erhält man über das statische Attribut `.class` Zugriff auf das `Class<T>`-Objekt:

```
// Es wird ein Import nötig und die Klasse wird gegebenenfalls geladen.
Class<?> clazz = MyClass.class;
```

Hinweis: Notation von Arrays in `Class.forName()`

Zum Ermitteln der Klassenbeschreibung für Arrays ist eine etwas kryptische Notation zu verwenden. Im folgenden Beispiel wird das für die Konstruktion eines zweidimensionalen `int`-Arrays gezeigt. Die führenden '`[`'-Zeichen beschreiben die Anzahl der Array-Dimensionen. Für den Typ `int` wird das Typkürzel '`I`' verwendet. Für Referenzvariablen ist dabei zusätzlich Folgendes zu beachten: Das Typkürzel ist ein '`L`' und erfordert einen voll qualifizierten Klassennamen, der durch ein Semikolon abgeschlossen wird – nachfolgend für die Klasse `java.lang.String` gezeigt:

```
// Zugriff auf den Typ „zweidimensionales int-Array": [[I
final Class<?> intintArrayClazz = Class.forName("[[I");
System.out.println(int[][].class);
System.out.println(intintArrayClazz == int[][].class);

// Zugriff auf den Typ „eindimensionales String-Array": [Ljava.lang.String;
final Class<?> stringArrayClazz = Class.forName("[Ljava.lang.String;");
System.out.println(String[].class);
System.out.println(stringArrayClazz == String[].class);
```

Listing 12.1 *Ausführbar als* '**REFLECTIONARRAYCLASSESEXAMPLE**'

Typkürzel Die Typkürzel werden nicht nur für Angaben in `Class.forName()` verwendet, sondern auch bei Ausgaben von Arrays mit `toString()` erzeugt. Bei der Notation repräsentiert die Anzahl der '`[`' die Array-Dimensionen. Anschließend wird der Typ der gespeicherten Elemente durch ein Buchstabenkürzel gekennzeichnet. Das große '`L`' steht für eine Referenzvariable (Klasse oder Interface). Viele weitere Kürzel sind intuitiv und folgen den Anfangsbuchstaben der Typen, etwa '`B`' für `byte`, '`I`' für `int` usw. Allerdings wird für den Typ `long` aufgrund des Konflikts zur Abkürzung für Referenzen der Buchstabe '`J`' verwendet. Auch für den Typ `boolean` musste mit '`Z`' wegen der Kollision mit '`B`' für `byte` ein anderer Buchstabe genutzt werden. Details finden Sie unter `http://download.oracle.com/javase/8/docs/api/java/lang/Class.html`.

Der erste Methodenaufruf per Reflection

Beginnen wir die Entdeckungsreise von Reflection mit einem Beispiel. Nehmen wir an, wir wollen einen Methodenaufruf von `equals(Object)` – das ist normalerweise nicht sinnvoll und geschieht hier nur zur Demonstration – dynamisch mit Reflection ausführen. Für Methodenaufrufe sind generell immer folgende Schritte notwendig:

1. Ermitteln der benötigten Informationen zur bereitstellenden Klasse
2. Ermitteln der benötigten Informationen zur gewünschten Methode
3. Dynamischer Aufruf der gewünschten Methode
4. Behandlung von Fehlersituationen

Der durch die obigen Schritte beschriebene Aufrufmechanismus wird in der folgenden Methode `callEquals(Person, Object)` gekapselt:

```
private static boolean callEquals(final Person person, final Object otherObject)
{
    final String methodName = "equals";
    final Class<?>[] parameterTypes = new Class<?>[] { Object.class };
    final String methodInfo = createMethodInfo(methodName, parameterTypes);

    // Schritt 1: Ermitteln der Klasseninformation
    final Class<?> clazz = person.getClass();
    try
    {
        // Schritt 2: Ermitteln der Methode
        final Method equalsMethod = clazz.getMethod(methodName, parameterTypes);

        // Schritt 3: Aufruf der Methode
        final Object[] parameters = new Object[] { otherObject };
        final Object result = equalsMethod.invoke(person, parameters);
        return Boolean.valueOf(result.toString());
    }
    // Schritt 4: Behandlung sämtlicher durch Reflection möglicher Exceptions
    catch (final NoSuchMethodException e)
    {
        // Es gibt keine solche Methode
        throw new IllegalStateException(clazz.getName() + " does not support " +
                                        methodInfo);
    }
    catch (final SecurityException e)
    {
        // Keine Erlaubnis, auf die Methode zuzugreifen
        throw new IllegalStateException(clazz.getName() + " insufficent " +
                                  " security rights to access " + methodInfo);
    }
    catch (final IllegalAccessException e)
    {
        // Kein Zugriff auf die Definition der Methode (im .class-File)
        throw new IllegalStateException(clazz.getName() + " can't access " +
                                        methodInfo);
    }
    catch (final IllegalArgumentException e)
    {
        // Ungültige Parameter beim Aufruf
        throw new IllegalStateException(clazz.getName() + " parameters are " +
                    "invalid " + methodInfo);
    }
    catch (final InvocationTargetException e)
```

```
    {
        // Ausführung der Methode löst eine Exception aus
        throw new IllegalStateException(clazz.getName() + " exception in " +
                                        methodInfo);
    }
}

private static String createMethodInfo(final String methodName,
                                       final Class<?>[] parameterTypes)
{
    return "method: " + methodName + "(" + Arrays.toString(parameterTypes) +")";
}
```

Listing 12.2 *Ausführbar als* '**REFLECTIONCALLEQUALSEXAMPLE**'

Dieses Beispiel verdeutlicht zwei wesentliche Nachteile beim Einsatz von Reflection:

1. **Es entsteht viel Sourcecode:** Ohne die Neuerungen aus JDK 7 werden ca. 40 – 50 Zeilen benötigt, um lediglich *einen* Methodenaufruf zu kapseln.
2. **Es sind viele unterschiedliche Fehlersituationen zu behandeln:** Bereits im Beispiel wird der Aufwand für die Fehlerbehandlung deutlich, selbst wenn diese wie hier nur rudimentär erfolgt: Exceptions werden lediglich mit Informationen angereichert und als `IllegalStateException` weiter propagiert. Seit JDK 7 sind die Nachteile ziemlich entschärft, wie es der folgende Praxishinweis beschreibt.

Hinweis: Varianten der Fehlerbehandlung bei Reflection seit JDK 7

Oftmals wird man die unterschiedlichen Exceptions nicht jeweils einzeln behandeln wollen. Ein allgemeines `catch (Exception)` ist keine geeignete Alternative (vgl. Abschnitt 16.3.3). Zwar gibt es Fälle, in denen man möglichst differenziert auf Fehler reagieren muss – für Reflection gilt dies eher selten. Seit JDK 7 hilft die Klasse `java.lang.ReflectiveOperationException` dabei, verschiedene, durch Reflection verursachte Exceptions gesamthaft behandeln zu können. Zum anderen lässt sich eine Menge von Exceptions mit dem Sprachfeature Multi Catch gleichartig abfangen.

12.1.2 Zugriff auf Methoden und Attribute

Hat man ein `Class<T>`-Objekt ermittelt, so kann man davon ausgehend weitere Metaobjekte ausfindig machen, die unter anderem Zugriff auf Attribute, Methoden, Konstruktoren, Basisklassen und implementierte Interfaces bieten. Außerdem erhält man Zugriff auf Annotations. Auf diese gehe ich im nachfolgenden Abschnitt 12.2 ein.

Methoden ermitteln

Im einführenden Beispiel haben wir ein `Method`-Objekt durch Aufruf der Methode `getMethod()` erhalten. Diese besitzt eine kleine Einschränkung. Sie erlaubt es lediglich, auf direkt in dieser Klasse definierte, öffentliche Methoden zuzugreifen. Teilweise sollen aber Methoden anderer, d. h. nicht öffentlicher Sichtbarkeiten oder auch solche von Basisklassen aufgerufen werden. Für beide Fälle ist etwas mehr Programmieraufwand nötig: Möchte man alle in einer Klasse definierten Methoden unabhängig von der Sichtbarkeit ausfindig machen, bietet das `Class<T>`-Objekt die Methode `getDeclaredMethod(String, Class<?>...)`. Sollen alle verfügbaren Methoden aller Klassenbestandteile (also inklusive aller Basisklassen) ermittelt werden, so muss die Methode `getDeclaredMethod(String, Class<?>...)` so lange für die von `getSuperclass()` zurückgelieferte Superklasse aufgerufen werden, bis entweder ein passendes `Method`-Objekt gefunden wurde oder aber die Suche erfolglos verläuft (was spätestens dann der Fall ist, wenn die Klasse `Object` erreicht wurde und dort keine passende Methode existiert, denn dann gibt es keine Superklasse mehr, die durchsucht werden könnte).

Anstatt diese Suchfunktionalität in der eigentlichen Applikation selbst zu programmieren, sollte man besser eine Hilfsmethode `findMethod(Class<?>, String, Class<?>...)` in einer Utility-Klasse `ReflectionUtils` wie folgt implementieren:

```
public static Method findMethod(final Class<?> clazz, final String methodName,
                                final Class<?>... parameterTypes)
{
    Objects.requireNonNull(methodName, "methodName must not be null");
    Objects.requireNonNull(parameterTypes, "parameterTypes must not be null");

    // Abbruch der Rekursion
    if (clazz == null)
        return null;

    try
    {
        return clazz.getDeclaredMethod(methodName, parameterTypes);
    }
    catch(final NoSuchMethodException ex)
    {
        // rekursive Suche in Superklasse
        return findMethod(clazz.getSuperclass(), methodName, parameterTypes);
    }
}
```

Für den Einsatz in eigenen Applikationen ist die Definition einer weiteren Hilfsmethode `getAllMethods(Class<?>)` praktisch, die Zugriff auf alle in der Klasse und in Superklassen definierten Methoden beliebiger Sichtbarkeiten bietet. Diese nutzt die Methode `getDeclaredMethods()` der Klasse `Class<T>`, um alle verfügbaren Methoden innerhalb eines Klassenbestandteils ausfindig zu machen. Durch den rekursiven Aufruf für alle Superklassen lassen sich dann alle Methoden einer Klasse ermitteln:

```
public static Method[] getAllMethods(final Class<?> clazz)
{
    Objects.requireNonNull(clazz, "class must not be null");

    final List<Method> methods = new ArrayList<>();
    methods.addAll(Arrays.asList(clazz.getDeclaredMethods()));

    if (clazz.getSuperclass() != null)
    {
        // rekursive Suche in Superklasse
        methods.addAll(Arrays.asList(getAllMethods(clazz.getSuperclass())));
    }

    return methods.toArray(new Method[0]);
}
```

Achtung: Fallstrick von `getMethod()`

Gibt man beim Aufruf von `getMethod(String)` versehentlich den Namen der zu ermittelnden Methode mit Klammern an, so kann die Methode nicht gefunden werden! Es wird `null` anstelle eines `Method`-Objekts zurückgeliefert.

Attribute ermitteln

Wenn man zur Laufzeit auf Attribute einer Klasse zugreifen muss, dann liefert die Methode `getField(String)` bei Übergabe des Attributnamens ein zugehöriges `Field`-Objekt. Ebenso wie beim Zugriff auf Methoden erfolgt hier eine Einschränkung auf öffentliche Attribute, die in der Klasse selbst definiert sind. Analog kann man mit der Methode `getDeclaredField(String)` Attribute beliebiger Sichtbarkeit bestimmen, die innerhalb der Klasse definiert sind. Eine Suche nach Attributen in Basisklassen kann, wie für Methoden dargestellt, rekursiv programmiert werden. Dies wird hier jedoch nicht nochmals gezeigt.

Hinweis: Nicht intuitive Namensgebung

Die Methoden `getMethod(String)` und `getField(String)` bieten Zugriff nur auf öffentliche Methoden und Attribute. Demnach wären `getPublicMethod()` und `getPublicField()` gelungenere Namen gewesen. Das gilt analog für die Namensgebung der Bulk-Methoden `getMethods()` und `getFields()`, die Zugriff auf alle öffentlichen Methoden bzw. Attribute bieten.

Hilfsfunktionalität zur Ausgabe von Informationen zu Klassen

Nachdem wir nun ein Basiswissen zu Reflection aufgebaut und bereits einige Hilfsmethoden geschrieben haben, wollen wir ein Programm erstellen, das beliebige Klassen untersuchen kann. Es soll Informationen zu Superklassen, implementierten Interfaces, Konstruktoren, Attributen und Methoden sowie die Annotations für diese ausgeben.

Dazu implementieren wir zunächst fünf weitere Hilfsmethoden in unserer Utility-Klasse `ReflectionUtils`:

```
public static void printCtorInfos(final Constructor<?> ctor)
{
    Objects.requireNonNull(ctor, "ctor must not be null");

    System.out.println(Modifier.toString(ctor.getModifiers()) + " " +
                       ctor.getName() +
                       buildParameterTypeString(ctor.getParameterTypes()));
    printAnnotations(ctor.getAnnotations());
}

public static void printMethodInfos(final Method method)
{
    Objects.requireNonNull(method, "method must not be null");

    System.out.println(Modifier.toString(method.getModifiers()) + " " +
                       method.getReturnType() + " " + method.getName() +
                       buildParameterTypeString(method.getParameterTypes()));
    printAnnotations(method.getAnnotations());
}

public static void printFieldInfos(final Field field)
{
    Objects.requireNonNull(field, "field must not be null");

    System.out.println(Modifier.toString(field.getModifiers()) + " " +
                       field.getType() + " " + field.getName());
    printAnnotations(field.getAnnotations());
}

public static String buildParameterTypeString(final Class<?>[] parameterTypes)
{
    Objects.requireNonNull(parameterTypes, "parameterTypes must not be null");

    if (parameterTypes.length > 0)
        return "(" + Arrays.toString(parameterTypes) + ")";

    return "()";
}

public static void printAnnotations(final Annotation[] annotations)
{
    Objects.requireNonNull(annotations, "annotations must not be null");

    if (annotations.length > 0)
        System.out.println("Annotations: " + Arrays.toString(annotations));
}
```

Beispiel: Untersuchung einer Klasse Betrachten wir die obigen Hilfsmethoden nun im Einsatz und analysieren exemplarisch die Klasse `String` des JDKs:

```
public static void main(final String[] args)
{
    inspectClass(String.class);
}

private static void inspectClass(final Class<?> clazz)
{
    System.out.println("Untersuchte Klasse: " + clazz.getCanonicalName());
    System.out.println("Superklasse:        " + clazz.getSuperclass());
    System.out.println("Interfaces:         " + Arrays.toString(clazz.
          getInterfaces()));

    // Zugriff und Ausgabe der Konstruktoren
    final Constructor<?>[] ctors = clazz.getDeclaredConstructors();
    System.out.println("\nKonstruktoren: ");
    for (final Constructor<?> ctor : ctors)
        ReflectionUtils.printCtorInfos(ctor);

    // Zugriff und Ausgabe der Attribute
    System.out.println("\nAttribute: ");
    for (final Field field : clazz.getDeclaredFields())
        ReflectionUtils.printFieldInfos(field);

    // Zugriff und Ausgabe aller Methoden
    System.out.println("\nAlle Methoden: ");
    for (final Method method : ReflectionUtils.getAllMethods(clazz))
        ReflectionUtils.printMethodInfos(method);
}
```

Listing 12.3 *Ausführbar als* '**INSPECTIONEXAMPLE**'

Das Programm INSPECTIONEXAMPLE produziert folgende, gekürzte Ausgabe:

```
Untersuchte Klasse: java.lang.String
Superklasse:        class java.lang.Object
Interfaces:         [interface java.io.Serializable, interface java.lang.
      Comparable, interface java.lang.CharSequence]

Konstruktoren:
public java.lang.String([class [B])
public java.lang.String([class [B, int, int])
// ...
Attribute:
private final class [B value
private final byte coder
// ...
Alle Methoden:
 class [B value()
public boolean equals([class java.lang.Object])
// ...
```

Eigenschaften ermitteln

Für Objekte vom Typ `Method` bzw. `Field` erlaubt die Methode `getModifiers()` verschiedene Eigenschaften in Form eines `int`-Werts abzufragen. Bei dessen Auswertung hilft die Klasse `java.lang.reflect.Modifier`, beispielsweise zur menschenlesba-

ren Ausgabe mittels der statischen Hilfsmethode `toString(int)`. Die Klasse bietet zudem statische Methoden zum Auslesen einzelner Eigenschaften, etwa der Sichtbarkeit. Ebenso sind Eigenschaften wie `static`, `final` oder `abstract` von Interesse. Auch für Multithreading relevante Angaben können ermittelt werden.

- `isPublic(int)`, `isProtected(int)` und `isPrivate(int)`
- `isStatic(int)`, `isFinal(int)` und `isAbstract(int)`
- `isSynchronized(int)` bzw. `isVolatile(int)`

Hinweis: Google Guava als Hilfestellung

Exemplarisch für die Ermittlung von Eigenschaften möchte ich darauf hinweisen, dass Google Guava hier mit der Klasse `Invokable` eine Erleichterung der Lesbarkeit bietet, und dies sowohl für Methoden als auch Konstruktoren, was bei Nutzung der JDK-Möglichkeiten meistens zu Code-Duplikation führt. Das Beispiel entstammt der Guava-Dokumentation: Es soll per Reflection festgestellt werden, ob eine Methode überschreibbar ist. Das Ganze erfordert folgenden Sourcecode mit dem JDK:

```
final boolean isOverridable = !((Modifier.isFinal(method.getModifiers())
                               || Modifier.isPrivate(method.getModifiers())
                               || Modifier.isStatic(method.getModifiers())
            || Modifier.isFinal(method.getDeclaringClass().getModifiers()))
```

Nutzt man dagegen `Invokable` wird daraus kurz und knackig:

```
final boolean isOverridable = Invokable.from(method).isOverridable();
```

12.1.3 Spezialfälle

In diesem Abschnitt werden einige Besonderheiten beim Aufruf von Konstruktoren und bei der Übergabe und Verwendung primitiver Datentypen mit Reflection vorgestellt.

Aufruf von Konstruktoren

Sollen Objektkonstruktionen per Reflection erfolgen, so kann man mithilfe der Methode `newInstance()` einen Konstruktor aufrufen. Zur Ausführung des Defaultkonstruktors wird `newInstance()` ohne Übergabeparameter direkt für das `Class<T>`-Objekt aufgerufen. Die Ausführung eines speziellen Konstruktors ist etwas komplizierter und im Ablauf ähnlich zu einem Methodenaufruf per Reflection. Zunächst wird über die Methode `getDeclaredConstructor(Class<?>...)` der gewünschte Konstruktor mit passender Signatur als `Constructor`-Objekt ermittelt. Anschließend kann dann ein Aufruf an `newInstance(Object...)` dieses `Constructor`-Objekts erfolgen.

Übergabe von primitiven Datentypen an Konstruktoren oder Methoden

Bisher haben wir beim Aufruf von Konstruktoren bzw. Methoden außer Acht gelassen, dass dort neben Referenzvariablen häufig auch primitive Datentypen genutzt werden. Zum Auffinden der passenden Konstruktoren bzw. Methoden müssen die Typen der Parameter jedoch als `Class<T>`-Objekt übergeben werden. Für jeden primitiven Datentyp existieren korrespondierende `Class<T>`-Objekte, etwa `int.class`, `long.class` oder für Arrays z. B. `int[].class`. Variable Argumentlisten, Varargs, werden dabei wie Arrays angegeben.

Beim Aufruf von Konstruktoren oder Methoden können Werte primitiver Typen per Reflection nicht direkt verwenden, da die Parameterübergabe in Form von `Object`-Referenzen erfolgt. Daher müssen primitive Datentypen in entsprechende Wrapper-Klassen umgewandelt werden. Das geschieht hier durch Auto-Boxing.

Objektkonstruktion und primitive Typen am Beispiel

Wir wollen das erworbene Wissen nutzen, um verschiedene Konstruktoraufrufe für die Klasse `String` des JDKs auszuführen. Zunächst wird das `Class<T>`-Objekt ermittelt und mit dessen Defaultkonstruktor erzeugt. Es soll außerdem der Konstruktor `String(char value[], int offset, int count)` aufgerufen werden.

Zunächst muss ein `Constructor`-Objekt mit passender Signatur gefunden werden. Zur Demonstration erfolgt hier ein Aufruf mit den Übergabewerten "`a Test`" als `char[]` und zwei `int`-Werten. Diese entsprechen dem Startindex 2 sowie der Länge 4. Dadurch wird der Text "`Test`" aus dem `char[]` extrahiert:

```
public static void main(final String[] args)
{
   try
   {
      final Class<?> stringClass = Class.forName("java.lang.String");

      // Aufruf des Defaultkonstruktors
      final String stringInstance1 = (String) stringClass.newInstance();
      System.out.println("String 1 = '" + stringInstance1 + "'");

      // Suche den Konstruktor String(char[], int, int)
      final Class<?>[] parameterTypes = { char[].class, int.class, int.class };
      final Constructor<?> ctor = stringClass.getDeclaredConstructor(
                                                          parameterTypes);

      // Aufruf des Konstruktors String(char[], int, int)
      final char[] input = { 'a', ' ', 'T', 'e', 's', 't' };
      final String stringInstance2 = (String) ctor.newInstance(input, 2, 4);
      System.out.println("String 2 = '" + stringInstance2 + "'");
   }
   // Behandlung sämtlicher durch Reflection möglicher Exceptions
   catch (final ReflectiveOperationException e)
   {
      throw new IllegalStateException("can't execute constructor: ", e);
   }
}
```

Listing 12.4 *Ausführbar als* '**REFLECTIONCTOREXAMPLE**'

Normalerweise führt der Einsatz von Reflection zu einem erhöhten Aufwand bei der Fehlerbehandlung. In diesem Fall wären sieben verschiedene Typen von Exceptions zu behandeln. Bekanntermaßen ist seit JDK 7 die `ReflectiveOperationException` verfügbar, die eine One-for-All-Behandlung erlaubt. Das ist aus Gründen der Übersichtlichkeit praktisch und weil man die Fehlersituation für Reflection in der Regel nicht unterscheiden möchte. Für den seltenen Fall, dass man einzelne Exceptions doch spezifisch behandeln möchte, verbleiben mehrere `catch`-Blöcke.

Zugriff auf private Dinge und Auslesen von Werten von Attributen

Mit Reflection werden im Normalfall die Sichtbarkeiten beachtet, wie sie im Sourcecode definiert sind. Ist ein Zugriff auf private Attribute oder Methoden anderer Klassen gewünscht, so kann der Sichtbarkeitsschutz durch einen Aufruf von `setAccessible(true)` deaktiviert werden.[2] Ohne dies führen Zugriffe auf private Attribute oder Methoden zu einer `java.lang.IllegalAccessException`.

Den Aufruf von Konstruktoren und Methoden sowie den Zugriff auf Attribute haben wir bereits kennengelernt. Nun zeige ich, wie man per Reflection auf deren Wertebelegung zugreifen kann. Zunächst erhält man durch einen Aufruf von `getField(String)` Zugriff auf ein `Field`-Objekt, das das namentlich übergebene Attribut repräsentiert. Um den Wert eines Attributs eines speziellen Objekts auszulesen, bietet die Klasse `Field` die Methode `get(Object)`. Als Parameter ist die Referenz des Objekts zu übergeben, dessen Attributwert man ermitteln möchte. Für statische Attribute wird statt einer konkreten Objektreferenz der Wert `null` übergeben.

Folgendes Beispiel zeigt den Zugriff auf ein öffentliches, ein statisches und ein privates Attribut. Zur Demonstration der Abfrage von Eigenschaften und Annotations ist das Attribut `value` als `volatile` und `@Deprecated` gekennzeichnet:

```
public final class AttributeAccessExample
{
    public static long instanceCounter = 0;
    @Deprecated public volatile int value;
    private String description = "Hello World";

    public static void main(final String[] args)
    {
        try
        {
            final AttributeAccessExample obj = new AttributeAccessExample();
            final Class<?> clazz = obj.getClass();

            // Zugriff auf das Attribut 'value'
            final Field field = clazz.getField("value");

            // Zugriff auf den Wert des Attributs 'value'
            final Object attributeValue = field.get(obj);
            System.out.println("value = " + attributeValue);

            // Zugriff auf die Annotation des Attributs 'value'
            ReflectionUtils.printAnnotations(field.getAnnotations());
```

[2] Seit Java 9 kann im Kontext von Modulen eine striktere Handhabung aktiviert werden.

```
            // Zugriff auf das statische Attribut 'instanceCounter'
            final Field staticfield = clazz.getField("instanceCounter");

            // Zugriff auf den Wert des statischen Attributs 'instanceCounter'
            final Object staticvalue = staticfield.get(null);
            System.out.println("instanceCounter = " + staticvalue);

            // Zugriff auf das private Attribut 'description'
            final Field field2 = clazz.getDeclaredField("description");

            // Zugriff ermöglichen
            field2.setAccessible(true);
            // Wertänderung des finalen Attributs 'description'
            field2.set(obj, "Changed FINAL attribute");
            final Object attributeValue2 = field2.get(obj);
            System.out.println("description = " + attributeValue2);
        }
        catch (final ReflectiveOperationException e)
        {
            // Behandlung sämtlicher durch Reflection möglicher Exceptions
            throw new IllegalStateException("can't access field!", e);
        }
        catch (final SecurityException e)
        {
            // Keine Erlaubnis, auf das Attribut zuzugreifen
            throw new IllegalStateException("insufficent security rights to " +
                                            "access field!", e);
        }
    }
}
```

Listing 12.5 *Ausführbar als* '**ATTRIBUTEACCESSEXAMPLE**'

Typsichere Konstruktion generischer Arrays

Wir wissen bereits, dass normalerweise `new` verwendet wird, um Objekte zu erzeugen. Soll dies dynamisch zur Laufzeit geschehen, kann man dazu Reflection und `newInstance()` nutzen. Nachfolgend wollen wir uns noch die Besonderheit der Erzeugung generischer Arrays anschauen. Für Arrays sind zwei überladene Varianten der Methode `newInstance()` folgendermaßen definiert:

```
public static Object newInstance(Class<?> componentType, int length)
throws NegativeArraySizeException

public static Object newInstance(Class<?> componentType, int... dimensions)
throws IllegalArgumentException, NegativeArraySizeException
```

Im ersten Fall wird ein eindimensionales und im zweiten ein n-dimensionales Array erzeugt, beispielsweise wie folgt:

```
String[] stringArray = (String[]) Array.newInstance(String.class, 3);
stringArray[0] = "First";
stringArray[1] = "Middle";
stringArray[2] = "Last";
System.out.println(Arrays.toString(stringArray));
```

```
String[][] twoDimStringArray = (String[][]) Array.newInstance(String.class,
                                                                3, 3);
twoDimStringArray[0][0] = "First";
twoDimStringArray[1][1] = "Middle";
twoDimStringArray[2][2] = "Last";
System.out.println(Arrays.deepToString(twoDimStringArray));
```

Diese Zeilen erzeugen folgende Ausgaben:

```
[First, Middle, Last]
[[First, null, null], [null, Middle, null], [null, null, Last]]
```

Hilfsmethode zum Erzeugen generischer Arrays Bereits in Abschnitt 3.7.2 hatten wir einführend betrachtet und auch diskutiert, warum man nicht einfach `new Object[]` nutzen und dieses passend casten kann. Als Abhilfe hatten wir dort folgende Hilfsmethode implementiert:

```
@SuppressWarnings("unchecked")
public static <T> T[] createTypedArray(final Class<T> clazz, final int size)
{
    return (T[]) Array.newInstance(clazz, size);
}
```

Den `componentType` ermitteln Manchmal möchte man dynamisch ein Array beliebigen generischen Typs erzeugen. Dazu benötigt man dann den Component Type.

Gehen wir noch mal einen Schritt zurück: In Java ist bekanntermaßen jedes Array vom Typ `Object`, unabhängig davon, ob die verwalteten Typen Objekte oder primitive Typen sind. Dadurch kann man für Arrays auch deren `Class<T>`-Objekt ermitteln. Darüber lässt sich mit `getComponentType()` der Typ der verwalteten Elemente bestimmen. Bei mehrdimensionalen Arrays kann man dies verschachtelt aufrufen – falls es sich nicht um ein Array handelt, erhält man `null` als Rückgabe:

```
String[] names = { "Tim", "Tom", "Mike" };
int[][] twoDimNumbers = { { 1, 2 }, { 3, 4 } };

componentTypeInfo(names.getClass());
componentTypeInfo(twoDimNumbers.getClass());
componentTypeInfo(twoDimNumbers[0].getClass());

protected static void componentTypeInfo(Class<?> clazz)
{
    Class<?> componentType = clazz.getComponentType();

    if (componentType != null)
    {
         System.out.println("ComponentType = " + componentType.getName());
    }
    else
    {
         System.out.println("no ComponentType given");
    }
}
```

Arrays generisch und typsicher erzeugen Kommen wir nun zum spannenden Thema: Wie erzeugen wir ein generisches Array typsicher und bringen die gesammelten Informationen zusammen?

Als Beispiel wollen wir eine generische Methode schreiben, die zwei Arrays zusammenfügt. Dabei soll der Typ egal sein, aber es soll typsicher sowohl für Referenz- als auch primitive Typen geschehen. Wir implementieren die Funktionalität wie folgt:

```
// T[] kann auch mehrdimensional sein
public static <T> T[] concat(final T[] first, final T[] second)
{
    final int len1 = first.length;
    final int len2 = second.length;

    final Class<?> componentType = first.getClass().getComponentType();

    @SuppressWarnings("unchecked")
    final T[] result = (T[]) createTypedArray(componentType, len1 + len2);
    System.arraycopy(first, 0, result, 0, len1);
    System.arraycopy(second, 0, result, len1, len2);

    return result;
}
```

Die nutzende Applikation definiert jeweils zwei Arrays, zum einen vom Typ `String[]` und zum anderen vom Typ `int[][]`:

```
public static void main(final String[] args)
{
   final String[] arr1 = { "Tim1", "Tom1", "Mike1" };
   final String[] arr2 = { "Tim2", "Tom2", "Mike2" };
   final int[][] values1 = { { 1, 2 }, { 3, 4 } };
   final int[][] values2 = { { 5, 6 }, { 7, 8 } };

   // type safe concatenate (=> using correct type, not Object[])
   final String[] result1 = concat(arr1, arr2);
   final int[][] result2 = concat(values1, values2);

   System.out.println(result1.getClass() + ": " + Arrays.toString(result1));
   System.out.println(result2.getClass() + ": " + Arrays.deepToString(result2));
}
```

Führen wir diese `main()`-Methode aus, so werden zwei passend typisierte Ergebnis-Arrays erzeugt und ausgegeben:

```
class [Ljava.lang.String;: [Tim1, Tom1, Mike1, Tim2, Tom2, Mike2]
class [[I: [[1, 2], [3, 4], [5, 6], [7, 8]]
```

12.1.4 Type Erasure und Typinformationen bei Generics

Im Internet und auch in Büchern liest man immer wieder mal Aussagen, dass durch die Type Erasure (vgl. Abschnitt 3.7.2) alle Typangaben zur Laufzeit verloren gehen und man darauf keinen Zugriff hat. Richtig ist, dass zur Laufzeit eine Vielzahl an Typangaben nicht mehr zugreifbar sind, beispielsweise sind tatsächlich keine Typangaben für Containerklassen vorhanden: Eine Liste »weiß« dann nicht mehr, auf welchen Typ sie noch beim Kompilieren eingeschränkt war. Allerdings ist es sowohl für Deklarationen von Variablen als auch für Parameter möglich, deren generische Typen zu ermitteln. Dazu dienen Methoden wie `getTypeParameters()` zur Bestimmung der generischen Parameter einer Klasse oder `getGenericType()` zur Abfrage des Typs eines Attributs sowie `getGenericReturnType()` und `getGenericParameterTypes()`, die Rückgabe- und generische Parametertypen liefern.

Wir werden uns nachfolgend eine Klasse `TypeErasureAndTypeInfoExample` ansehen, die zur Veranschaulichung zwei Attribute und zwei Methoden mit generischen Typenangaben definiert:

```
public class TypeErasureAndTypeInfoExample<A, B, C>
{
   private final String[] infoArray = {"Reflection", "and", "generic", "types"};
   protected List<String> info = Arrays.asList(infoArray);

   protected Set<SimplePerson> persons =
                               Collections.singleton(new SimplePerson("Tim"));

   // Generische Parameter nur zur Demonstration, hier funktional nutzlos
   public List<String> getInfo(final Map<Integer, SimplePerson> mapping)
   {
      return this.info;
   }

   // Generische Parameter nur zur Demonstration, hier funktional nutzlos
   public Set<SimplePerson> getPersons(final List<Long> values)
   {
      return this.persons;
   }

   public static void main(final String[] args)
   {
      final Class<?> clazz = TypeErasureAndTypeInfoExample.class;

      // Zugriff auf die Typparameter der Klasse
      System.out.println("getTypeParameters():  " +
                         Arrays.asList(clazz.getTypeParameters()));
      System.out.println();

      System.out.println("Fields:");
      final Field[] fields = clazz.getDeclaredFields();
      for (final Field field : fields)
      {
         System.out.println("getName():        " + field.getName());
         System.out.println("getType():        " + field.getType());
         // Zugriff auf generischen Typ des Attributs
         System.out.println("getGenericType(): " + field.getGenericType());
         System.out.println();
      }
```

```
      System.out.println("Methods:");
      final Method[] methods = clazz.getDeclaredMethods();
      for (final Method method : methods)
      {
         System.out.println("getName():                  " +
                            method.getName());
         System.out.println("getReturnType():            " +
                            method.getReturnType());
         // Zugriff auf generischen Rückgabetyp
         System.out.println("getGenericReturnType():     " +
                            method.getGenericReturnType());
         System.out.println("getParameterTypes():        " +
                            Arrays.asList(method.getParameterTypes()));
         // Zugriff auf generische Parametertypen
         System.out.println("getGenericParameterTypes(): " +
                            Arrays.asList(method.getGenericParameterTypes()));
         System.out.println();
      }
   }
}
```

Listing 12.6 *Ausführbar als* **'TYPEERASUREANDTYPEINFOEXAMPLE'**

Führt man das obige Programm TYPEERASUREANDTYPEINFOEXAMPLE aus, so kommt es zu folgenden Ausgaben, die zeigen, dass zur Laufzeit trotz Type Erasure sehr wohl Zugriff auf einige generische Typangaben besteht, und zwar für die Deklaration von Attributen und Parametern:

```
getTypeParameters():  [A, B, C]

Fields:
getName():        infoArray
getType():        class [Ljava.lang.String;
getGenericType(): class [Ljava.lang.String;

getName():        info
getType():        interface java.util.List
getGenericType(): java.util.List<java.lang.String>

getName():        persons
getType():        interface java.util.Set
getGenericType(): java.util.Set<ch12_advancedjava.reflection.SimplePerson>

Methods:
getName():                  main
// ... Infos zu main() ausgelassen ...

getName():                  getInfo
getReturnType():            interface java.util.List
getGenericReturnType():     java.util.List<java.lang.String>
getParameterTypes():        [interface java.util.Map]
getGenericParameterTypes(): [java.util.Map<java.lang.Integer,
                            ch12_advancedjava.reflection.SimplePerson>]

getName():                  getPersons
getReturnType():            interface java.util.Set
getGenericReturnType():     java.util.Set<ch12_advancedjava.
                                          reflection.SimplePerson>?
getParameterTypes():        [interface java.util.List]
getGenericParameterTypes(): [java.util.List<java.lang.Long>]
```

12.1.5 Fazit

Reflection bietet zusätzliche Möglichkeiten bei der Programmierung. Mit Bedacht eingesetzt können elegante Lösungen entstehen. ***Ist ein direkter Zugriff auf Klassen möglich, sollte dieser jedoch dem Einsatz von Reflection vorgezogen werden***. Dadurch erhöhen sich die Lesbarkeit und die Nachvollziehbarkeit.

Achtung: Typsicherheit und Refactorings

Durch Reflection verliert man Typsicherheit, da diverse Prüfungen nicht mehr zur Kompilierzeit, sondern erst zur Laufzeit erfolgen. Werden `Class<T>`-Referenzen über voll qualifizierte Namen mithilfe der Methode `Class.forName(String)` ermittelt, so kann dies zur Laufzeit Probleme auslösen, wenn Klassen umbenannt oder in andere Packages verschoben werden. Als Strings definierte Klassennamen werden von den Refactoring-Automatiken der IDEs in der Regel nicht erkannt und daher auch nicht angepasst.

Besonderheiten bei Reflection seit Java 9 und Modularisierung

Bis einschließlich Java 8 lässt sich mit Reflection so ziemlich jedes Sicherheitsmerkmal von Java umgehen. Beispielsweise konnte man sich mit `setAccessible(true)` selbst auf private Attribute und Methoden Zugriff verschaffen. Es war sogar möglich, wenn auch sicher nicht ratsam, als `final` definierte Attribute nachträglich noch mit einem anderen Wert zu versehen. Reflection war also ein mächtiges, aber auch gefährliches Hilfsmittel, mit dem sich die sonst üblichen Regeln außer Kraft setzen lassen.

Sofern man in Java 9 und später den Kompatibilitätsmodus nutzt und nicht mit eigenen Modulen arbeitet, wird man nicht allzu viele Unterschiede bemerken. Erst im Kontext der Modularisierung wird deutlich, dass Reflection nun auch die von Modulen vorgegebenen Sichtbarkeitsregeln beachtet und somit eingeschränkter verwendbar ist als früher. Details finden Sie in Abschnitt 15.3.2.

12.2 Annotations

Manchmal ist es wünschenswert, den Sourcecode mit Zusatzinformationen, sogenannten ***Metainformationen***, zu versehen. Dazu kann man etwa Kommentare im Sourcecode hinterlegen, um relevante Abläufe, Methoden, Parameter usw. zu beschreiben. Folgen diese Kommentare dem Javadoc-Stil, so kann man daraus mithilfe des Javadoc-Tools des JDKs eine Programmdokumentation erstellen. Dazu werden spezielle Teile und relevante Informationen aus den Kommentaren extrahiert.

Natürlich ist man bei der Kommentierung nicht auf die Beschreibung des Sourcecodes beschränkt, sondern man kann in Kommentaren auch Angaben etwa zu Konfigurationseinstellungen oder Datenbankverbindungsparametern hinterlegen. Genau dies wurde im Laufe der Zeit erkannt und vermehrt eingesetzt, nachdem neben dem Javadoc-

Tool weitere Tools entwickelt wurden, die die Auswertung von in Kommentaren angegebenen Informationen erlauben. Beispielsweise kann man spezielle Angaben in Kommentaren zur Sourcecode-Generierung nutzen. Darüber hinaus sind weitere Einsatzzwecke denkbar. Es lassen sich Parametrierungen oder Konfigurationseinstellungen aus den Angaben im Sourcecode auslesen. Diese Variante ist oftmals praktischer, als separate Metainformationsdateien zu verwenden (z. B. umfangreiche und mitunter unübersichtliche XML-Konfigurationsdateien).

Die ganze Sache hat aber einen Haken, da beim Einsatz von Metainformationen in Kommentaren folgendes Problem existiert: Die Angaben gehen beim Kompilieren verloren und stehen zur Laufzeit (in der `.class`-Datei) nicht mehr zur Verfügung. Insbesondere für Tools und Laufzeitumgebungen, wie z. B. Applikationsserver, ist es aber wichtig, auf diese Zusatzinformationen auch zur Laufzeit zugreifen zu können. Genau das wird durch Annotations möglich. Mit diesen lassen sich die Metainformationen direkt mit dem Sourcecode (und nicht separat) pflegen und somit häufig Konfigurationsaufwände reduzieren.

12.2.1 Einführung in Annotations

Seit JDK 5 existieren Annotations als Java-Sprachelemente. Eine Annotation beginnt immer mit einem '`@`'-Zeichen und wird vor demjenigen Programmelement (z. B. Klasse, Methode, Attribut) notiert, auf das sie sich bezieht. Im folgenden Beispiel wird die Annotation `@CreationInfo` an der Klasse `MyClass` sowie die Annotation `@Override` an der Methode `checkValues(String, int)` genutzt. Für die Annotation `@CreationInfo` sehen wir, dass Informationen als benannte Parameter (hier `author`, `description`) in Form einer kommaseparierten Angabe von Schlüssel-Wert-Paaren bereitgestellt werden können:

```
@CreationInfo(author = "Michael Inden",
              description = "This class is responsible for XYZ")
public class MyClass implements ValueChecker
{
    @Override
    public void checkValues(final String name, final int age)
    {
        ...
    }
}
```

Anhand der einleitenden Beschreibung und des obigen Beispiels kann man sich verschiedene Einsatzgebiete für Annotations vorstellen:

1. **Informationsbereitstellung für Compiler** – Im JDK sind diverse Annotations definiert. Einige davon erlauben es uns Entwicklern, dem Compiler Hinweise über das Programm selbst bereitzustellen. So kann z. B. mithilfe der Annotation `@Override` ausgedrückt werden, dass eine Methode einer Basisklasse überschrieben werden soll. Der Compiler kann diesen Hinweis nutzen, um mögliche Verstöße beim Überschreiben, etwa durch einen Tippfehler im Methodennamen, erkennen zu können.

2. **Informationsbereitstellung für Tools** – Die zuvor im Listing genutzte Annotation `@CreationInfo` kann beispielsweise von einem speziellen Tool, einem sogenannten ***Annotation Processor***, ausgewertet werden, um die in der Annotation hinterlegten Angaben zum Autor (`author`) und zur Beschreibung (`description`) zu ermitteln und zu einer Implementierungsdokumentation zusammenzutragen. Sinnvollerweise würde man dies über Javadoc regeln, hier soll das Beispiel lediglich als Idee einen möglichen Anwendungsfall verdeutlichen. Normalerweise dienen Annotations als Eingabe für Tools, die daraus beispielsweise Konfigurationsdateien, Datenbankabfragen oder sogar Sourcecode generieren können.
3. **Informationsbereitstellung zur Laufzeit** – Annotations und deren Parametrierung kann man zur Laufzeit über Reflection (vgl. Abschnitt 12.1) auslesen. Damit lassen sich z. B. Cross Cutting Concerns, wie Logging, umsetzen. Annotations können zudem als Hilfsmittel dienen, um Abhängigkeiten aufzulösen und Referenzen an annotierte Attribute zuzuweisen. Für beide Anwendungsfälle müssen Annotations von der jeweiligen Laufzeitumgebung (Webcontainer, Applikationsserver etc.) ausgelesen und verarbeitet werden: Beispielsweise erfolgt die Dependency Injection des Spring-Frameworks auf Basis von Annotations.

Nicht alle Annotations sind per se für jeden der oben genannten Anwendungsfälle nutzbar, da Annotations verschiedene Lebensdauern haben. Einige sind nur während des Kompilierens verfügbar, andere sogar noch zur Laufzeit des Programms: Abhängig von ihrer Definition verwirft oder überträgt der Java-Compiler die Annotations bei der Übersetzung der Source-Datei in die `.class`-Datei. Analog gilt dies beim Laden der Klassen in die JVM: Annotierte Informationen werden nur dann übertragen, wenn die Lebensdauer in der Definition entsprechend festgelegt wurde. Darauf gehe ich später bei der Beschreibung der Definition der selbst definierten Annotation `@CreationInfo` ein.

12.2.2 Standard-Annotations des JDKs

Wie eingangs erwähnt, ist ein Anwendungsfall von Annotations die Bereitstellung von Informationen für den Compiler. Die Annotations `@Deprecated`, `@Override` und `@SuppressWarnings` sind im Package `java.lang` definiert und werden während der Kompilierung ausgewertet. Diese Annotations haben wir bereits verwendet, ohne uns dabei viele Gedanken zu machen. Rekapitulieren wir deren Einsatzzweck.

Die Annotation `@Deprecated`

Diese Annotation zeigt an, dass ein markiertes Element (Klasse, Methode, ...) veraltet ist und nicht mehr benutzt werden sollte. Wird eine derart markierte Methode eingesetzt, so kommt es beim Kompilieren zu Warnungen. Zusätzlich zu dieser Annotation sollte der Hinweis `@deprecated` im Javadoc-Kommentar genutzt werden. Ergänzend kann dort auf mögliche alternativ zu nutzende Klassen oder Methoden eingegangen werden:

```
/**
 * @deprecated Diese Methode führt zu Problemen. Bitte stattdessen
 *             {@link #newMethod()} nutzen.
 */
@Deprecated
public void oldMethod(int someValue)
```

Die Annotation `@Override`

Diese Annotation drückt aus, dass eine damit annotierte Methode eine gleichnamige Methode einer Basisklasse überschreibt bzw. eines Interface implementiert. Obwohl sich dieser Einsatz womöglich nicht sinnvoll anhören mag, gibt es doch gute Gründe dafür. Durch Angabe dieser Annotation gleicht der Compiler die Signatur einer Methode mit derjenigen einer Basisklasse ab und warnt, wenn kein Überschreiben der gekennzeichneten Methode vorliegt. Auf diese Weise kann man Fehler finden, wenn man sich beim Methodennamen vertippt. Dadurch würde man nämlich, statt – wie gewollt – die Methode der Basisklasse zu überschreiben, eine neue Methode definieren. Dieser Fehler ist ohne den Einsatz der Annotation `@Override` schwieriger zu entdecken und könnte möglicherweise zu ungewünschtem Programmverhalten führen.

Der Einsatz der Annotation `@Override` ist beispielsweise bei der Definition einer `equals()`-Methode in eigenen Klassen nützlich, um zu verhindern, dass deren Parameter fälschlicherweise nicht vom Typ `Object` ist und die Methode dann überladen anstatt überschrieben wird:

```
@Override  // => Compile-Error
public void equals(final Person person)
{
    ...
}
```

Die Annotation `@SuppressWarnings`

Diese Annotation erlaubt es, Compiler-Warnungen zu unterdrücken. Der zu unterdrückende Typ von Warnung wird als Parameter übergeben, etwa `@SuppressWarnings("unchecked")`. Um Fehler zu entdecken und nicht zu verstecken, sollte man nur ausnahmsweise und im kleinstmöglichen Scope Gebrauch von `@SuppressWarnings` machen. Wichtige vordefinierte Parameterwerte sind:

- `boxing` – Es werden keine Warnungen für Typumwandlungen mit Auto-Boxing und Auto-Unboxing erzeugt.
- `deprecation` – Beim Einsatz veralteter Methoden oder Klassen kommt es nicht zu Warnungen.
- `unused` – Unbenutzte Variablen oder Methoden führen nicht zu Warnungen. Das kann beim Einsatz externer Bibliotheken hilfreich sein, um deren Warnungen auszublenden.

- `unchecked` – Wenn man Generics und untypisierte Klassen kombiniert verwendet, so kann durch den Compiler keine Typsicherheit mehr garantiert werden. Somit wird beim Einsatz dieser Annotation für möglicherweise problematische Zugriffe keine Warnung erzeugt. Das sollte man aber mit großer Vorsicht und nur nach genauer Prüfung der Zugriffe verwenden.

Das folgende (Negativ-)Beispiel zeigt einige Annotations für eine Methode `getPersons()`, die noch keine Generics nutzt, deren Rückgabe jedoch für eine typsichere Iteration in der `main()`-Methode eingesetzt werden soll. Durch die Verwendung der zuvor genannten Annotations kompiliert folgender Sourcecode ohne Warnungen:

```
@SuppressWarnings("unchecked")
public static void main(final String[] args)
{
    // Achtung: nur zur Demonstration der Möglichkeiten von Annotations
    @SuppressWarnings(value={"unchecked", "deprecation"})
    final List<Person> persons = getPersons();
    for (final Person currentPerson : persons)
    {
        doSomethingWithPerson(currentPerson);
    }
}

@Deprecated
public static List getPersons()
{
    return new ArrayList();
}
```

Beachten Sie bitte, dass es in diesem Beispiel wirklich nur um die Darstellung der Nutzung von Annotations geht. In der Praxis sollte man besser die Methode `getPersons()` überarbeiten, statt die Fehler mithilfe von Annotations auszublenden.

In diesem Listing lernen wir ein weiteres Feature kennen: Es kann nicht nur ein zu unterdrückender Warnungstyp, sondern auch eine Aufzählung zu unterdrückender Warnungstypen angegeben werden. Dazu besitzt die Annotation einen impliziten logischen Parameter namens `value`. Diesem lassen sich – eine entsprechende Definition der Annotation vorausgesetzt – auch eine kommaseparierte Folge von Strings in der Notation `value={"Wert1", ..., "Wert n"}` übergeben. Heißt das Attribut `value` und ist es das einzige, braucht man es nicht anzugeben. Man notiert kürzer Folgendes: `@SuppressWarnings("unchecked", "deprecation")`.

12.2.3 Definition eigener Annotations

Im Programmiereralltag setzt man die zuvor beschriebenen vordefinierten Annotations ein, ohne sich (viele) Gedanken darüber zu machen, wo diese Annotations definiert sind und vor allem, wie sie ausgewertet werden. Tatsächlich ist diese Spezialkenntnis auch eher selten wichtig.

Zum besseren Verständnis der Arbeitsweise von Annotations lohnt sich allerdings ein Blick hinter die Kulissen. Annotations werden in eigenen Dateien analog zu normalen Klassen und Interfaces definiert. Statt des Schlüsselworts `class` bzw. `interface`

wird hier `@interface` verwendet. Das sorgt automatisch dafür, dass die Annotation den Basistyp `Annotation` aus dem Package `java.lang.annotation` besitzt.

Betrachten wir die Realisierung unserer im einführenden Beispiel kennengelernten Annotation `@CreationInfo`. Dieser wurden verschiedene Informationen in Form von Parametern übergeben. In der Annotation-Definition ist für jeden Parameter eine parameterlose Methode gleichen Namens zu erstellen:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

// Meta-Annotations
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)

// Annotation-Definition
public @interface CreationInfo
{
    // Methoden zur Übergabe von Informationen durch Parameter
    String author() default "Michael Inden";
    String description();
    Class<?>  baseclass() default java.lang.Object.class;
    Class<?>[] interfaces() default {};
    String[] tags() default {};
}
```

Das ist tatsächlich alles an Sourcecode zur Definition dieser eigenen Annotation! Dabei sehen wir hier einige Schreibweisen und Elemente, die uns in dieser Form oder aus der bisher vorgestellten Java-Syntax (noch) nicht geläufig sind. Das sind vor allem das Schlüsselwort `default`[3] hinter dem Methodennamen sowie die beiden Annotations `@Retention` und `@Target`: Mit `@Retention(RetentionPolicy.RUNTIME)` wird für Annotations definiert, dass sie später zur Laufzeit verfügbar sein sollen. Über die Angabe von `@Target(ElementType.TYPE)` wird festgelegt, dass beliebige Typen, d. h. Klassen, Interfaces, Annotations und Enums, markiert werden können.

Konzentrieren wir uns hier zunächst auf die Methoden, die in der Annotation-Definition aufgeführt sind. Durch diese lassen sich die Parameter einer Annotation beschreiben, die bei Verwendung der Annotation im Sourcecode angegeben werden müssen bzw. können (im Falle von Defaultparametern mit dem Schlüsselwort `default`), um Informationen zu übermitteln. Der Datentyp der Parameter wird durch deren Rückgabetyp bestimmt. Dabei dürfen als Einschränkung nur primitive Typen, Strings, Klassentypen, Enums und Annotation-Typen bzw. eindimensionale Arrays der zuvor genannten Typen als Rückgabewert genutzt werden. Achten Sie zudem darauf, dass die Methoden immer parameterlos anzugeben sind.

Zur einfacheren Handhabung beim Einsatz dieser Annotation wurden hier durch Angabe des Schlüsselworts `default` und eine korrespondierende Wertangabe einige Defaultparameter definiert.

[3] Mit Java 8 wurden in Interfaces als Syntaxerweiterung die Defaultmethoden zur Vorgabe von Standardverhalten eingeführt (vgl. Abschnitt 4.2.1).

Bei der Verwendung der Annotation können derartige Parameter einen Übergabewert erhalten, müssen es aber nicht. Fehlt bei der Angabe im Sourcecode der Wert eines solchen Parameters, so wird der in der Definition der Annotation hinterlegte Defaultwert genutzt. Alle anderen Parameter sind erforderlich und müssen angegeben werden. Das bedeutet insbesondere, dass eine fehlende Angabe eines Parameters ohne Defaultwert im Sourcecode zu Kompilierfehlern führt.

Um Fehler zu vermeiden und vollständige Angaben zu den Parametern einer Annotation beim Editieren des Sourcecodes machen zu können, ist die Auto-Complete-Funktionalität in Eclipse sehr hilfreich. Diese gibt kontextbezogene Hinweise. Das ist in Abbildung 12-1 für unsere selbst definierte Annotation `@CreationInfo` dargestellt.

Abbildung 12-1 *Auto-Complete-Funktionalität für Annotations in Eclipse*

Meta-Annotations (`@Retention` und `@Target`)

Bei der obigen Definition der Annotation `@CreationInfo` haben wir weitere Elemente kennengelernt: Meta-Annotations, die vor der eigentlichen Definition der Annotation `@CreationInfo` notiert sind. Über die Meta-Annotations `@Retention` und `@Target` werden Informationen bzw. Eigenschaften der hier definierten Annotation `@CreationInfo` festgelegt. Dabei kann man folgende Angaben machen:

- **`@Retention`** – Beschreibt die Lebensdauer einer Annotation. Diese wird als Parameter durch eine sogenannte `RetentionPolicy` spezifiziert:
  - `Source`: Die Annotation ist *nur* im Sourcecode vorhanden und wird *nicht* in den Bytecode, die `.class`-Datei, übernommen. Nach der Kompilierung stehen die Annotation-Informationen demnach nicht mehr zur Verfügung.

- `Class`: Die Annotation wird in die generierte `.class`-Datei übernommen. Zur Laufzeit stehen die Informationen aber nicht mehr zur Verfügung. ***Das ist die Defaulteinstellung***.
- `Runtime`: Die Annotation ist während der Laufzeit verfügbar. Auf sie kann dann per Reflection zugegriffen werden. Der folgende Abschnitt 12.2.4 geht darauf genauer ein.

- **`@Target`** – Über diese Meta-Annotation wird geregelt, welche Elemente mit der Annotation markiert werden dürfen. Das wird über `ElementType` genauer spezifiziert. Folgende selbsterklärende Aufzählungswerte sind erlaubt: `ANNOTATION_TYPE`, `TYPE`, `CONSTRUCTOR`, `METHOD`, `FIELD`, `PARAMETER` und `PACKAGE`. Für `TYPE` sei angemerkt, dass dieser für beliebige Typen, also für Klassen, Interfaces, Annotations sowie Enums, steht. Mit `ANNOTATION_TYPE` beschreibt man, dass die Annotation nur für Annotations genutzt werden darf.

12.2.4 Annotations zur Laufzeit auslesen

Die Definition der Annotation `@CreationInfo` ist nun bekannt. Jetzt wollen wir die Annotation im Einsatz erleben. Als Beispiel dient eine auf das Wesentliche gekürzt dargestellte Klasse `AnnotationUsage`, die die Interfaces `Runnable` und `Serializable` implementiert und eine direkte Subklasse von `Object` ist. Diese Informationen wollen wir mit unserer Annotation `@CreationInfo` beschreiben und kommen zu folgender Realisierung:

```
@CreationInfo(author="Mike",
              description="Demonstration einer eigenen Annotation",
              baseclass=java.lang.Object.class,
              interfaces={java.lang.Runnable.class,
                          java.io.Serializable.class},
              tags={"Annotation", "Definition", "Advanced Java"})
public class AnnotationUsage implements Runnable, Serializable
{
    @Override
    public void run()
    {
        // ...
    }
}
```

Interessant ist die Syntax zur Angabe der Array-Parameter `interfaces` und `tags`. Diese erfolgt in geschweiften Klammern als kommaseparierte Liste, die Objekte vom Typ `Class<?>` bzw. im zweiten Fall Objekte vom Typ `String` enthält.

Die Annotation `@CreationInfo` haben wir zur Beschreibung von Metainformationen über die Klasse `AnnotationUsage` eingesetzt. Nun wollen wir die dort hinterlegten Informationen mithilfe eines Java-Programms auslesen. Dazu nutzen wir hier Reflection – im Speziellen die Methode `getAnnotation(Class<A extends Annotation>)` –, um eine Instanz unserer Annotation `@CreationInfo` zu erhalten und deren Methoden aufzurufen. Wie im vorherigen Abschnitt über Reflection bereits

beschrieben, könnte man viele der Informationen auch ohne die Annotation auslesen, z. B. diejenigen über implementierte Interfaces – zudem wären die Angaben verlässlicher, falls mal eine Änderung nicht in der Annotation nachgezogen wird. Im folgenden Listing betrachten wir zu Demonstrationszwecken trotzdem eine programmatische Umsetzung mithilfe von Reflection:

```
public static void main(final String[] args) throws Exception
{
   // Auslesen des selbst definierten Annontationstyps CreationInfo
   final CreationInfo creationInfo = AnnotationUsage.class.
                                        getAnnotation(CreationInfo.class);

   if (creationInfo != null)
   {
      printCreationInfo(creationInfo);
   }
   else
   {
      System.out.println("No " + CreationInfo.class.getSimpleName() +
                         " annotation present!");
   }
}

private static void printCreationInfo(final CreationInfo creationInfo)
{
   System.out.println("author():      " + creationInfo.author());
   System.out.println("description(): " + creationInfo.description());
   System.out.println("baseclass():   " + creationInfo.baseclass());
   System.out.println("interfaces():  " + Arrays.toString(creationInfo.
         interfaces()));
   System.out.println("tags():        " + Arrays.toString(creationInfo.tags()));
}
```

Listing 12.7 *Ausführbar als* '**ANNOTATIONREADEREXAMPLE**'

Führen Sie das Programm ANNOTATIONREADEREXAMPLE aus, so erscheint in etwa folgende Ausgabe auf der Konsole:

```
author():      Mike
description(): Demonstration einer eigenen Annotation
baseclass():   class java.lang.Object
interfaces():  [interface java.lang.Runnable, interface java.io.Serializable]
tags():        [Annotation, Definition, Advanced Java]
```

Fazit

Dieser Abschnitt hat eine kurze Einführung in die Definition und Verarbeitung von Annotations gegeben. Das erlangte Wissen ist für den normalen Programmiereralltag oft ausreichend. Der Gebrauch eigener Annotations ist eher unüblich – für Toolhersteller oder fortgeschrittene Entwickler kann diese Thematik jedoch von größerem Interesse sein. Am Kapitelende finden Sie Hinweise auf weiterführende Literatur.

12.3 Serialisierung

Java bietet einen Automatismus, der Objekte in eine Folge von Bytes umwandeln bzw. Objekte aus einer solchen erzeugen kann. Diese Vorgänge nennt man ***Serialisierung*** bzw. ***Deserialisierung***. Vereinfachend wird im Folgenden für beides der Begriff Serialisierung verwendet. Über das Marker-Interface `java.io.Serializable` werden Klassen als serialisierbar markiert. Damit wird es der JVM möglich, Objekte in einem beliebigen `OutputStream` zu speichern und aus einem `InputStream` später wieder einzulesen. Die Serialisierung ist vom Betriebssystem unabhängig und ermöglicht dadurch den Austausch von Daten bzw. Objekten zwischen JVMs auf verschiedenen Rechnern.

Aufgrund der genannten Eigenschaften lässt sich Serialisierung dazu nutzen, Objekte über ein Netzwerk zu transportieren. Die Technik eignet sich auch dazu, den momentanen Objektzustand in einer Datei persistent zu speichern. Dieser kann dann später wieder eingelesen werden und kann so z. B. Undo-Operationen erleichtern.

12.3.1 Grundlagen der Serialisierung

Betrachten wir als einführendes Beispiel das Speichern und Laden von `Person`-Objekten mithilfe von Serialisierung. Dazu verwenden wir die Klassen `ObjectInputStream` und `ObjectOutputStream`. Diese bieten folgende Methoden:

- `writeObject(Object)` – Schreibt ein Objekt in einen `ObjectOutputStream`.
- `Object readObject()` – Liest ein Objekt aus einem `ObjectInputStream`.

Damit diese Methoden korrekt arbeiten können, müssen die damit zu verarbeitenden Klassen das Marker-Interface `Serializable` implementieren.

Klassen als serialisierbar markieren

Gemäß der Einleitung muss die bereits bekannte Klasse `Person` nur derart erweitert werden, dass sie das Interface `Serializable` implementiert:

```
public final class Person implements Serializable
{
   private final String name;
   private final String city;
   private final LocalDate birthday;

   public Person(final String name, final String city, final LocalDate birthday)
   {
      this.name = Objects.requireNonNull(name, "name must not be null");
      this.city = Objects.requireNonNull(city, "city must not be null");
      this.birthday = Objects.requireNonNull(birthday, "birthday must " +
                                                       "not be null");
   }

   // ...
```

Die Klassen `ObjectInputStream` und `ObjectOutputStream`

Nehmen wir an, wir wollten ein `Person`-Objekt in einer Datei `Test.ser` speichern. Die Ausgabe in die Datei nutzt einen `FileOutputStream`. Die Konvertierung in eine Folge von Bytes und das Speichern des Objekts in diesen Stream wird durch die Klasse `ObjectOutputStream` und dessen Methode `writeObject(Object)` durchgeführt:

```
public static void main(final String[] args) throws IOException
{
   // Nutze ARM, um lesbare Ressourcenzugriffe zu schreiben
   try (final ObjectOutputStream objectOutStream = new ObjectOutputStream(
            new BufferedOutputStream(new FileOutputStream("Test.ser"))))
   {
      // Schreibe Objekt in die Datei
      final Person person = new Person("Test", "TestCity", LocalDate.now());
      objectOutStream.writeObject(person);
      System.out.println("Wrote to stream: " + person);
   }
}
```

Listing 12.8 *Ausführbar als* '**SERIALIZATIONEXAMPLE**'

An diesem Beispiel erkennt man gut, wie wenig Aufwand die Speicherung aufseiten der Applikation erfordert – dabei hilft auch das Sprachfeature ARM zum automatischen Schließen von Ressourcen (vgl. Abschnitt 5.6.4). Das Speichern des Objekts mithilfe der Methode `writeObject(person)` stellt gegenüber dem Einsatz von Methoden der Klasse `DataOutputStream` einen enormen Fortschritt dar. Jene Art der Verwendung haben wir bereits in Abschnitt 5.5.2 besprochen und dort erfahren, dass für jedes Attribut eine typabhängige `write()`-Methode, etwa `writeLong(long)`, aufgerufen werden muss, um die Daten in den Stream zu schreiben. Allerdings stehen nicht für jeden Datentyp passende `write()`-Methoden bereit. Dies erfordert dann eine vom Entwickler implementierte Umwandlung der Attribute in eine Folge von Bytes.

Als Nächstes betrachten wir das Deserialisieren zum Einlesen der gespeicherten Daten aus der gerade geschriebenen Datei `Test.ser`. Wir nutzen einen `FileInputStream` zum Dateizugriff und die Methode `readObject()` der Klasse `ObjectInputStream` zur Rekonstruktion eines Objekts. Diese Methode liest alle benötigten Informationen aus einem beliebigen Stream ein. Da die Methode nur eine Referenz vom Typ `Object` zurückliefert, ist für ein sinnvolles Weiterarbeiten mit dem Objekt in der Regel ein anschließender Cast auf den erwarteten Typ notwendig:

```
public static void main(final String[] args) throws Exception
{
   try (final ObjectInputStream objectInStream = new ObjectInputStream(
            new BufferedInputStream(new FileInputStream("Test.ser"))))
   {
      // Rücklesen des Objekts, ohne Konstruktoraufruf
      final Person personFromStream = (Person) objectInStream.readObject();
      System.out.println("Back from stream: " + personFromStream);
   }
}
```

Listing 12.9 *Ausführbar als* '**DESERIALIZATIONEXAMPLE**'

So einfach die Serialisierung in diesem Beispiel scheint, so komplex sind teilweise die Zusammenhänge und Abläufe im Hintergrund. Im Datenformat der Serialisierung müssen alle diejenigen Informationen zu Typen und Werten enthalten sein, die ein späteres Wiederherstellen eines Objekts ermöglichen. Beim Einlesen muss ein Objekt mit seinen Attributen aus den gespeicherten Informationen rekonstruiert werden – mit gleichen Inhalten, allerdings anderen Referenzen. Wenn man die beiden Programme SERIALIZATIONEXAMPLE und DESERIALIZATIONEXAMPLE nacheinander ausführt, erkennt man dies anhand der unterschiedlichen Referenzen, die für die beiden `Person`-Objekte auf der Konsole beim Speichern und Einlesen ausgegeben werden:

```
Person: [name=Test, ...] / ch12_advancedjava.serializable.Person@8b2fd8f
Person: [name=Test, ...] / ch12_advancedjava.serializable.Person@2626d4f1
```

Den Serialisierungsvorgang, d. h. die Konvertierung in eine und aus einer Folge von Bytes sowie den Transfer in und aus Streams, selbst zu programmieren wäre fehleranfällig und aufwendig, wie dies schon in Abschnitt 5.5.4 deutlich wurde. Generell erleichtert die Serialisierungsautomatik die Arbeit. Wie bereits bekannt, muss man im einfachsten Fall lediglich das Interface `Serializable` implementieren. Diese Vereinfachung hat allerdings ihren Preis – das verwendete Datenformat ist recht »gesprächig«. Prüfen Sie es durch Ausführen des Programms SERIALIZATIONEXAMPLE. Die so entstehende Datei `Test.ser` wird 191 Bytes groß, weil in den Datenstrom die Klassennamen und die Typen in Form voll qualifizierter Angaben gespeichert werden:

```
<AC><ED>^@^Esr^@%ch12_advancedjava.serializable.PersonA<8C><CC>É<94>~<88>^B^@^CL
      ^@^Hbirthdayt^@^ULjava/time/LocalDate;L^@^Dcityt^@^RLjava/lang/String;L^@^
      Dnameq^@~^@^Bxpsr^@^Mjava.time.Ser<95>]<84><BA>ESC"H<B2>^L^@^@xpw^G^C^@^@^G
      <E3>^C^Xxt^@^HTestCityt^@^DTest
```

Das Ganze erscheint doch relativ viel, wenn man bedenkt, dass lediglich zwei kurze Texte und ein Datumswert gespeichert werden. Der Overhead nimmt bei komplexeren Objekten noch zu und kann sich negativ auf die Performance auswirken. Bevor wir einige Optimierungsmöglichkeiten bei der Speicherung kennenlernen, betrachten wir zunächst weitere Grundlagen zur Serialisierung.

Der Serialisierungsvorgang im Detail

Bei der Serialisierung eines Objekts werden alle nicht statischen[4] Attribute der Klasse (*einschließlich der privaten*) verarbeitet, sofern diese nicht explizit mit dem Schlüsselwort `transient` gekennzeichnet und damit von der Verarbeitung ausgeschlossen werden. Der Automatismus der Serialisierung kann »von Hause aus« alle primitiven Datentypen in einen Stream schreiben. Damit Objektreferenzen verarbeitet werden können, müssen deren Typen ebenfalls das Interface `Serializable` implementieren. Dies ist für viele Klassen des JDKs der Fall. Für Arrays gilt: Wenn ein Array serialisierbar sein soll, müssen auch die gespeicherten Elemente serialisierbar sein.

[4] Statische Attribute werden nicht serialisiert, da sie keiner Objektinstanz zugeordnet sind.

Ansonsten kommt es während des Serialisierungsvorgangs zu einer `java.io.NotSerializableException`. Im einführenden Beispiel haben wir diese Eigenschaft der Klassen `String` und `LocalDate` genutzt, ohne diese Details zu kennen.

Nicht serialisierbare Attribute und `transient` Eben erwähnte ich kurz, dass man den expliziten Ausschluss eines Attributs von der automatischen Verarbeitung durch die Angabe des Schlüsselworts `transient` erreicht. Jedes derart gekennzeichnete Attribut wird nicht in den Datenstrom geschrieben und auch bei einem späteren Einlesen nicht berücksichtigt. Um solche Attribute korrekt zu initialisieren, ist daher beim Einlesen ein spezielles Vorgehen zur Initialisierung erforderlich. Vergisst man dies, so bleiben die Attribute mit ihrem Defaultwert belegt, d. h. `false` für boolesche Variablen, 0 für Zahlen und `null` für Referenzen. Auf Abhilfen gehe ich gleich genauer in Abschnitt 12.3.2 ein.

Serialisierung der Klassenhierarchie Der Vorgang der Serialisierung wird für alle Bestandteile der Klassenhierarchie durchgeführt. Wie bereits beschrieben, werden referenzierte Objekte wiederum komplett serialisiert. Es wird also der gesamte Objektgraph in den Stream geschrieben. Die Serialisierungsautomatik sorgt allerdings dafür, dass mehrfach referenzierte Objekte nur einmal serialisiert werden. Das Einlesen geschieht ähnlich: Solange alle Attribute und Bestandteile der Klassenhierarchie ***serialisierbar*** sind, erledigt der Automatismus sämtliche Aufgaben. Es erfolgt insbesondere ***kein*** Konstruktoraufruf. Ein Spezialfall ist jedoch zu beachten: Ist eine Basisklasse nicht serialisierbar, so kommt es während der Rekonstruktion unter Umständen zu Exceptions. Betrachten wir dies genauer.

Spezialfall: Nicht serialisierbare Basisklassenbestandteile Es ist möglich, dass eine Klasse das Interface `Serializable` implementiert, eine Basisklasse dies jedoch nicht tut. In Abbildung 12-2 ist eine solche Situation für die Klasse `SerializableClass` gezeigt.

Abbildung 12-2 *Spezialfall: Nicht serialisierbare Basisklassenbestandteile*

Zur Rekonstruktion eines nicht serialisierbaren Basisklassenbestandteils erfolgt beim Deserialisierungsvorgang – im Gegensatz zum Standardfall – ein Konstruktoraufruf: Der Defaultkonstruktor wird dazu genutzt, um den Initialisierungsschritt für diesen

Klassenbaustein vorzunehmen. Gibt es keinen Defaultkonstruktor, so kommt es beim Deserialisieren zu einer `java.io.InvalidClassException` mit dem Hinweis »`no valid constructor`«. Man fragt sich nun, wieso sich ein derartiges Objekt überhaupt serialisieren lässt und erst beim Einlesen Probleme auftreten. Die Antwort ist einfach: Der Automatismus kann über Reflection zwar alle zur Speicherung notwendigen Bestandteile ermitteln, der Rekonstruktionsprozess benötigt dagegen weitere Informationen, nämlich die Anzahl und Typen der Attribute sowie deren Speicherbelegung. Für serialisierbare Klassen kümmert sich die Automatik um diese Details. Für nicht serialisierbare Basisklassen macht dies der Defaultkonstruktor.

Es ist eher unwahrscheinlich, dass die Attribute durch einen Aufruf des Defaultkonstruktors in den gewünschten Zustand versetzt werden. Um einen korrekten Objektzustand herzustellen, müssen dann die Daten der Basisklasse vom Entwickler selbst gespeichert und wieder eingelesen werden. Dazu benötigt man eine Möglichkeit, sich in den Ablauf des Automatismus einzuklinken, um erforderliche Daten geeignet zu verarbeiten. Der folgende Abschnitt geht darauf genauer ein.

Tipp: Serialisierung vs. Kapselung und Versionierung

Das Format der Serialisierung nutzt zur Identifizierung von Typen neben der konkreten Typangabe als voll qualifizierter Name zusätzlich eine spezielle Versionsnummer. Diese kann explizit als Variable `serialVersionUID` im Sourcecode angegeben werden. Geschieht dies nicht, so wird beim Kompilieren automatisch eine solche Versionsnummer berechnet und beim Serialisieren verwendet.

In die Berechnung gehen unter anderem die Methoden einer Klasse und der Zeitpunkt der Kompilierung ein. Jede neu eingeführte Methode verändert damit diese Versionsnummer und führt zu Inkompatibilitäten zu älteren Versionen des Objekts. Die Versionsnummer ändert sich demnach häufiger, als es tatsächlich zu Änderungen an der Schnittstelle oder in der Semantik kommt.

Mit Serialisierung zerstört man zudem Kapselung: Jede Änderung an der Struktur einer Klasse führt zu Inkompatibilitäten mit früher serialisierten Versionen der Objekte. Der Grund ist folgender: Alle nicht explizit von der Serialisierung ausgeschlossenen Attribute einer Klasse werden verarbeitet und in die Ausgabe geschrieben. Das gilt auch für private Attribute, die gegebenenfalls Implementierungsdetails darstellen. Bei einem späteren Einlesen werden diese Informationen exakt erwartet.

Finden strukturelle Änderungen statt, so kann ein zuvor serialisiertes Objekt als Folge nicht mehr verarbeitet werden: Diesem fehlen die durch die Änderungen hinzugefügten Informationen. Die Argumentation gilt gleichermaßen für das Entfernen und Ändern von Attributen und Methoden.

Zur Lösung der genannten Probleme ist die Vergabe einer speziellen vom Entwickler definierten `serialVersionUID` möglich. Nur bei Änderungen, die wirklich zu einer Inkompatibilität zwischen verschiedenen Versionen eines Objekts führen, wird die Versionsnummer vom Entwickler angepasst. Sollen verschiedene Versionen verarbeitet werden können, so muss der Serialisierungsvorgang für jede Version einzeln selbst programmiert werden. Abschnitt 12.3.3 beschreibt dies detaillierter.

12.3.2 Die Serialisierung anpassen

Für viele Anwendungsfälle reicht die von Java bereitgestellte Serialisierung bereits aus. Allerdings gibt es einige Situationen, in denen man den Serialisierungsprozess anpassen möchte oder muss. In der Regel ist dies der Fall, wenn eine Klasse eine andere Klasse referenziert, die nicht serialisierbar ist. Weiterhin gibt es Attribute, die aus anderen Gründen nicht sinnvoll persistiert werden können. Dazu gehören z. B. Attribute, die eine Ressource repräsentieren, etwa eine Verbindung zu einer Datenbank. Solche Attribute kann man explizit von der Verarbeitung durch die Serialisierungsautomatik ausschließen. Weitere Anwendungsfälle bestehen darin, eine platzsparendere Speicherung zu erzielen oder verschiedene Versionen einer Klassendefinition zu unterstützen (vgl. Abschnitt 12.3.3).

Problemkontext

Nehmen wir an, die Klasse `NonSerializableClass` modelliert eine Verbindung zu einer Datenbank und erhält zur Konstruktion verschiedene Informationen zum Aufbau einer Verbindung zur Datenbank als String. Instanzen dieser Klasse können nicht serialisiert werden, da verschiedene Bestandteile nicht serialisierbar sind. Die folgende Klassendefinition implementiert daher das Interface `Serializable` nicht:

```
public class NonSerializableClass
{
    private final String databaseConnectionParams;

    public NonSerializableClass(final String databaseConnectionParams)
    {
        this.databaseConnectionParams = databaseConnectionParams;
    }

    public String getDatabaseConnectionParams()
    {
        return databaseConnectionParams;
    }

    // ...
```

Nehmen wir lediglich zu Demonstrationszwecken an – aber keinem schönen OO-Design folgend –, dass die Klasse `Person` um ein nicht serialisierbares Attribut `nonSerializable` vom Typ `NonSerializableClass` folgendermaßen erweitert worden ist:

```
public final class Person implements Serializable
{
    private String name;
    private String city;
    private LocalDate birthday;

    // Ausschluss von der Serialisierung
    private transient NonSerializableClass nonSerializable;

    // ...
```

Modifikation des Serialisierungsvorgangs

Zur korrekten Verarbeitung `transient` definierter Attribute muss der Serialisierungsvorgang angepasst werden. Durch ***Implementierung der folgenden zwei privaten Methoden in der zu serialisierenden Klasse*** lässt sich dies erreichen:[5]

```
private void writeObject(ObjectOutputStream outStream) throws IOException
private void readObject(ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
```

Die Methode `writeObject(ObjectOutputStream)` realisiert das Speichern von Objektinformationen in einen Stream, die Methode `readObject(ObjectInputStream)` das Einlesen. Auf den ersten flüchtigen Blick ist kein Unterschied zu den zuvor genutzten Methoden `writeObject(Object)` und `Object readObject()` zu erkennen. Letztere entstammen aber den Klassen `ObjectOutputStream` und `ObjectInputStream` und nutzen den Typ `Object` als Ein- bzw. Rückgabe. Dahingegen müssen die beiden oben gelisteten Methoden innerhalb von eigenen Klassen implementiert werden und Instanzen der Klassen `ObjectInputStream` bzw. `ObjectOutputStream` als Parameter übergeben bekommen.

```
public class Person implements Serializable
{
    // ...
    private void writeObject(final ObjectOutputStream outStream) throws
          IOException
    {
            // ...
    }

    private void readObject(ObjectInputStream inStream) throws IOException,
          ClassNotFoundException
    {
            // ...
    }
}
```

Der Serialisierungsautomatismus ruft diese Methoden, sofern vorhanden, während der Verarbeitung auf. Die Serialisierung der Attribute möglicher Basis- und Subklassen erfolgt nach wie vor durch den Automatismus der Serialisierung. Die beiden Methoden sind dafür zuständig, *alle* Attribute des zugehörigen Klassenbausteins zu verarbeiten, d. h. in den Stream zu schreiben bzw. daraus zu lesen oder anderweitig zu initialisieren.

Realisierung der Serialisierung Bei der Implementierung der Serialisierung sind alle benötigten Informationen in den Ausgabestream zu schreiben und später wieder korrekt einzulesen. Das bedeutet, dass sowohl alle serialisierbaren Attribute verarbeitet werden müssen als auch eine Spezialbehandlung für die nicht serialisierbaren, transienten Attribute durchgeführt werden muss.

[5] Dabei ist zu beachten, dass die Signatur exakt eingehalten wird, da diese Methoden per Reflection durch den Serialisierungsautomatismus ermittelt und aufgerufen werden.

Für die Klasse `Person` sind vier Attribute zu serialisieren. Ein erster Ansatz ist, dazu die Methoden des Interface `DataOutput` zu benutzen. Dort sind diverse `write()`-Methoden für primitive Datentypen, Byte-Arrays und UTF-8-codierte Strings definiert. Das Interface `DataOutput` wird sowohl von der Klasse `DataOutputStream` als auch von der Klasse `ObjectOutputStream` implementiert. Wenn man die im Interface `DataOutput` bereitgestellte Funktionalität einsetzt, dann sind für jedes Attribut spezifische, typabhängige `write()`- und `read()`-Methoden aufzurufen. Für serialisierbare Objekte kann auch die Methode `writeObject(Object)` eingesetzt werden, wie hier für das `LocalDate`-Objekt:

```
private void writeObject(final ObjectOutputStream outStream) throws IOException
{
    // Verarbeitung aller »normalen« Attribute
    outStream.writeUTF(name);
    outStream.writeUTF(city);
    outStream.writeObject(birthday);

    // Spezialbehandlung des nicht serialisierbaren Attributs
    outStream.writeUTF(nonSerializable.getDatabaseConnectionParams());
}
```

Hier nutzen wir aus, dass sich eine Instanz des nicht serialisierbaren Attributs vom Typ `NonSerializableClass` durch die textuelle Repräsentation der Verbindungsparameter zur Datenbank beschreiben lässt.[6] Beim Speichern schreiben wir diese Information mithilfe der Methode `writeUTF(String)` in den Stream. Auf das Einlesen gehe ich gleich ein, nachdem wir eine Vereinfachung betrachtet haben.

Standardmechanismus aufrufen All dies von Hand zu programmieren ist mühselig und fehleranfällig. Das haben wir bereits eingangs bei der Beschreibung der Serialisierung erkannt. Was kann man also tun? Wünschenswert ist es, den Standardmechanismus für serialisierbare Attribute verwenden zu können und nur für die `transient` definierten Attribute eine Spezialbehandlung durchführen zu müssen. Genau für diesen Anwendungsfall sind in den Klassen `ObjectOutputStream` und `ObjectInputStream` zwei Methoden definiert:

1. Die Methode `defaultWriteObject()` speichert alle nicht `transient` definierten Attribute einer Klasse.
2. Die Methode `defaultReadObject()` liest diese Attribute wieder ein.

Diese Methoden können genutzt werden, um die eigene Realisierung der Serialisierung einfacher zu gestalten. Beide Methoden dürfen nur während einer laufenden Serialisierung aufgerufen werden, d. h. nur aus den beiden zuvor genannten privaten Methoden `readObject(ObjectInputStream)` bzw. `writeObject(ObjectOutputStream)`. Ansonsten wird eine `java.io.NotActiveException` ausgelöst.

[6] In der Praxis ist das häufig ein wenig komplizierter, aber vieles kann man auf wenige zu serialisierende Informationen zurückführen.

Basierend auf diesen Überlegungen ergeben sich in der Klasse `Person` folgende Implementierungen der Methoden `writeObject(ObjectOutputStream)` und `readObject(ObjectInputStream)`:

```
private void writeObject(final ObjectOutputStream outStream) throws IOException
{
    // Behandlung aller »normalen« Attribute
    outStream.defaultWriteObject();

    // NonSerializableClass wird nicht geschrieben, stattdessen lediglich
    // Informationen, die zur Rekonstruktion benötigt werden
    outStream.writeUTF(nonSerializable.getDatabaseConnectionParams());
}

private void readObject(final ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
{
    // Behandlung aller »normalen« Attribute
    inStream.defaultReadObject();

    // Rekonstruktion der NonSerializableClass aus dem gelesenen String
    final String databaseConnectionParams = inStream.readUTF();
    nonSerializable = new NonSerializableClass(databaseConnectionParams);
}
```

Beim Einlesen der Informationen zur Rekonstruktion der nicht serialisierbaren Klasse nutzen wir die Methode `readUTF()`. Anschließend erzeugen wir per Konstruktor ein neues Objekt vom Typ `NonSerializableClass`.

Hinweis Zur Verarbeitung von Attributen werden Zuweisungen in der Methode `readObject()` benötigt. Dadurch können diese Attribute nicht `final` sein. Dies stellt eine Einschränkung beim Design dar. Insbesondere kann eine Klasse dann möglicherweise nicht mehr als unveränderliche Klasse realisiert werden.

12.3.3 Versionsverwaltung der Serialisierung

Der Automatismus der Serialisierung sorgt dafür, dass nur miteinander kompatible Versionsstände einer Klasse verarbeitet werden können. Wie bereits beschrieben, ändert sich die dazu verwendete, automatisch berechnete Versionskennung `serialVersionUID` bei jeder Änderung an der Struktur der Klasse. Dieses Standardverhalten ist oft unpraktisch, weil es dadurch schnell zu einer Inkompatibilität zwischen semantisch gleichen Versionen eines Objekts kommt. Als Abhilfe kann man explizit ein privates statisches Attribut `serialVersionUID` definieren und mit einem fixen Wert belegen. Wenn Änderungen an Methoden und Attributen der Klasse vorgenommen werden, wird diese Versionskennung dann *nicht* automatisch neu berechnet, sondern bleibt konstant. Betreffen die strukturellen Änderungen auch die serialisierten Daten, so sollte die `serialVersionUID` vom Entwickler modifiziert werden. Damit hat man zwar das Problem der Inkompatibilität semantisch gleicher Klassenstände gelöst, allerdings ist damit über die Serialisierung immer noch kein Datenaustausch zwischen einer alten und einer neuen Version möglich.

Wenn aber eine Rückwärtskompatibilität zu erhalten ist und verschiedene Versionen zu unterstützen sind, dann muss der Serialisierungsvorgang für jede Version einzeln selbst programmiert werden. Zudem ist erforderlich, dass sich mehrere serialisierte Versionen eindeutig im Datenstrom identifizieren lassen. Dazu verwenden wir eine eigene Versionsverwaltung und fixieren zudem den Wert der `serialVersionUID` beim Erstellen der Klasse einmalig. Das erlaubt selbst bei strukturellen Änderungen an der Klasse deren sichere Identifikation im Datenstrom. Zur Versionsverwaltung verwenden wir ein Attribut `CLASS_VERSION`, das in den Methoden `readObject(ObjectInputStream)` und `writeObject(ObjectOutputStream)` ausgewertet wird.

Da wir sowohl die Versionsverwaltung als auch die Serialisierung der Objekte vollständig selbst realisieren, müssen *alle* Attribute `transient` definiert werden. Daher lassen sich die praktischen Methoden `defaultWriteObject()` bzw. `defaultReadObject()` nicht mehr nutzen, da diese nur nicht transiente Attribute verarbeiten.

Im folgenden Listing ist die Klasse `PersonVersion1` gezeigt, die die erste Version einer Klasse `Person` darstellen soll. Vor den eigentlichen Nutzdaten wird eine Versionsnummer in Form eines `int` in den Stream geschrieben. Anschließend werden alle Attribute manuell gespeichert (durch Aufruf entsprechender `write()`-Methoden). Beim späteren Deserialisieren erfolgen die korrespondierenden Schritte zum Einlesen. Hierbei wird zudem die Versionsnummer geprüft. Nur für den Fall eines gültigen Werts findet das Einlesen statt. Ansonsten wird eine `ClassNotFoundException` ausgelöst:

```
public final class PersonVersion1 implements Serializable
{
   // durch beliebige, immer gleichbleibende Kennung
   // Berechnungsautomatismus deaktivieren
   private static final long serialVersionUID = 1L;

   // eigene Kennung der Klassenversion
   private static final long CLASS_VERSION = 1L;

   private transient String name;
   private transient String city;
   private transient LocalDate birthday;

   public PersonVersion1(final String name, final String city,
                         final LocalDate birthday)
   {
      this.name = name;
      this.city = city;
      this.birthday = birthday;
   }

   private void writeObject(final ObjectOutputStream outStream) throws
         IOException
   {
      // eigene Versionsinformation schreiben
      outStream.writeLong(CLASS_VERSION);

      // Attributdaten schreiben
      outStream.writeUTF(name);
      outStream.writeUTF(city);
      outStream.writeObject(birthday);
   }
```

```
   private void readObject(final ObjectInputStream inStream) throws IOException,
                                                                   ClassNotFoundException
   {
      // eigene Versionsinformation lesen
      final long version = inStream.readLong();

      if (version == CLASS_VERSION)
      {
          // Attributdaten lesen
          name = inStream.readUTF();
          city = inStream.readUTF();
          birthday = (LocalDate) inStream.readObject();
      }
      else
      {
          // Diese Version unterstützt keine anderen Formate
         throw new ClassNotFoundException("Unsupported version " + version);
       }
   }
```

Nehmen wir an, wir hätten die genannten Schritte für die Klasse `Person` durchgeführt und bei einer nachfolgenden Erweiterung der Klasse (Speicherung der Augenfarbe) wäre die interne Klassenversion hochgezählt worden:

```
public final class PersonVersion2 implements Serializable
{
    // beliebige, immer gleichbleibende Kennung
    private static final long serialVersionUID = 1L;

    // eigene Kennung der Klassenversion
    private static final long CLASS_VERSION = 2L;

    private transient String name;
    private transient String city;
    private transient LocalDate birthday;
    private transient Color eyeColor;

    // ...
```

Um die zwei Versionen deutlich voneinander zu unterscheiden, wurde eine Optimierung der Speicherung in das Datenformat in die Version 2 übernommen. Diese besteht darin, das `LocalDate`-Objekt als `String` in der normierten Form, beispielsweise `2019-02-07`, zu speichern und anstatt des `Color`-Objekts lediglich die drei Farbwerte für Rot, Grün und Blau als `int` zu verwenden. Darauf gehe ich im nachfolgenden Abschnitt noch mal genauer ein.

```
private void writeObject(final ObjectOutputStream outStream) throws IOException
{
    // Versionsinformation schreiben
    outStream.writeLong(CLASS_VERSION);

    // Daten
    outStream.writeUTF(name);
    outStream.writeUTF(city);

    // Optimierte Darstellung für LocalDate
    outStream.writeUTF(birthday.toString());
```

```
    // Optimierte Darstellung für Color
    outStream.writeInt(eyeColor.getRed());
    outStream.writeInt(eyeColor.getGreen());
    outStream.writeInt(eyeColor.getBlue());
}
```

Das Einlesen ist etwas komplizierter als das Schreiben, da beim Lesen verschiedene Versionen unterschieden und behandelt werden müssen. In folgender Methode `readObject(ObjectInputStream)` wird dazu zunächst die Versionsnummer aus dem Stream gelesen und anschließend der jeweilige Sourcecode zum Einlesen ausgeführt:

```
private void readObject(final ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
{
    // Versionsinformation ermitteln
    final long version = inStream.readLong();

    // versionsabhängiges Einlesen
    if (version == CLASS_VERSION)
    {
        readDataInCurrentVersion(inStream);
    }
    else if (version == 1)
    {
        readDataOfVersion1(inStream);
    }
    else
    {
        throw new ClassNotFoundException("Unsupported version " + version);
    }
}

private void readDataOfVersion1(final ObjectInputStream inStream) throws
      IOException, ClassNotFoundException
{
    name = inStream.readUTF();
    city = inStream.readUTF();
    birthday = (LocalDate) inStream.readObject();
    eyeColor = Color.BLUE; // Vereinfachung für Beispiel: Defaultwert Blau
}

private void readDataInCurrentVersion(ObjectInputStream inStream) throws
      IOException
{
    name = inStream.readUTF();
    city = inStream.readUTF();

    final String birthdayAsText = inStream.readUTF();
    birthday = LocalDate.parse(birthdayAsText);

    final int red = inStream.readInt();
    final int green = inStream.readInt();
    final int blue = inStream.readInt();
    eyeColor = new Color(red, green, blue);
}
// ...
```

Durch Implementierung dieser beiden Methoden können wir die Serialisierung beliebig anpassen. Damit können verschiedene Versionen eines Objekts verwaltet werden. Es ist relativ aufwendig, die Kompatibilität von Versionen sicherzustellen, dies wird hier nicht weiter behandelt. In diesem Beispiel habe ich stark vereinfachend die Augenfarbe auf Blau gesetzt, da diese Information in einer älteren Version nicht verfügbar ist.

12.3.4 Optimierung der Serialisierung

Bislang haben wir den Standardmechanismus der Serialisierung eingesetzt bzw. nur ein wenig angepasst. Zum Teil kann es zur Performance-Optimierung sinnvoll sein, die Datenmenge gegenüber dem Standardformat zu reduzieren. Dazu müssen Änderungen am Format der herausgeschriebenen Informationen vorgenommen werden.

Um ein besseres Gespür für Optimierungsmöglichkeiten zu bekommen, verwenden wir die eingangs vorgestellte Klasse `Person` als Basis. Zunächst fügen wir wieder die Augenfarbe als Attribut in Form eines `Color`-Objekts hinzu und nennen diese Klasse `PersonWithEyeColorV1`. Folgende `main()`-Methode zeigt die Erzeugung eines Objekts und dessen Speicherung in der Datei `TestWithEyeColor1.ser`:

```
public static void main(final String[] args) throws IOException
{
    final PersonWithEyeColorV1 original = new PersonWithEyeColorV1("Test",
                                "TestCity", LocalDate.now(), Color.GREEN);

    try (final ObjectOutputStream objectOutStream = new ObjectOutputStream(
      new BufferedOutputStream(new FileOutputStream("TestWithEyeColor1.ser"))))
    {
        // Schreibe Objekt in die Datei
        objectOutStream.writeObject(original);
        System.out.println("Wrote to stream: " + original);
    }
}
```

Listing 12.10 *Ausführbar als* **'SERIALIZATIONOPTIMIZATIONEXAMPLE'**

Serialisiert man diese Klasse durch Aufruf des Programms SERIALIZATIONOPTIMIZATIONEXAMPLE, so wird die entstehende Datei `TestWithEyeColor1.ser` 360 Bytes groß. Um Optimierungspotenziale aufzudecken, öffnen wir die Datei mit einem HEX- oder Texteditor. Wir erkennen, dass die Speicherung für das Datum bzw. die Farbinformation noch relativ viele, für diesen Anwendungsfall irrelevante Informationen (voll qualifizierten Klassenname, Transparenz usw.) in die Datei schreibt:

```
<AC><ED>^@^Esr^@3ch12_advancedjava.serializable.PersonWithEyeColorV1<8B><BD><F5>
      lz<D4>p;^B^@^DL^@^Hbirthdayt^@^ULjava/time/LocalDate;L^@^Dcityt^@^RL
      java/lang/String;L^@^HeyeColort^@^PLjava/awt/Color;L^@^Dnameq^@~^@^Bxpsr^@^
      Mjava.time.Ser<95>]<84><BA>ESC"H<B2>^L^@^@xpw^G^C^@^@^G<E3>^D^Ext^@^
      HTestCitysr^@^Njava.awt.Color^A<A5>^W<83>^P<8F>3u^B^@^EF^@^FfalphaI^@^
      EvalueL^@^Bcst^@ESCL/java/awt/color/ColorSpace;[^@ frgbvaluet^@^B[F[^@^
      Ffvalueq^@~^@
xp^@^@^@^@<FF>^@<FF>^@pppt^@^DTest
```

Optimierung durch eigene Repräsentationsform

Anstelle eines `Color`-Objekts werden lediglich die Farbwerte Rot, Grün und Blau zur Modellierung der Augenfarbe benötigt. Auch die Repräsentation des gespeicherten Geburtsdatums scheint zu umfangreich. Für beide Attribute ist eine Darstellung durch die primitiven Datentypen `int` sowie das Datum als `String` relativ einfach möglich. Minimal aufwendiger als das Schreiben wird die Rekonstruktion korrespondierender `LocalDate`- bzw. `Color`-Objekte. Folgendes Listing zeigt die dazu notwendigen Realisierungen der Methoden `writeObject(ObjectOutputStream)` und `readObject(ObjectInputStream)`:

```
// Achtung: falsche Lösung
private void writeObject(final ObjectOutputStream outStream) throws IOException
{
    outStream.defaultWriteObject();

    outStream.writeUTF(birthday.toString());
    outStream.writeInt(eyeColor.getRed());
    outStream.writeInt(eyeColor.getGreen());
    outStream.writeInt(eyeColor.getBlue());
}

private void readObject(final ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
{
    inStream.defaultReadObject();

    birthday = LocalDate.parse(inStream.readUTF());
    eyeColor = new Color(inStream.readInt(), inStream.readInt(),
                         inStream.readInt());
}
```

Serialisiert man diese Klasse durch Aufruf des nicht als Listing gezeigten Programms SERIALIZATIONOPTIMIZATIONEXAMPLE2, so entsteht die Datei `TestWithEyeColor2.ser`. Diese benötigt 387 Bytes Speicherplatz. Offensichtlich ist diese Lösung sogar noch schlechter als der Standard! Wie kann das denn sein?

Die Antwort ist einfach: Es wird zunächst die Standardserialisierung vorgenommen und dann werden zusätzlich die Informationen der primitiven Daten in den Stream geschrieben. Dieser Flüchtigkeitsfehler ist schnell gemacht, aber auch schnell behoben. Soll zwar der Standardmechanismus für eine korrekte Objektserialisierung der `String`-Attribute sorgen, aber sowohl das `LocalDate`- als auch das `Color`-Objekt selbst behandelt werden, so müssen wir diese natürlich auch `transient` definieren:

```
public class PersonWithEyeColorV3 implements Serializable
{
    private String name;
    private String city;

    // beide komplexeren Attribute nicht vom Standard behandeln lassen
    private transient LocalDate birthday;
    private transient Color eyeColor;

   // ...
```

Führen wir die nicht als Listing gezeigte Korrektur des Programms SERIALIZATIONOPTIMIZATIONEXAMPLE3 aus, so entsteht die Datei `TestWithEyeColor3.ser`, die nur noch 157 Bytes groß ist.

Mit dieser modifizierten Form der Speicherung haben wir einiges an Speicherplatz eingespart. ***Für ein einzelnes Objekt lohnen sich derartige Mühen nicht***. Werden aber umfangreichere Datenstrukturen serialisiert, so kann eine eigene Repräsentation sinnvoll sein. Auch bei einer Übertragung über ein Netzwerk können so Verbesserungen in der Übertragungsgeschwindigkeit erzielt werden.

Hinweis: Serialisierbare Attribute vorgeben

Zwar ist die Standardserialisierung bereits für diverse Anwendungsfälle ausreichend, aber eben nicht für alle. Daher kann man die Serialisierung in verschiedenen Stufen anpassen, wie wir dies bereits kennengelernt haben. Im Normalfall bestimmt der Serialisierungsautomatismus die zu verarbeitenden Attribute und deren Werte mithilfe von Reflection. Sind nun die Attribute überwiegend transient, so bietet es sich an, die tatsächlich zu serialisierenden Attribute anzugeben. Dazu muss man eine statische Variable mit dem Namen `serialPersistentFields` in die Klassendefinition aufnehmen, etwa wie folgt für zwei Attribute `attr1` und `attr2` mit den Typen `attr1Type` und `attr2Type`. Diese in `ObjectStreamField` angegebenen Typen müssen natürlich serialisierbar sein.

```
private static final ObjectStreamField[] serialPersistentFields =
{
    new ObjectStreamField("attr1", attr1Type.class);
    new ObjectStreamField("attr2", attr2Type.class);
};
```

Das Interface `Externalizable`

Manchmal wird eine noch weiter gehende Einflussnahme des Serialisierungsvorgangs benötigt, als dies über die Methoden `readObject(ObjectInputStream)` und `writeObject(ObjectOutputStream)` möglich ist. Für diese Anwendungsfälle ist das Interface `java.io.Externalizable` zu implementieren. Über dieses Interface, das das Interface `Serializable` erweitert, kann eine Variante der Serialisierung beschrieben werden, die es dem Entwickler erlaubt, das Datenformat vollkommen selbst zu bestimmen, d. h. festzulegen, welche Attribute in welcher Form verarbeitet (geschrieben und gelesen) werden. Dadurch lässt sich beispielsweise das Datenformat noch kompakter gestalten. Allerdings müssen dazu auch die Attribute der Basisklassen selbst verarbeitet werden. Das Interface `Externalizable` definiert dazu folgende zwei Methoden:

```
public interface Externalizable extends java.io.Serializable
{
   void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
   void writeExternal(ObjectOutput out) throws IOException;
}
```

Wir entfernen das zuvor eingeführte Schlüsselwort `transient` für die Attribute und lassen die Klasse `Person` nun das Interface `Externalizable` implementieren:

```
public final class PersonExternalizable implements Externalizable
{
    private String name;
    private String city;
    private LocalDate birthday;
    private Color eyeColor;
    // ...
```

Durch die Implementierung von `Externalizable` wird man vom Datenformat der Serialisierung unabhängig. Dies ist insbesondere dann von Vorteil, wenn keine Objektgraphen, sondern lediglich flache Objekte verarbeitet werden sollen. Dann können Speicherplatz sparende Dateiformate gewählt werden. Schauen wir uns nun die Realisierung der beiden Methoden an, die die schon vorgestellte Optimierung der Repräsentation von Objekten über primitive Werte ihrer relevanten Daten realisiert:

```
@Override
public void writeExternal(final ObjectOutput objectOut) throws IOException
{
    objectOut.writeUTF(name);
    objectOut.writeUTF(city);

    objectOut.writeUTF(birthday.toString());

    objectOut.writeInt(eyeColor.getRed());
    objectOut.writeInt(eyeColor.getGreen());
    objectOut.writeInt(eyeColor.getBlue());
}

@Override
public void readExternal(final ObjectInput objectIn) throws IOException,
      ClassNotFoundException
{
    name = objectIn.readUTF();
    city = objectIn.readUTF();

    birthday = LocalDate.parse(inStream.readUTF());

    eyeColor = new Color(objectIn.readInt(), objectIn.readInt(),
                         objectIn.readInt());
}
```

Weil das Interface `Externalizable` das Interface `Serializable` erweitert, könnte man die Klasse `PersonExternalizable` wie in den vorherigen Beispielen verarbeiten. Schauen wir uns dazu einen Start des Programms SERIALIZATIONFOREXTERNALIZABLECLASSEXAMPLE an. Die so erzeugte Datei `TestSerializationForExternalizableClass.ser` ist nur noch 115 Bytes groß. Das ist eine deutliche Reduktion. Aber es geht noch viel besser!

Betrachten wir, was passiert, wenn man das Ganze in die eigene Hand nimmt und auf die Serialisierungsautomatik verzichtet. Dann nutzt man Aufrufe der Methoden `readExternal(ObjectInput)` und `writeExternal(ObjectOutput)` aus dem Interface `Externalizable`. Werfen wir einen Blick auf die korrespondierende

`main()`-Methode, die ein Objekt vom Typ `PersonExternalizable` ohne Einbindung in die Serialisierungsautomatik speichert und wieder einliest:

```
public static void main(final String[] args) throws IOException,
      ClassNotFoundException
{
   final PersonExternalizable original = new PersonExternalizable("Test",
                                       "TestCity", LocalDate.now(), Color.GREEN);

   final String filename = "TestExternalizable.ser";

   try (final ObjectOutputStream objectOutStream = new ObjectOutputStream(
        new BufferedOutputStream(new FileOutputStream(filename))))
   {
       // Schreibe Objekt mit writeExternal() in die Datei
       original.writeExternal(objectOutStream);
       System.out.println("Wrote to stream: " + original);
   }

   try (final ObjectInputStream objectInStream = new ObjectInputStream(
        new BufferedInputStream(new FileInputStream(filename))))
   {
       // Rücklesen des Objekts: Konstruktoraufruf und readExternal()
       final PersonExternalizable readInObject = new PersonExternalizable();
       readInObject.readExternal(objectInStream);
       System.out.println("Back from stream: " + readInObject);
   }
}
```

Listing 12.11 *Ausführbar als* '**EXTERNALIZABLEEXAMPLE**'

Dieser Sourcecode ist recht ähnlich zu dem der Serialisierung. Allerdings muss im Gegensatz dazu ein Konstruktoraufruf der einzulesenden Klasse erfolgen.[7] Für die neue Instanz wird die Methode `readExternal(ObjectInput)` aufgerufen. Der entscheidende Unterschied liegt darin, dass es sich bei den beiden Methoden des Interface `Externalizable` um Objektmethoden handelt, denen ein Stream übergeben wird. Bei der Serialisierung sind die Zuständigkeiten genau andersherum geregelt: Ein spezieller Stream bekommt eine Objektreferenz, die er verarbeitet.

Das obige Programm EXTERNALIZABLEEXAMPLE erzeugt die Datei `TestExternalizable.ser`, die lediglich noch einen Speicherbedarf von 46 Bytes besitzt. Im Vergleich zum Ausgangspunkt der Optimierung der Serialisierung mit 360 Bytes ist dies nur noch $1/8$ des Speicherplatzes. Bezogen auf die letzte optimierte Version der Serialisierung mit 157 Bytes wird der Speicherbedarf auf rund $1/3$ reduziert.

Fazit

Die Reduktion des Datenvolumens geht mit einem erhöhten Aufwand bei der Implementierung und insbesondere bei einer späteren Wartung des Speicherns und Einlesens einher. Derartige Lösungen sind deshalb nur bei Anwendungen sinnvoll einzusetzen, die kritisch bezüglich Performance oder Speicherplatz sind.

[7] Daher besitzen Klassen, die `Externalizable` erfüllen, oftmals einen Defaultkonstruktor.

12.4 Garbage Collection

Während der Ausführung eines Java-Programms werden in der Regel neue Objekte angelegt. Dadurch steigt der Speicherverbrauch eines Programms kontinuierlich. Um diesem Trend entgegenzuwirken, besitzt die JVM ein automatisches Speicherbereinigungssystem, das den nicht mehr durch Objekte benötigten Speicher wieder freigibt. Alle noch referenzierten Objekte müssen dabei natürlich erhalten bleiben. Erlischt allerdings die letzte Referenz auf ein Objekt, so kann dieses eingesammelt und der dadurch belegte Speicherplatz freigegeben werden. Dieser Aufräumvorgang wird als ***Garbage Collection*** (GC) bezeichnet und findet im Hintergrund parallel zur eigentlichen Applikation statt. Die ausführende Einheit wird ***Garbage Collector*** genannt.

Durch diesen Automatismus werden Probleme verhindert, die in Programmiersprachen mit expliziter Speicherfreigabe durch zu früh oder niemals freigegebene Objekte entstehen. Allerdings benötigt die Garbage Collection einen gewissen Anteil der Rechenzeit: In vielen Applikationen ist dieser Mehraufwand kaum von Relevanz. In Echtzeitsystemen kann dieser Overhead jedoch problematisch werden, da kaum Aussagen, geschweige denn Garantien zu den einzelnen Zeitpunkten und den jeweiligen Laufzeiten von Garbage-Collection-Vorgängen gemacht werden können.

12.4.1 Grundlagen zur Garbage Collection

Bei der Abarbeitung eines Programms werden normalerweise dynamisch neue Objekte erzeugt. Erreicht die Speicherbelegung einen gewissen Grenzwert, so erfolgt eine Garbage Collection, um wieder für mehr freien Speicher zu sorgen.

Freizugebene Objekte erkennen und beseitigen

Vor einer Freigabe müssen überhaupt erst einmal die möglichen Kandidaten dafür ermittelt werden. Für die Garbage Collection sind immer folgende Aufgaben zu erfüllen:

1. **Garbage-Erkennung** – Zunächst sind alle Objekte ausfindig zu machen, die freigegeben werden können. Dies ist dann der Fall, wenn Objekte von keinem anderen Objekt mehr referenziert werden. Die folgende Hintergrundinfo »Erreichbarkeit von Objekten« geht darauf genauer ein.
2. **Garbage-Beseitigung** – Es muss der durch diese unreferenzierten Objekte belegte Speicher dem Programm wieder zur Verfügung gestellt werden. Zuvor wird Objekten noch die Möglichkeit gegeben, letzte Aufräumarbeiten durchzuführen. Dazu existiert die Methode `finalize()`, die automatisch durch den Garbage Collector aufgerufen wird, bevor das Objekt entfernt wird.

Hintergrundinfo: Erreichbarkeit von Objekten

Der Automatismus der Speicherbereinigung setzt voraus, dass man nicht mehr referenzierte Objekte sicher erkennen kann. Ein (naives) Zählen von Referenzen auf Objekte, das sogenannte ***Reference Counting***, ist dazu nicht ausreichend, weil damit zyklische Referenzen nicht zu erkennen sind.

Die JVM berechnet daher stattdessen die ***Erreichbarkeit*** durch Nachverfolgen von Objektreferenzen. Objekte sind erreichbar, wenn es im Programm eine Referenz von einer sogenannten Wurzelreferenz über beliebig viele Indirektionen gibt. Alle derart erreichbaren Objekte sind lebendig. Alle anderen können freigegeben werden, da sie unreferenziert sind und keinen Einfluss auf nachfolgende Berechnungen im Programm mehr besitzen.

Die Ausgangsmenge der Wurzelreferenzen setzt sich wie folgt zusammen:

- Alle Referenzen aus lokalen Variablen im momentanen Stack
- Alle Referenzen, die von lebenden Threads ausgehen
- Alle Referenzen in statischen Attributen[a]

[a] Nicht nur aus OO-Sicht sind statische Attribute möglichst zu vermeiden. Diese wirken sich negativ auf die Laufzeit von Garbage Collections aus.

Lebensdauer von Objekten

Zur Optimierung der Speicherverwaltung nutzt man die Erkenntnis, dass die Lebensdauern von Objekten stark variieren, sich aber grob in zwei Kategorien einteilen lassen: *kurzlebige* Objekte und *langlebige* Objekte. Die große Masse an Objekten besitzt nur eine geringe Lebenszeit, da diese Objekte nur temporär zur Durchführung von Aufgaben erzeugt und benötigt werden. Einige wenige davon leben länger. Daneben findet man auch eine von Applikation zu Applikation variierende Anzahl an langlebigen Objekten – meistens sind dies die zentralen Komponenten des Programms und zum Teil auch andere Daten. Man kann daher Objekte gemäß ihrem Alter grob in eine ***junge*** und eine ***alte Generation*** (***Young Generation*** und ***Old Generation***) aufteilen.

Automatik vs. manueller Aufruf

In der Regel befreit die Automatik der Speicherbereinigung den Programmierer davon, sich allzu viele Gedanken über die Speicherverwaltung machen zu müssen. Als Programmierer hat man kaum Einfluss auf die Garbage Collection, weder auf den Zeitpunkt noch auf den Umfang.

Normalerweise wird die Garbage Collection automatisch aufgerufen, falls Speicher freigegeben und wieder bereitgestellt werden muss. Darüber hinaus kann man zur Laufzeit eines Programms den Wunsch zur Ausführung einer Garbage Collection explizit durch einen Aufruf der statischen Methode `System.gc()` »äußern«. Tatsächlich löst

ein Aufruf von `System.gc()` die Garbage Collection nicht in jedem Fall aus. Der JVM steht es frei, diese Aufforderung zu ignorieren. Ein expliziter Aufruf von `System.gc()` ist in der Regel auch überflüssig und sollte daher unterbleiben. Eine Ausnahme bilden Situationen, in denen zu einem bestimmten Zeitpunkt so viel Speicher wie möglich freigegeben werden soll, um anschließend einen größeren freien Speicherbereich bereitstellen zu können. Da die Aufrufe von `System.gc()` aber potenziell ignoriert werden können, muss man die Aufrufe normalerweise mehrfach wiederholen.

Oftmals ist es sinnvoller, dem Garbage Collector indirekt dadurch zu helfen, dass man Referenzvariablen wieder auf `null` setzt – mit dem Ziel, dass diese damit nicht bis in die Old Generation »überleben« und noch vorher entfernt werden können. Gleiches gilt für Collections: Werden größere Datenbestände nicht mehr benötigt, so sollte man diese möglichst schnell aufräumen, etwa indem diese durch einen Aufruf von `remove()` oder `clear()` aus der Collection entfernt werden.

12.4.2 Der Garbage Collector »G1«

Die früher in der JVM (von Sun) verwendeten Garbage-Collection-Algorithmen teilen den Heap in unterschiedliche Bereiche für kurz lebende und lang lebende Objekte ein. Seit JDK 9 wurde ein neuer Garbage Collector namens G1 (als Abkürzung für »Garbage First«) als Standard eingeführt, der eine andere Speicheraufteilung verwendet und die Schwachpunkte der bisherigen Verfahren adressiert. Dazu erlaubt er es, mehr Einfluss auf die Garbage Collection zu nehmen und im Speziellen die Unterbrechungen für Aufräumarbeiten kurz zu halten.

Arbeitsweise

Der G1-Collector funktioniert wie folgt: Der Heap wird in eine Reihe von fixen Teilbereichen zerlegt. Zu jedem Teilbereich wird eine Liste mit Referenzen von Objekten geführt. Diese wird »Remember Set« genannt und enthält Einträge von Referenzen auf die Objekte in diesem Bereich. Veränderungen von Referenzen werden durch die JVM registriert und führen zu einer Anpassung der Remember Sets.

Wenn eine Garbage Collection notwendig wird, werden zunächst die Regionen aufgeräumt, in denen sich nur wenig »lebende«, d. h. weiterhin referenzierte Objekte befinden, in denen sich also viel »Müll« angesammelt hat. Daher stammt auch der Name »Garbage First«. Im besten Fall kann ein ganzer Teilbereich ohne großen Aufwand freigegeben werden, wenn das Remember Set leer ist. Zudem besteht der Vorteil, dass »Müll« leicht erkannt werden kann: Existieren keine eingehenden Referenzen, so ist kein aufwendiges Markieren und Kompaktieren (Mark-and-Sweep) notwendig, sondern ein Bereich kann dann einfach als frei betrachtet werden. Außerdem kompaktiert der G1, indem er alle aktiven Objekte an das Ende des Heaps bewegt. Dadurch wird einer Fragmentierung entgegengewirkt und es steht immer ein möglichst großes Stück freier Speicher zur Verfügung.

12.5 Dynamic Proxies

In diesem Abschnitt möchte ich Ihnen dynamische Proxys als spezielle Form von Proxys vorstellen. Rekapitulieren wir kurz, dass man unter einem Proxy einen Stellvertreter für ein anderes Objekt versteht. Nutzer wenden sich immer an den Proxy und nicht direkt an das vertretene Objekt. Das erlaubt dem Proxy zu kontrollieren bzw. zu steuern, wann, wie und von wem auf das vertretene Objekt zugegriffen wird. Damit ein Proxy wie das Objekt selbst angesprochen werden kann, muss der Proxy all diejenigen öffentlichen Methoden implementieren, die das ursprüngliche Objekt in seiner Schnittstelle definiert, nachfolgend in Form von `IService`. Diese Schnittstelle stellt sicher, dass sowohl der Proxy in Form der Klasse `ServiceProxy` als auch das eigentliche Objekt vom Typ `Service` nach außen gleich behandelt werden können.

Abbildung 12-3 *Schematische Struktur eines Proxys*

Der Einsatz eines Proxys ermöglicht es, zusätzlich zum Verhalten der Methoden eines Objekts durch den Proxy verschiedene weitere Aktionen auszuführen. Ein Proxy kann die Aktionen sowohl vor als auch gegebenenfalls nach der Delegation des Methodenaufrufs ausführen. Dadurch ist es möglich, Funktionalität zu ergänzen und sogar zu entfernen (indem ein Methodenaufruf nicht weitergeleitet wird). Wie in Abschnitt 18.3.6 zum Entwurfsmuster PROXY genauer beschrieben, sind die Anwendungsgebiete von Proxys vielfältig und reichen von Performance-Messungen, Logging oder Caching bis hin zu Berechtigungsprüfungen. Diese Funktionalität wird für den Anwender durch den Proxy transparent bereitgestellt.

Mit Proxys kann man verschiedene Arten der Stellvertretung realisieren, etwa Remote Proxy als Stellvertreter für Objekte, die über ein Netzwerk (remote) angesprochen werden sollen, oder Virtual Proxy als Platzhalter, etwa beim Zugriff auf Objekte, die aufwendig zu konstruieren sind. Nachfolgend schauen wir aber mehr auf die Ausprägung als Decorator bzw. Interceptor. Dieser Typ von Proxy fügt transparent weitere Funktionalität gemäß dem DEKORIERER-Muster (vgl. Abschnitt 18.2.3) hinzu. Häufig spricht man in diesem Kontext auch von Interceptoren.

Wenn man einen Proxy selbst implementiert, sind schnell relativ viele Methoden zu schreiben. Man spricht dann auch von einem statischen Proxy, da dieser bereits während der Kompilierung vorliegt. Manchmal ist es aber wünschenswert, einen Proxy zu einem bestimmten Interface erst zur Laufzeit zu erstellen. Einen solchen bezeichnet man dann als dynamischen Proxy. Damit diese dynamische Erstellung zur Laufzeit jedoch funktionieren kann, bedarf es verschiedener Spezialitäten von Java.

Nach diesen etwas theoretischen Vorüberlegungen wollen wir das Ganze aus dem Blickwinkel der Praxis an einem konkreten Beispiel betrachten.

Basisfunktionalität

Beginnen wir mit dem zu vertretenden Objekt, das durch die Klasse `Service` modelliert wird und dessen Business-Funktionalität durch das Interface `IService` beschrieben wird.[8] Eine recht simple Implementierung ist in folgendem Listing gezeigt:

```
public interface IService
{
    public void doSomething();
    public String calculateSomething(final int value);
}

public class Service implements IService
{
    @Override
    public void doSomething()
    {
        System.out.println("doSomething");
    }

    @Override
    public String calculateSomething(final int value)
    {
        System.out.println("calculateSomething");
        try
        {
             TimeUnit.SECONDS.sleep(3);
        }
        catch (final InterruptedException e)
        {
            // can't happen here, no other thread to interrupt us
        }

        return "" + value;
    }
}
```

Die Implementierung der Business-Funktionalität ist so weit vereinfacht und nicht von weiterem Interesse, sodass wir uns hier nachfolgend voll auf die Realisierung von Proxys konzentrieren können.

[8] Bezüglich der Namensgebung findet man auch einige Variationen, etwa Interfaces mit einem abschließenden `IF` oder auch gar keiner expliziten Kennzeichnung. Die implementierende Klasse kann dann den Zusatz `Impl` erhalten. Ich bevorzuge die gewählte Namensgebung, was jedoch Geschmackssache ist und was Sie im Team abstimmen sollten.

12.5.1 Statischer Proxy

Ziel ist es nun, einen statischen Proxy zu realisieren, um Zeitmessungen vorzunehmen. Diesen realisieren wir im Verlauf dieses Abschnitts in der Klasse `ServicePerformanceProxy`. Wie schon erwähnt, muss diese Klasse das Interface `IService` implementieren. Die Proxy-Funktionalität der Zeitmessung muss dort (leider) in jeder Methode erneut realisiert werden, was nicht wirklich elegant ist:

```
public class ServicePerformanceProxy implements IService
{
    private final IService service;

    ServicePerformanceProxy(final IService service)
    {
        this.service = service;
    }

    @Override
    public void doSomething()
    {
        final long startTime = System.nanoTime();

        service.doSomething();

        printExecTime("doSomething", System.nanoTime() - startTime);
    }

    @Override
    public String calculateSomething(int value)
    {
        final long startTime = System.nanoTime();

        final String result = service.calculateSomething(value);

        printExecTime("calculateSomething", System.nanoTime() - startTime);
        return result;
    }

    private void printExecTime(final String methodName, final long duration)
    {
        System.out.println("Method call of '" + methodName + "' took: " +
                           TimeUnit.NANOSECONDS.toMillis(duration) + " ms");
    }
}
```

Betrachten wir kurz, wie wir den Proxy zur Performance-Messung einsetzen:

```
public static void main(final String[] args)
{
    final IService service = createService();
    service.calculateSomething(42);
}

private static IService createService()
{
    final IService service = new Service();
    return new ServicePerformanceProxy(service);
}
```

Listing 12.12 *Ausführbar als* '**STATICPROXYEXAMPLE**'

Wir haben einen einfachen Proxy zur Zeitmessung implementiert und wollen nun einen kritischen Blick darauf werfen und Verbesserungspotenziale erkennen: Je umfangreicher die zu vertretende Klasse oder auch die durch den Proxy realisierte Funktionalität wird und je mehr Methoden derart implementiert werden müssen, desto aufwendiger wird das Ganze. Insbesondere ist viel »mechanisches«, immer wiederkehrendes Implementieren dabei. Darüber hinaus muss jede Interface-Änderung an der Originalklasse im Proxy nachgeführt werden.

Wünschenswert ist es, Proxys auch einfacher, mit weniger Implementierungsaufwand erstellen zu können. Seit JDK 1.3 ist dies mithilfe von dynamischen Proxys möglich. Ganz nebenbei lassen sich so auch Proxys realisieren, ohne dass diese zur Kompilierzeit vorliegen müssen. Wie geht das Ganze?

Hinweis: Vereinfachung mit Java 8

In der Implementierung der Klasse `ServicePerformanceProxy` entstand recht viel ähnlicher Sourcecode. Seit Java 8 gibt es das ***Execute-Around-Pattern***, bei dem man vor und nach einer durch ein `Callable<V>` modellierten Aktion einige Anweisungen ausführt. Dies wird nachfolgend in der Methode `measuredExecute()` realisiert, die dann in den jeweiligen Methoden mit dem auszuführenden Sourcecode-Fragment aufgerufen wird, wie dies im Listing gekürzt gezeigt ist:

```
@Override
public String calculateSomething(final int value)
{
    return measuredExecute(() -> original.calculateSomething(value),
                           "calculateSomething");
}

private <V> V measuredExecute(final Callable<V> callable,
                              final String methodName)
{
    final long startTime = System.nanoTime();

    V result = null;
    try
    {
        result = callable.call();
    }
    catch (final Exception e)
    {
        System.out.println("exception in call of '" + methodName +
                           "': " + e.getMessage());
    }

    printExecTime(methodName, System.nanoTime() - startTime);
    return result;
}
```

Hier erkennen wir sehr schön, dass die vom Proxy bereitzustellende Funktionalität sauber in einer Methode gekapselt ist. Sehr ähnlich arbeiten auch die dynamischen Proxys, die wir nun kennenlernen werden.

12.5.2 Dynamischer Proxy

Das JDK bietet im Package `java.lang.reflect` die Klasse `Proxy` zur Konstruktion von Proxys und das Interface `InvocationHandler`, um einen sogenannten dynamischen Proxy zu implementieren, der erst zur Laufzeit (daher dynamisch) anhand der Angabe einer Menge zu erfüllender Interfaces erzeugt wird.

Proxy-Funktionalität mit einem `InvocationHandler` realisieren

Beginnen wir mit dem Interface `InvocationHandler` mit folgender Methode:

```
Object invoke(Object proxy, Method method, Object[] args)  throws Throwable;
```

Die gewünschte Funktionalität des dynamischen Proxys wird durch eine Implementierung der Methode `invoke()` vorgegeben. Dabei ist der erste Parameter das Proxy-Objekt, über das der Aufruf erfolgt, und nicht das Zielobjekt – dazu gleich mehr. Der zweite Parameter ist das Methodenobjekt (vgl. Abschnitt 12.1 über Reflection) und der dritte beinhaltet die Werte der benötigten Aufrufparameter.

Schauen wir uns die Umsetzung einer Performance-Messung basierend auf einem `InvocationHandler` an:

```
public class PerformanceMeasureInvocationHandler implements InvocationHandler
{
    private final IService service;

    public PerformanceMeasureInvocationHandler(IService service)
    {
        this.service = service;
    }

    @Override
    public Object invoke(final Object proxy, final Method method,
                         final Object[] args) throws Throwable
    {
        final long startTime = System.nanoTime();

        Object result = null;
        try
        {
            // Achtung! Hier nicht versehentlich proxy übergeben
            result = method.invoke(service, args);
        }
        catch (InvocationTargetException ex)
        {
            throw ex.getTargetException();
        }

        printExecTime("calculateSomething", System.nanoTime() - startTime);
        return result;
    }

    private void printExecTime(final String methodName, final long duration)
    {
        // ...
    }
}
```

Normalerweise wird in `invoke()` die Methode des Zielobjekts aufgerufen – muss sie aber nicht. Um in der Realisierung des `InvocationHandler`s eine Methode des Zielobjekts aufrufen zu können, muss man Zugriff darauf besitzen. Daher benötigt die Implementierung des `InvocationHandler`s – wie im Listing – ein Attribut, das das Zielobjekt zwischenspeichert.

Eine etwas knifflige Sache beim Methodenaufruf besteht darin, bei einer auftretenden Exception passend zu reagieren. Laut Signatur von `invoke()` kann eine beliebige Ausnahme vom allgemeinsten Typ `Throwable` auftreten. Diese Flexibilität wird benötigt, weil in der Originalmethode auch eine Ausnahme des unspezifischen Fehlertyps `Throwable` ausgelöst werden könnte. Allerdings muss man Besonderheiten durch den Methodenaufruf per Reflection beachten:

- Allgemeine Probleme bei Reflection
- Exceptions durch die per Reflection aufgerufene Methode

Eine möglicherweise bei der Methodenausführung ausgelöste Exception wird über eine `InvocationTargetException` signalisiert. Die ursprüngliche Exception muss erst aus der gefangenen `InvocationTargetException` herausgelöst werden und kann dann weiter propagiert werden, wie es im Listing gezeigt ist.

Wir wissen zwar nun, wie wir die Funktionalität implementieren, aber wie wird diese aufgerufen? Die Methode `invoke()` des `InvocationHandler`s wird vom Proxy-Objekt aufgerufen, wenn ein Aufruf an eine Methode der realisierten Interfaces erfolgt. Was uns demnach noch fehlt, ist das Proxy-Objekt.

Achtung: Fallstrick

In der Implementierung von `InvocationHandler` beim Methodenaufruf kann es schnell passieren, dass man versehentlich das Proxy-Objekt beim Methodenaufruf angibt, etwa so:

```
final Object result = method.invoke(proxy, args);
```

Das führt zu einer Endlosaufrufkette, die schließlich mit einer Fehlermeldung auf der Konsole endet.

Erzeugung und Einsatz eines dynamischen Proxy-Objekts

Nach Implementierung dieser Grundlagen wollen wir einen dynamischen Proxy erstellen und in Aktion erleben. Dazu bietet die Klasse `Proxy` die Fabrikmethode `newProxyInstance()`, der sowohl der gewünschte `InvocationHandler` als auch die Menge der durch den Proxy zu realisierenden Interfaces übergeben werden müssen. Für unser initiales Beispiel würde man folgendermaßen einen dynamischen Proxy erzeugen:

```
public static void main(final String[] args)
{
    final IService service = createService();
    service.calculateSomething(42);
}

private static IService createService()
{
    final IService service = new Service();

    final InvocationHandler handler =
                            new PerformanceMeasureInvocationHandler(service);

    final Class<?>[] proxyInterfaces = { IService.class };
    return (IService) Proxy.newProxyInstance(Service.class.getClassLoader(),
                                             proxyInterfaces, handler);
}
```

Listing 12.13 *Ausführbar als* '**DYNAMICPROXYEXAMPLE**'

Die erzeugte Proxy-Klasse besitzt neben den Methoden aus den übergebenen Interfaces auch diejenigen der Klasse `Object`. All diese werden an den `InvocationHandler` delegiert, insbesondere `toString()`, `equals()` und `hashCode()`. Dabei folgt jeder Methodenaufruf an den so erzeugten Proxy dann der Aufrufhierarchie: Client → Proxy → `InvocationHandler` → Originalobjekt.

Das obige Programm DYNAMICPROXYEXAMPLE gibt in etwa Folgendes aus:

```
calculateSomething
Method call of 'calculateSomething' took: 3004 ms
```

Wir sehen, dass die Originalmethode aufgerufen wird und auch eine Zeitmessung erfolgt.

Motivation für dynamische Proxys

In diesem Abschnitt möchte ich nochmals motivieren, wieso Proxys und im Speziellen dynamische Proxys hilfreich sein können. Nehmen wir dazu eine beliebige Klasse an, die etwas Business-Funktionalität über ein Interface bereitstellt. Oft gibt es im Verlauf der Implementierung den Wunsch nach Erweiterung, etwa für Logging, Performance-Messung, Zugriffskontrolle usw. Dabei handelt es sich oft um sogenannte Querschnittsfunktionalität, die immer wieder ähnlich realisiert wird.

Man findet folgende drei Varianten:

1. **Eigene Realisierung** – Die Funktionalität wird in jeder Methode wieder neu ausprogrammiert. Das besitzt jedoch folgende Negativpunkte:
   - Einiges an Aufwand
   - Existenz von doppeltem oder mehrfachem (ähnlichem) Code
   - Gefahr von Inkonsistenzen und fehlender Durchgängigkeit
   - Fehlende Trennung von Nutzcode und Utility-Code

2. **Proxy / Decorator** – Die Funktionalität wird in einer spezifischen Klasse selbst realisiert – die originale Klasse muss nicht angepasst werden, sondern sie wird bei Bedarf aufgerufen. Das besitzt folgende Konsequenzen:
   - \+ Klare Trennung von Nutzcode und Utility-Code
   - \+ Orthogonale Kombinierbarkeit der Utility-Funktionalitäten
   - o Etwas Aufwand für den Proxy
3. **Dynamische Proxys** – Zur Laufzeit wird basierend auf einem Interface und einem die Funktionalität bereitstellenden `InvocationHandler` dynamisch ein Proxy erzeugt. Dynamische Proxys lassen sich somit wie folgt charakterisieren:
   - \+ Weniger Aufwand
   - \+ Führt zu Konsistenz und Verständlichkeit, da der Sourcecode nur einmal existiert
   - \+ Klare Trennung von Nutzcode und Utility-Code
   - \+ Orthogonale Kombinierbarkeit der Utility-Funktionalitäten
   - \- Minimal schlechtere Ausführungszeit, da diese mit Reflection erfolgt

Anhand von ein paar Abbildungen möchte ich das Ganze nochmals verdeutlichen. Die erste Abbildung zeigt die Situation, dass es keinen Proxy gibt. Wollte man Funktionalität hinzufügen, so müsste das in der Klasse `Target` erfolgen. Nutzt man dagegen einen Proxy, so implementiert dieser die Funktionalität und delegiert die Methodenaufrufe. Dadurch wird auch bereits der größte Nachteil direkt sichtbar: Der Proxy muss sämtliche Methoden implementieren und bei Erweiterungen oder Änderungen in der Schnittstelle vom `Target` müssen diese im Proxy nachgezogen werden.

Abbildung 12-4 *Einführung eines statischen und dynamischen Proxys*

Die letzte Abbildung zeigt die Abläufe bei einem dynamischen Proxy. Ein Client spricht diesen ganz normal über die Schnittstelle an. Dass dahinter ein `InvocationHandler` seine Arbeit verrichtet, ist transparent. Auch, dass der Proxy nur eine dünne Aufrufschicht darstellt, die per Reflection angesprochen wird, wird nicht veröffentlicht. Im `InvocationHandler` selbst kommt dann in der Regel wieder Reflection zum Einsatz, um die gewünschten Methoden aufzurufen. Demnach muss der `InvocationHandler` das `Target` kennen, wie wir es eingangs auch schon diskutiert haben. Der große Vorteil an dieser Variante ist, dass bei einer Änderung am Interface des Targets (oftmals) keine Anpassung im Proxy notwendig werden. Es ist auch nicht notwendig, eine Vielzahl an Methoden zu realisieren, sondern man konzentriert sich auf die Umsetzung der benötigten Funktionalität in Anlehnung an das bereits erwähnte Execute-Around-Pattern.

Proxy zum generischen Logging

Nehmen wir an, wir wollten auch ein einfaches Logging als Proxy realisieren. Dazu könnten wir folgenden `InvocationHandler` schreiben:

```
public class LoggingInvocationHandler implements InvocationHandler
{
    private Object target;

    public LoggingInvocationHandler(Object target)
    {
        this.target = target;
    }

    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
    {
        System.out.println("Invoking " + m.getName() + " " +
                           Arrays.toString(args));
        return m.invoke(target, args);
    }
}
```

Schauen wir uns nun noch an, wie man diesen `InvocationHandler` mit dem zuvor realisierten kombiniert:

```
private static IService createService()
{
    final IService origService = new Service();
    final InvocationHandler handler1 = new PerformanceMeasureInvocationHandler(
          origService);

    final Class<?>[] proxyInterfaces = { IService.class };
    final IService service = (IService) Proxy.newProxyInstance(Service.class.
          getClassLoader(), proxyInterfaces, handler1);

    final InvocationHandler handler2 = new LoggingInvocationHandler(service);
    return (IService) Proxy.newProxyInstance(Service.class.getClassLoader(),
          proxyInterfaces, handler2);
}
```

Listing 12.14 *Ausführbar als* '**DYNAMICPROXYCOMBINATIONEXAMPLE**'

Das Programm DYNAMICPROXYCOMBINATIONEXAMPLE produziert folgende Ausgaben, die die Hintereinanderausführung der Proxys zeigen:

```
Invoking calculateSomething [42]
calculateSomething
Method call of 'calculateSomething' took: 3001 ms
```

Fazit zu dynamischen Proxys

Schrittweise habe ich dynamische Proxys eingeführt und dabei verdeutlicht, dass sich damit bestehende Klassen ohne viel Mühe um Funktionalität erweitern lassen. Vor allem Querschnittsfunktionalitäten, wie Performance-Messung, Logging usw., sind ideale Kandidaten, die mit dynamischen Proxys realisiert werden können. Zudem lassen sich die Proxys einfach kombinieren. Überlegen Sie einmal, wie viel Sourcecode Sie schon für die hier gezeigten beiden Proxys hätten in Ihren Applikationsklassen schreiben müssen – besser noch, man kann die Funktionalität nahezu überall wiederverwenden.

12.6 HTTP/2-API

HTTP (Hypertext Transfer Protocol) ist der Standard im Internet zum Transport von Webinhalten, vor allem Webseiten. Bis zum Jahr 2015 galt immer noch der 1997 verabschiedete Standard HTTP 1.1. Mittlerweile nutzen diverse Webseiten das neue HTTP/2. Dieses basiert auf dem binären, von Google entwickelten SPDY-Protokoll (SPDY steht für speedy). Unter anderem durch diese binäre Datenübertragung statt der textorientierten bei HTTP 1.1 lassen sich deutliche Geschwindigkeitsverbesserungen erzielen.[9] Trotzdem bleibt die Kompatibilität zu HTTP 1.1 gewahrt.

12.6.1 Einführung

Bis einschließlich Java 10 war eine HTTP-Kommunikation nur mühselig und wenig intuitiv implementierbar, da nur eine Low-Level-Abstraktion existierte. Mit Java 11 wurde das HTTP/2-API in den Packages `java.net.http.*` ins JDK integriert. Dabei orientiert sich das Programmiermodell an den Bestandteilen einer HTTP-Kommunikation, also etwa Request und Response. Diese an der Fachlichkeit ausgerichtete Terminologie erleichtert das Verständnis. Erwähnenswert ist außerdem, dass die Verarbeitung des Bodys der Antwort mithilfe eines sogenannten `BodyHandler`s geschieht.

[9] Eine interessante Präsentation von der Devoxx 2015 finden Sie als YouTube-Video unter `https://www.youtube.com/watch?v=QpLtBftqM04`.

Einführendes Beispiel mit dem neuen HTTP/2-API

Ein Zugriff auf die Seite von Oracle und das Aufbereiten des Inhalts als String geschieht mit dem HTTP/2-API aus JDK 11 lesbar und verständlich wie folgt:

```
private static void readOraclePageJdk11() throws URISyntaxException,
                                                 IOException,
                                                 InterruptedException
{
    final URI uri = new URI("https://www.oracle.com/index.html");

    final HttpRequest request = HttpRequest.newBuilder(uri).GET().build();
    final BodyHandler<String> asString = HttpResponse.BodyHandlers.ofString();

    final HttpClient httpClient = HttpClient.newHttpClient();
    final HttpResponse<String> response = httpClient.send(request,
                                                          asString);

    printResponseInfo(response);
}

private static void printResponseInfo(final HttpResponse<String> response)
{
    System.out.println("Status:  " + response.statusCode());
    System.out.println("Body:    " + response.body());
    System.out.println("Headers: " + response.headers().map());
}
```

Ein `java.net.http.HttpRequest` wird durch Methoden wie `newBuilder(URI)`, `GET()` oder `POST()` erzeugt. Für die Verarbeitung der Antwortdaten ist dann eine Instanz eines `java.net.http.BodyHandler`s zuständig. Davon sind einige vordefiniert, die über die Fabrikmethoden der inneren Klasse `BodyHandlers` erzeugt werden, hier mit `ofString()`. Ausgangsbasis für eine HTTP-Kommunikation ist eine Instanz vom Typ `java.net.http.HttpClient`, die mit der Fabrikmethode `newHttpClient()` erzeugt wird. Die Kommunikation kann dann synchron durch einen Aufruf von `send()` gestartet werden – auf eine asynchrone Verarbeitung gehe ich weiter unten ein.

Die zurückgelieferte `java.net.http.HttpResponse` bietet vor allem die beiden Methoden `statusCode()` und `body()`. Die erste ermöglicht den Zugriff auf den HTTP-Statuscode und die zweite erlaubt es, den Inhalt der Webseite auszulesen. Zudem liefert die Methode `headers()` ein `java.net.http.HttpHeaders`-Objekt, das Informationen zu den Header-Werten der Antwort bereitstellt.

Alternative Datenformate Alternativ zu der im Beispiel genutzten Stringrepräsentation kann ein `BodyHandler` unter anderem mit den Methoden `ofLines()`, `ofByteArray()`, `ofFile()` und `ofFileDownload()` in der durch den Namen beschriebenen Ergebnisform erzeugt werden.

Webseite als Datei herunterladen

Mit dem gerade erwähnten `BodyHandler` namens `ofFile()` wollen wir uns noch etwas eingehender beschäftigen und die zuvor als String ausgelesene Oracle-Webseite nun als Datei herunterladen:

```
public static void main(final String[] args) throws Exception
{
    var uri = new URI("https://www.oracle.com/index.html");

    var request = HttpRequest.newBuilder(uri).GET().build();

    var downloadPath = Paths.get("oracle-index.html");
    var asFile = HttpResponse.BodyHandlers.ofFile(downloadPath);

    var httpClient = HttpClient.newHttpClient();
    var response = httpClient.send(request, asFile);

    if (response.statusCode() == 200)
    {
        System.out.println("Content written to file: " +
                           downloadPath.toAbsolutePath());
    }
}
```

Asynchrone Verarbeitung

Vor allem für länger dauernde Berechnungen oder I/O-Operationen ist es sinnvoll, diese asynchron auszuführen, um Blockierungen, wie sie durch eine synchrone Verarbeitung auftreten können, zu vermeiden. Mit dem neuen HTTP/2-API ist diese Anforderung kinderleicht umzusetzen. Praktischerweise stimmen dabei die meisten Schritte bei der asynchronen HTTP-Kommunikation mit denen der synchronen überein. Man muss lediglich statt der Methode `send()` die Methode `sendAsync()` aufrufen, um eine asynchrone Kommunikation zu erzielen.

Weil das Ergebnis eines asynchronen Aufrufs noch nicht direkt verfügbar ist, wird zur weiteren Verarbeitung ein `CompletableFuture<HttpResponse<T>>` zurückgeliefert. Mit diesem lässt sich auf die Fertigstellung warten und über `isDone()` abfragen, ob die Verarbeitung beendet wurde. Ist das der Fall, kann man dann über die Methode `get()` auf die im `CompletableFuture<HttpResponse<T>>` vorliegende `HttpResponse<T>` zugreifen. Die Parallelverarbeitung und das Warten auf die Fertigstellung sind im Listing durch eine Methode `waitForCompletion()` angedeutet. Sollen keine Aktionen für den Timeout erfolgen, so kann man die Verarbeitung auch einfacher durch Aufruf von `thenAccept(Consumer<? super T>)` ausdrücken – hier auskommentiert als alternative Variante dargestellt:

```
public static void main(final String[] args) throws Exception
{
    var uri = new URI("https://www.oracle.com/index.html");

    var request = HttpRequest.newBuilder(uri).GET().build();
    var asString = HttpResponse.BodyHandlers.ofString();
```

```
    var httpClient = HttpClient.newHttpClient();
    final CompletableFuture<HttpResponse<String>> asyncResponse = httpClient.
                                                  sendAsync(request, asString);

    // Variante: Verarbeitung mit eigenem Warten, Abfrage auf Erfolg und
    // bei nicht erfolgreichem Abschluss mit einem Abbruch per cancel(true)
    waitForCompletion(asyncResponse);
    if (asyncResponse.isDone())
    {
        final HttpResponse<String> response = asyncResponse.get();
        printResponseInfo(response);
    }
    else
    {
        asyncResponse.cancel(true);
        System.err.println("timeout");
    }
}
```

Zum Warten auf Fertigstellung kommt im obigen Listing folgende Hilfsmethode zum Einsatz:

```
static void waitForCompletion(final CompletableFuture<HttpResponse<String>>
                              asyncResponse) throws InterruptedException
{
    int i = 0;
    while (!asyncResponse.isDone() && i < 10)
    {
        System.out.println("Step " + i);
        i++;
        Thread.sleep(200);
    }
}
```

Alternativ schreibt man das Warten und die Verarbeitung mit den bereits bekannten Möglichkeiten von `CompletableFuture<T>` eleganter wie folgt:

```
// Variante: Verarbeitung, sobald die Response eintrifft
// Warten und Verarbeitung: Variante rein mit CompletableFuture
asyncResponse.thenAccept(response -> printResponseInfo(response)).
              orTimeout(2, TimeUnit.SECONDS).
              exceptionally(ex ->
              {
                  asyncResponse.cancel(true);
                  System.err.println("timeout");
                  return null;
              });
```

12.6.2 Real-World-Example: Wechselkurs mit REST

Zum Abschluss der HTTP/2-Betrachtung wollen wir Wechselkurse für Zeitabschnitte durch REST-Abfragen an `https://exchangeratesapi.io/` bereitstellen. Hierzu kann man das HTTP/2-API gewinnbringend nutzen. Die Abfrage kapseln wir in einer Methode `performGet()`. Um den Zeitabschnitt zu definieren, nutzt man die Neuerung `datesUntil()` aus JDK 9 und erzeugt damit einen `Stream<LocalDate>`. Das

geschieht hier durch die Angabe einer `Period` mit monatlicher Schrittweite. Mit einem passenden `TemporalAdjuster` lässt sich jeweils an das Monatsende springen. Zur Speicherung des Ergebnisses des REST-Aufrufs kommt die Neuerung `var` zum Einsatz und ersetzt die Nutzung eines konkreten Typs. Seit Java 9 bietet `Optional<T>` schließlich die Methode `ifPresentOrElse()`, um sowohl eine Aktion im Positiv- als auch Negativfall ausführen zu können. Das alles kombinieren wir wie folgt:

```
public static void main(String[] args) throws IOException, InterruptedException
{
    // HTTP/2-API + Date and Time API
    var httpClient =
         HttpClient.newBuilder().connectTimeout(Duration.ofSeconds(5)).build();

    // Date and Time API => Stream mit monatlicher Schrittweite
    LocalDate startDay = LocalDate.of(2020, Month.FEBRUARY, 1);
    LocalDate endDay = LocalDate.of(2020, Month.JUNE, 1);
    Period stepOneMonth = Period.ofMonths(1);

    // DATESUNTIL
    startDay.datesUntil(endDay, stepOneMonth).forEach(localDate -> {

        // TEMPORALADJUSTERS
        var endOfMonth = localDate.with(TemporalAdjusters.lastDayOfMonth());

        // var seit Java 10 und Aufruf mit HTTP/2-API
        var optResponseBody = performGet(httpClient, endOfMonth);

        Runnable noDataInfo =
                 () -> System.out.println("No data for " + endOfMonth);
        Consumer<String> dataNotification =
                         value -> System.out.println(endOfMonth +
                                                     " reported " + value);

        // OPTIONAL-Handling
        optResponseBody.ifPresentOrElse(dataNotification,
                                        noDataInfo);
    });
}

private static Optional<String> performGet(final HttpClient httpClient,
                                           final LocalDate desiredDate)
{
    try
    {
        URI url = URI.create("https://api.exchangeratesapi.io/" +
                             desiredDate + "?symbols=CHF");

        HttpRequest httpRequest =
                    HttpRequest.newBuilder().GET().uri(url).build();

        var response = httpClient.send(httpRequest, BodyHandlers.ofString());
        String responseBody = response.body();

        return Optional.of(responseBody);
    }
    catch (Exception ex)
    {
        // in diesem Beispiel ausnahmsweise ignorieren
    }
    return Optional.empty();
}
```

Führt man das obige Programm aus, so werden einem die Wechselkurse von Schweizer Franken zu Euro von Februar 2020 bis Juni 2020 ausgegeben. Das sieht etwa wie folgt aus:

```
2020-02-29 reported {"rates":{"CHF":1.0614},"base":"EUR","date":"2020-02-28"}
2020-03-31 reported {"rates":{"CHF":1.0585},"base":"EUR","date":"2020-03-31"}
2020-04-30 reported {"rates":{"CHF":1.0558},"base":"EUR","date":"2020-04-30"}
2020-05-31 reported {"rates":{"CHF":1.072},"base":"EUR","date":"2020-05-29"}
```

Fazit

Dieses in Bezug auf die Länge des Sourcecodes sehr überschaubare Beispiel verdeutlicht hervorragend, wie sich die Bestandteile des JDKs als Bausteine gewinnbringend zusammenschalten lassen. Hier kombinieren wir die APIs zu HTTP/2, `Optional<T>` sowie dem Stream- und dem Date and Time-API und nutzen ganz nebenbei noch `var` zur verkürzten Definition lokaler Variablen.

12.6.3 Fazit

Die Unterstützung von HTTP/2 ist eine sinnvolle Ergänzung, die zudem eine Rückwärtskompatibilität zu HTTP 1.1 bietet. Nebenbei erhält man Geschwindigkeitssteigerungen von bis zu 50 % sowie ein verständliches API. Auch die Möglichkeit, asynchrone Aufrufe auszuführen, ist hervorzuheben, da sich diese durch den Einsatz von `CompletableFuture<T>` in der Programmierung recht einfach gestalten.

12.7 Weiterführende Literatur

Die in diesem Kapitel behandelten Themen sind komplex und umfangreich. Daher konnte hier nur ein kurzer Einblick gegeben werden. Weiterführende Informationen finden Sie in folgenden Büchern:

- »**Java Reflection in Action**« von Ira R. und Nate Forman [19]
  Dies ist eines der wenigen Bücher, die das Thema Reflection mit größerem Tiefgang behandeln.
- »**Core Java – Band 2 Expertenwissen**« von Cay S. Horstmann und Gary Cornell [33]
  In diesem Buch liefern Cay S. Horstmann und Gary Cornell einen umfassenden Überblick über fortgeschrittene Java-Themen. Insbesondere Annotations werden dort ausführlicher als in diesem Kapitel dargestellt.
- »**Java 6 Core Techniken**« von Friedrich Esser [17]
  In diesem Buch taucht Friedrich Esser tief in fortgeschrittene Java-Themen ein. Insbesondere werden Generics und Reflection sowie die Verarbeitung von Annotations ausführlich behandelt.

13 Basiswissen Internationalisierung

Soll eine Anwendung in verschiedenen Ländern eingesetzt werden, so sind lokale Besonderheiten, wie verschiedene Darstellungen von Währungen, Uhrzeiten und Datumswerten, aber auch die Übersetzung und Anpassung der in der grafischen Oberfläche dargestellten Texte und Informationen, wichtige Themen. Gerade Letzteres ist für Benutzer ein offensichtlicher Aspekt der Internationalisierung. Bei der Realisierung spielen diverse weitere Dinge eine wichtige steuernde Rolle, etwa die (im Betriebssystem) gewählte Sprache und das Land. Die folgenden Abschnitte behandeln verschiedene Themen der Internationalisierung so weit, dass eine Umsetzung in eigenen Programmen möglich wird.

Abschnitt 13.1 gibt eine Einführung in das Thema Internationalisierung und stellt insbesondere die formatierte, landesspezifische Ein- und Ausgabe vor, etwa Darstellungen von Uhrzeit, Datum und Währungen. Die beschriebenen Funktionalitäten werden in Abschnitt 13.2 zu Programmbausteinen zur Internationalisierung ausgebaut.

13.1 Internationalisierung im Überblick

Nachfolgend wird Basiswissen aufgebaut, das wir für die Erstellung wiederverwendbarer Bausteine in unseren Applikationen benötigen. Abschnitt 13.1.1 beschreibt zunächst einige Grundlagen. Dann wird in Abschnitt 13.1.2 die Klasse `java.util.Locale` vorgestellt, die die Basis zur Verwaltung landes- oder sprachabhängiger Besonderheiten bildet. Für die Bereitstellung von textuellen Informationen ist die Klasse `java.util.PropertyResourceBundle` hilfreich. Sie wird in Abschnitt 13.1.3 beschrieben. Danach gibt Abschnitt 13.1.4 einen kurzen Überblick über die `Format`-Klassen aus dem Package `java.text`. In den darauffolgenden Abschnitten werden dann einige Spezialisierungen genauer vorgestellt. Im Einzelnen sind dies die Klassen `DateFormat`, `NumberFormat` sowie `MessageFormat` in den Abschnitten 13.1.5 bis 13.1.7. Im Anschluss daran stelle ich in Abschnitt 13.1.8 die Klasse `Collator` vor, die es erlaubt, beim Vergleich von Strings landes- oder sprachspezifische Besonderheiten zu beachten.

13.1.1 Grundlagen und Normen

Eine sprachabhängige Benutzeroberfläche fällt direkt bei der Benutzung eines Programms auf. Für eine gelungene Internationalisierung ist die Darstellung übersetzter Texte jedoch nur ein erster Schritt. Meistens sind weitere regionale Unterschiede in der Darstellung zu berücksichtigen, damit sich ein Benutzer wirklich heimisch fühlt. Am Beispiel der Schweiz wird das besonders deutlich. Eine Internationalisierung lediglich über die Angabe des Landes ist dort unmöglich, da zumindest vier Sprachen gesprochen werden – alle mit ihren lokalen Besonderheiten.

Um in einer Anwendung verschiedene Übersetzungen anbieten zu können, ist es erforderlich, Sprachen und Regionen beschreiben und unterscheiden zu können. Betrachten wir im Folgenden einige Details dazu.

Sprache

Sprachen lassen sich mithilfe der Norm ISO 639 differenzieren. Dort werden Kennungen für Sprachen in Form von zwei Kleinbuchstaben definiert, etwa `de` für Deutsch, `en` für Englisch, `fr` für Französisch, `it` für Italienisch usw.

Land und Region

Zur Codierung von Ländern wird die Norm ISO 3166 verwendet. Diese beschreibt Länder über zwei Großbuchstaben. Einige Beispiele gebräuchlicher Kennungen sind: `CH` für die Schweiz, `DE` für Deutschland, `GB` für Großbritannien, `FR` für Frankreich, `IT` für Italien usw.

Zahlen

Die Formatierung von Zahlen ist regional unterschiedlich. Am auffälligsten sind die verschiedenen Dezimaltrennzeichen (oder Trennzeichen für Nachkommastellen). Je nach Region wird ein Punkt oder ein Komma verwendet. Auch das Tausendertrennzeichen kann abweichen. In der Schweiz ist etwa ein Apostroph statt eines Punkts gebräuchlich.

Währungen

Die Darstellung von Währungen variiert von Region zu Region. Dabei ist sowohl die Position als auch das Währungskürzel selbst verschieden:

```
Deutschland:           12.345,67 €
USA:                   $12,345.67
Italien:               € 12.345,67
Italienische Schweiz:  SFr. 12'345.67
```

Wie man sieht, sind die Zahlenwerte außerdem unterschiedlich formatiert.

Datum und Uhrzeit

Es existieren weltweit einige Varianten bei der Darstellung von Datum und Uhrzeit. Eine Datumsangabe 01/03/05 kann man als 1. März 2005 (Tag / Monat / Jahr), als 3. Januar 2005 (Monat / Tag / Jahr) oder aber als 5. März 2001 (Jahr / Monat / Tag) deuten. Um zumindest bei der Jahreszahl Verwechslungen zu vermeiden, ist die Angabe einer vierstelligen Jahreszahl ratsam.

Die Norm ISO 8601 definiert Formate für numerische Datums- und Zeitangaben und schlägt folgende Vereinheitlichung vor:

```
Datum:                  2010-08-26 (YYYY-MM-DD)
Uhrzeit:                20:08:16   (hh:mm:ss)
```

In der Regel wird eine Applikation eher den regionalen Besonderheiten und nicht der Norm folgen:

```
Deutschland:            26.08.2010 (DD.MM.YYYY)
USA:                    08/26/2010 (MM/DD/YYYY)
```

Achtung: In Java werden in der Klasse `DateFormat` von den gezeigten Kürzeln abweichende Kürzel verwendet! Tatsächlich führt etwa die Verwendung des Großbuchstabens `Y` zur Verarbeitung von Kalenderwochen bei der Jahresangabe und verändert das Ergebnis des Parsings massiv.

13.1.2 Die Klasse `Locale`

Die Klasse `java.util.Locale` hilft bei der Beschreibung landes- oder sprachabhängiger Besonderheiten: Es wird unter anderem spezifiziert, welches Format ein Datum besitzt. Außerdem wird die Formatierung von Zahlen festgelegt. Jedes `Locale`-Objekt repräsentiert eine bestimmte Region (geografisch, kulturell usw.) inklusive der dort gesprochenen Sprache. Für einige gebräuchliche `Locale`-Objekte existieren vordefinierte Konstanten. Weitere `Locale`-Objekte lassen sich durch einen Konstruktoraufruf mit der Angabe von Sprache und Land erzeugen:

```
public Locale(String language, String country)
```

Die Angaben folgen den Normen ISO 639 und ISO 3166 und sind relativ intuitiv, wie dies folgendes Listing zeigt:

```
public final class LocaleExample
{
    public static final Locale LOCALE_US             = Locale.US;

    public static final Locale LOCALE_FRENCH_SWISS   = new Locale("fr", "CH");
    public static final Locale LOCALE_ITALIAN_SWISS  = new Locale("it", "CH");

    public static final Locale LOCALE_SPANISH        = new Locale("es");
    public static final Locale LOCALE_SPANISH_MEXICO = new Locale("es", "MX");
    // ...
```

Die Angabe des Landes an den `Locale`-Konstruktor ist optional. Wird hier ein Leerstring übergeben oder der überladene Konstruktor mit nur einem Parameter für die Sprache aufgerufen, so repräsentiert das derart erzeugte `Locale`-Objekt lediglich eine Sprache *ohne* Landesinformation. Das sehen wir im Beispiel für die Konstante `LOCALE_SPANISH`.

Einstellungen zu Land und Sprache kann man über folgende Methoden der Klasse `Locale` ermitteln:

- `getLanguage()` – Liefert das Sprachkürzel, beispielsweise `de`, `fr`, `en` usw.
- `getCountry()` – Ermittelt das Land. Der Rückgabewert kann leer sein, wenn es sich um eine reine »Sprach«-`Locale` handelt.

Neben den Kürzeln sind oftmals die tatsächlichen Namen der Länder und die der dort gesprochenen Sprachen interessant. Eine menschenlesbare, textuelle Repräsentation liefern die zwei Methoden `getDisplayCountry()` und `getDisplayLanguage()`, für die es jeweils eine überladene Variante mit einem `Locale`-Objekt als Parameter gibt. Die parameterlose Variante stellt eine textuelle Repräsentation für die momentan in der JVM gewählte Default-Locale bereit – was in Deutschland meistens der vordefinierten `Locale.GERMANY` entspricht. Durch Aufruf der überladenen Methoden kann eine lesbare Darstellung in einer beliebigen anderen Sprache erfolgen, die durch ein übergebenes `Locale`-Objekt spezifiziert wird. Verdeutlichen wir uns dies anhand eines Beispiels. Nehmen wir folgende Aufrufe an:

```
public static void main(final String[] args)
{
    System.out.println(LOCALE_US.getDisplayCountry());
    System.out.println(LOCALE_US.getDisplayCountry(LOCALE_ITALIAN_SWISS));
    System.out.println(LOCALE_US.getDisplayLanguage());
    System.out.println(LOCALE_US.getDisplayLanguage(LOCALE_ITALIAN_SWISS));
}
```

Listing 13.1 *Ausführbar als* **'LOCALEEXAMPLE'**

Führt man das Programm LOCALEEXAMPLE aus, so werden Informationen zu Land und Sprache zu den zuvor definierten `Locale`-Objekten auf der Konsole ausgegeben:

```
Vereinigte Staaten von Amerika
Stati Uniti
Englisch
inglese
```

Zugriff auf alle verfügbaren `Locale`-Objekte

Alle verfügbaren `Locale`-Objekte kann man durch Aufruf der statischen Methode `getAvailableLocales()` der Klasse `Locale` ermitteln. Beachten Sie bitte, dass das zurückgelieferte Array von `Locale`-Objekten unsortiert ist!

Einsatz der Klasse `Locale`

Zur Demonstration der bislang vorgestellten Methoden schreiben wir ein Beispielprogramm, das alle verfügbaren `Locale`-Objekte in einer Tabelle auflistet und einige Informationen sprachabhängig gemäß der Auswahl einer beliebigen Sprache darstellt. Dazu sind zunächst zwei Funktionalitäten zu entwickeln:

1. Die Liste der dargestellten Informationen soll alphabetisch sortiert werden. Von Hause aus ist die Klasse `Locale` nicht sortierbar, da sie das Interface `Comparable<T>` nicht erfüllt und damit auch keine natürliche Ordnung definiert. Zur Realisierung eines Sortierkriteriums muss man daher das Interface `Comparator<T>` nutzen (vgl. Abschnitt 6.1.8).
2. Zur Darstellung der Sprachinformationen sollen aus der Gesamtmenge der verfügbaren `Locale`-Objekte all diejenigen herausgefiltert werden, die lediglich Sprachinformationen anbieten.

Sortierung Im praktischen Einsatz wäre eine Sortiermöglichkeit wünschenswert, die jedoch standardmäßig für die Klasse `Locale` nicht existiert. Abhilfe bietet hier die Implementierung von `Comparator<Locale>`: Die bereits bekannten Methoden `getLanguage()` und `getCountry()` nutzen wir zur Definition eines Sortierkriteriums gemäß Sprache und Land. Bei der Implementierung profitieren wir von den in Java 8 eingeführten Neuerungen im Interface `Comparator<T>` (vgl. Abschnitt 6.2.4). Damit definieren wir folgende Komparatoren innerhalb einer hier nicht gezeigten Utility-Klasse `LocaleUtils`:

```
public static final Comparator<Locale> byLanguage =
                                       Comparator.comparing(Locale::getLanguage);
public static final Comparator<Locale> byCountry =
                                        Comparator.comparing(Locale::getCountry);

public static final Comparator<Locale> LOCALE_COMPARATOR =
                                        byLanguage.thenComparing(byCountry);
```

Filterung Nehmen wir an, es sollen `Locale`-Objekte in einer Combobox auswählbar sein, wobei nur die `Locale`-Objekte angezeigt werden dürfen, die reine Sprach-Locales darstellen, d. h., deren Land unbesetzt ist. In diesem Fall verwenden wir eine recht einfache Umsetzung. Auch hierbei greifen wir mit dem Stream-API (vgl. Abschnitt 7.1) auf eine fundamentale Neuerung aus Java 8 zurück. Folgendes Listing zeigt die Methode `getLanguageOnlyLocales()`, die ebenfalls in der Utility-Klasse `LocaleUtils` realisiert ist:

```
public static final List<Locale> getSortedLocales()
{
    final Locale[] availableLocale = Locale.getAvailableLocales();

    return Arrays.stream(availableLocale).
                  sorted(LOCALE_COMPARATOR).
                  collect(Collectors.toList());
}

public static List<Locale> getLanguageOnlyLocales()
{
    final List<Locale> sortedLocales = getSortedLocales();

    final Predicate<Locale> isLanguageOnly =
                            locale -> locale.getCountry().isEmpty();

    return sortedLocales.stream().
                         filter(isLanguageOnly).collect(Collectors.toList());
}
```

Die Beispielapplikation Wir kombinieren die vorgestellten Bausteine zu einem Beispielprogramm, das über eine Combobox eine Sprachauswahl ermöglicht und verschiedene Informationen zu `Locale`-Objekten in einer Tabelle anzeigt. Abbildung 13-1 zeigt einen Screenshot.

Abbildung 13-1 *Beispielapplikation* AVAILABLELOCALESEXAMPLE

Zentral im Fenster wird eine Tabelle dargestellt, die alle verfügbaren `Locale`-Objekte auflistet. Die einzelnen Spalten werden durch die uns bereits bekannten Methoden, unter anderem `getDisplayLanguage()`, mit Werten versorgt. Im oberen Teil des Fensters bietet eine Combobox Zugriff auf alle vordefinierten Sprach-Locales. Eine Auswahl aktiviert die gewünschte Locale. Dadurch werden in der Tabelle entsprechende Locale-spezifische Informationen in den hinteren beiden Spalten angezeigt. Die im Beispiel gewählte Locale ist Thai. Daher kommt es zu den exotischen Schriftzeichen in den beiden letzten Spalten. Starten Sie das Programm AVAILABLELOCALESEXAMPLE, um eigene Experimente zu beginnen.

13.1.3 Die Klasse `PropertyResourceBundle`

Sollen in einer Applikation Texte entsprechend einer gewählten Sprache und Region angezeigt werden, sind dazu einige Vorarbeiten durchzuführen. Wünschenswert ist eine Trennung von dargestellten Informationen und dem nutzenden Sourcecode. Eine Repräsentation von Texten der Benutzeroberfläche als Magic Strings schließt sich daher von selbst aus. Statt die benötigten Übersetzungen direkt im Programm zu hinterlegen, ist es sinnvoll, im Sourcecode lediglich einige Konstanten zu definieren, die eindeutige Schlüssel zur Identifikation von Texten und Informationen darstellen. Die eigentlichen Übersetzungen sind in einer Datei[1] abgelegt und werden über die Schlüssel abgerufen.

Definition verschiedener Sprachdateien

Die Klasse `java.util.PropertyResourceBundle` erweitert die abstrakte Klasse `java.util.ResourceBundle` und nutzt die Klasse `Properties` zum Einlesen von Schlüssel-Wert-Paaren aus Dateien (vgl. Abschnitt 9.5.2). Betrachten wir zunächst mögliche Eingaben, bevor wir die Verarbeitung detaillierter kennenlernen. Deutsche Texte könnten wie folgt definiert sein:

```
txt_file=Datei
txt_new=Neu...
txt_open=Öffnen...
txt_properties=Einstellungen...
txt_quit=Beenden
```

Das englische Gegenstück dazu ist:

```
txt_file=File
txt_new=New
txt_open=Open...
txt_properties=Properties...
txt_quit=Quit
```

Einlesen der Texte

Das Einlesen der passenden Sprachdateien erledigt die statische Methode `getBundle(String, Locale)` der Klasse `PropertyResourceBundle`. Der Methode wird lediglich der Pfad und das Präfix der gewünschten Sprachdatei sowie ein `Locale`-Objekt übergeben:[2]

```
final String BUNDLE_BASENAME = "ch13_i18n.PDFEditor";

final ResourceBundle germanBundle = PropertyResourceBundle.getBundle(
                                        BUNDLE_BASENAME, Locale.GERMANY);
final ResourceBundle englishBundle = PropertyResourceBundle.getBundle(
                                        BUNDLE_BASENAME, Locale.UK);
```

[1] Jeweils eine für jede zu unterstützende Sprache und Region.

[2] Voraussetzung ist jedoch, dass die Property-Dateien im `CLASSPATH` liegen.

Die in der Locale genutzten Sprach- und Landesinformationen ergeben einen Dateinamen im Format 'Name_<Sprachkürzel>_<Länderkürzel>.properties' und die dadurch referenzierte Property-Datei wird eingelesen.[3] Praktischerweise lassen sich verschiedene Sprachdateien allein durch Variation des übergebenen `Locale`-Objekts ansprechen. Zur Unterstützung mehrerer Sprachen muss lediglich für jede gewünschte Sprache eine Property-Datei existieren, wie dies zuvor für eine deutsche und englische Locale gezeigt wurde.

Zugriff auf die Texte

Der Zugriff auf die so hinterlegten Texte erfolgt über spezielle Zugriffsschlüssel, die in einer Konstantenklasse oder besser einer `enum`-Aufzählung wie folgt definiert werden:

```
public enum ResourceKeys
{
    txt_file, txt_new, txt_open, txt_properties, txt_close, txt_save, txt_quit
}
```

Mithilfe solcher Zugriffsschlüssel lassen sich die jeweils zugehörigen sprachspezifischen Texte in einem `PropertyResourceBundle` suchen. Wir definieren diese Funktionalität als Methode `getLangString(ResourceBundle, ResourceKeys)` in der folgenden Utility-Klasse `ResourceBundleUtils`:

```
public final class ResourceBundleUtils
{
    private static final String INDICATOR_MISSING_RESOURCE = "?";
    private static final String INDICATOR_MISSING_KEY = "??";

    public static String getLangString(final ResourceBundle resourceBundle,
                                       final ResourceKeys key)
    {
        if (resourceBundle != null)
        {
            try
            {
                return resourceBundle.getString(key.name());
            }
            catch (final MissingResourceException e)
            {
                return INDICATOR_MISSING_KEY + key;
            }
        }

        return INDICATOR_MISSING_RESOURCE + key;
    }
    // ...
```

[3] Nur wenn man sich an diese Konvention bei der Benennung der Property-Dateien hält, ist ein zuverlässiges Einlesen durch die Methode `getBundle(String, Locale)` möglich. Weitere Informationen zur Arbeitsweise finden Sie in der API-Dokumentation zu dieser Methode.

Beispiel

Die folgende `main()`-Methode verwendet die zuvor vorgestellten Methoden und liest zwei verschiedene Sprachdateien ein. Anschließend werden die Übersetzungen für die Konstanten `txt_file` und `txt_open` auf Deutsch und Englisch ausgegeben:

```
public static void main(final String[] args) throws MissingResourceException
{
   final String BUNDLE_BASENAME = "ch13_i18n.PDFEditor";

   showTexts(PropertyResourceBundle.getBundle(BUNDLE_BASENAME, Locale.GERMANY));
   showTexts(PropertyResourceBundle.getBundle(BUNDLE_BASENAME, Locale.UK));
}

private static void showTexts(final ResourceBundle resourceBundle)
{
   final String txt_file = ResourceBundleUtils.getLangString(resourceBundle,
                                                             ResourceKeys.txt_file);
   final String txt_open = ResourceBundleUtils.getLangString(resourceBundle,
                                                             ResourceKeys.txt_open);

   System.out.println("txt_file='" + txt_file +  "' / " +
                      "txt_open='" + txt_open + "'");
}
```

Listing 13.2 *Ausführbar als* '**PROPERTYRESOURCEBUNDLEEXAMPLE**'

Beim Start des Programms PROPERTYRESOURCEBUNDLEEXAMPLE kommt es zu folgender Ausgabe auf der Konsole:

```
txt_file='Datei' / txt_open='Öffnen...'
txt_file='File' / txt_open='Open...'
```

Ausblick

Diese Einführung hat einen Anwendungsfall nicht betrachtet: In der Regel sind nicht nur einzelne Begriffe, sondern Kombinationen von Wörtern oder ganze Sätze zu lokalisieren. ***Aufgrund unterschiedlicher Grammatikregeln verschiedener Sprachen kann man ganze Sätze nicht durch einfaches Verketten von übersetzten Teilstrings zusammenfügen***. Stattdessen nutzt man die Möglichkeit, ganze Sätze in Property-Dateien abzulegen und an einigen Stellen Platzhalter einzufügen. Wie man dann mithilfe der Klasse `MessageFormat` auf diese Weise ganze Sätze sprachabhängig gestalten kann, zeigt Abschnitt 13.1.7.

13.1.4 Formatierte Ein- und Ausgabe

In Abschnitt 5.3.3 haben wir zur Formatierung von Zahlen, Datumsangaben usw. die Methode `format(String, Object...)` der Klasse `String` sowie die Methoden `format(String, Object...)` und `printf(String, Object...)` der Klasse `PrintStream` kennengelernt. Eine flexiblere und objektorientierte Lösung ist durch den Einsatz der im Folgenden vorgestellten `Format`-Klassen aus dem Package `java.text` möglich.

Überblick über die `Format`-Klassen

Zum Teil sind die Ausgaben von Programmen individuell zu formatieren oder auf die Bedürfnisse verschiedener Nationalitäten anzupassen. Die abstrakte Basisklasse `Format` und ihre Subklassen `DateFormat`, `NumberFormat` und `MessageFormat` stellen die dazu notwendige Funktionalität bereit. Die Klasse `MessageFormat` kann zur Anpassung von Ausgaben weitere `Format`-Klassen verwenden. Abbildung 13-2 zeigt die Klassenhierarchie.

Abbildung 13-2 *Klassenhierarchie der* `Format`*-Klassen*

Durch Einsatz der `Format`-Klassen lassen sich nicht nur landes- bzw. sprachabhängige Ausgaben realisieren, sondern es können auch Zeichenketten wieder in Typen wie `Number` oder `Date` zurückgewandelt werden. Die Basisklasse `Format` bietet dazu folgende zwei Methoden:

```
public String format(Object obj)
public Object parseObject(String source) throws ParseException
```

Die Methode `format(Object)` konvertiert übergebene Objekte in einen String. Durch Aufruf der Methode `parseObject(String)` wird versucht, einen String in ein spezielles Objekt zu konvertieren. Die Umwandlung beliebiger Strings mit Datums- und Zeitinformationen kann jedoch fehlschlagen, wenn die Eingabe nicht dem erwarteten sprachabhängigen Format entspricht. In solchen Fällen löst die Methode `parseObject(String)` eine `java.text.ParseException` aus.

Achtung: API-Merkwürdigkeit der `Format`-Klassen

In den Subklassen der Basisklasse `Format` wird zusätzlich eine öffentliche Methode `parse(String)` angeboten. Diese wiederum wird durch die Methode `parseObject(String)` aufgerufen. In Applikationen spielt letztgenannte Methode kaum eine Rolle: Stattdessen wird meistens direkt mit den `parse(String)`-Methoden der Subklassen gearbeitet.

Achtung: `format()`-Inkompatibilität mit den `Format`-Klassen

Weder die Methode `format(String, Object...)` der Klasse `String` noch die Methoden `format(String, Object...)` und `printf(String, Object...)` der Klasse `PrintStream` verwenden `Format`-Objekte, sondern nutzen die Klasse `Formatter`. Diese bietet eine weitgehende Kompatibilität zu `printf()`-Ausgaben in C/C++. Die eingesetzten Formatzeichen sind dabei jedoch unterschiedlich zu denen der `Format`-Klassen.

13.1.5 Datumswerte und die Klasse `DateFormat`

Die Basisklasse `DateFormat` und ihre konkrete Subklasse `SimpleDateFormat` erleichtern die formatierte Ausgabe von Datumswerten. Beide Klassen ermöglichen zudem die Konvertierung von Strings, die Datums- und/oder Zeitinformationen enthalten, in ein Objekt vom Typ `Date`. Bedenken Sie bitte, dass bei Neuentwicklungen möglichst das mit Java 8 eingeführte Date and Time API (vgl. Kapitel 8) zum Einsatz kommen sollte, das auch die Formatierung und das Parsing erleichtert. Deswegen erfolgt hier lediglich ein Kurzüberblick.

Formatierte Ausgabe

Zur Ausgabe von Datumswerten stellen folgende Fabrikmethoden verschiedene Arten von Datumsformaten bereit:

- **`getInstance()`** – Formatierung von Datums- und Zeitangaben
- **`getTimeInstance()`** – Formatierung von Zeitangaben
- **`getDateInstance()`** – Formatierung von Datumsangaben
- **`getDateTimeInstance()`** – Formatierung von Datums- und Zeitangaben

Einsatz von `SimpleDateFormat` für Anpassungen

Die mit den Fabrikmethoden erzeugten `DateFormat`-Instanzen sind für einige Einsatzgebiete bereits ausreichend. Allerdings gibt es immer wieder Situationen, in denen man ein Datumsformat nach eigenen Bedürfnissen gestalten möchte oder muss. Mithilfe der

Klasse `SimpleDateFormat` wird es möglich, eigene Datumsformate zu definieren, deren Beschreibung durch Angabe eines speziellen Musterstrings erfolgt. Die wichtigsten Formatzeichen sind im Muster '`dd. MMMM yyyy HH:mm:ss`' enthalten. Dieses führt beispielsweise zu folgender Ausgabe von Datum und Uhrzeit:

```
20. Dezember 2016 15:54:54
```

13.1.6 Zahlen und die Klasse `NumberFormat`

Bei der Verarbeitung von Zahlen, Prozentangaben und Währungen hilft die Klasse `NumberFormat`.

Formatierte Ausgabe

Zur Formatierung von Ausgaben können über verschiedene Fabrikmethoden Instanzen der Klasse `NumberFormat` erzeugt werden:

- **`getInstance()`** und **`getNumberInstance()`** – Beide stellen ein Format für Gleitkommazahlen bereit, das standardmäßig auf drei Stellen nach dem Komma rundet. Mit der Methode `setMaximumFractionDigits(int)` kann die Anzahl an Nachkommastellen eingestellt bzw. über `getMaximumFractionDigits()` abgefragt werden. Analog dazu existieren Methoden, um die minimale Anzahl an Nachkommastellen festzulegen bzw. zu ermitteln.
- **`getIntegerInstance()`** – Erzeugt ein auf Ganzzahlen ausgerichtetes Format, das den Nachkommateil abschneidet und die Zahl vorher rundet.
- **`getCurrencyInstance()`** – Erzeugt ein Format für Währungen, das auf zwei Nachkommastellen rundet.
- **`getPercentInstance()`** – Dient zum Darstellen von Prozentwerten. Dazu wird die übergebene Zahl auf zwei Stellen nach dem Komma gerundet und dann mit 100 multipliziert: Beispielsweise wird die Zahl 0.7271 als 73 % dargestellt.

Um zusätzlich zu den voreingestellten Landesspezifika andere Formate bei der Ausgabe zu berücksichtigen, existieren jeweils überladene Methoden mit einem `Locale`-Objekt als Parameter. Das erinnert an die Realisierungen der Methoden `getDisplayLanguage()` und `getDisplayCountry()` in der Klasse `Locale`.

Achtung: Zweifelhaftes OO-Design im JDK

Einige Fabrikmethoden der Klasse `NumberFormat` erzeugen Objekte vom spezielleren Typ `DecimalFormat`. Es ist ein fragwürdiges OO-Design, wenn eine Basisklasse ihre Subklassen kennt und diese sogar erzeugt. Korrekterweise hätten die Fabrikmethoden in der Klasse in `DecimalFormat` definiert werden sollen.

Beispiel

Das folgende Beispielprogramm NUMBERFORMATEXAMPLE nutzt die vorgestellten Fabrikmethoden, um den Einsatz verschiedener Formate zur Ausgabe von Zahlen und Währungen zu zeigen. Dabei werden etwa lokale Unterschiede für Währungsausgaben am Beispiel von Italien und der italienischen Schweiz demonstriert. Auch außereuropäische Besonderheiten werden mit den Locales für die USA und Hongkong gezeigt:

```
public static void main(final String[] args)
{
    final double value = 12345.67890;

    NumberFormat numFormat = NumberFormat.getInstance(Locale.GERMANY);
    System.out.println("getInstance de DE\t\t" + numFormat.format(value));

    numFormat = NumberFormat.getNumberInstance(Locale.US);
    System.out.println("getNumberInstance en US\t\t" + numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(Locale.GERMANY);
    System.out.println("getCurrencyInstance de DE\t" + numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(Locale.US);
    System.out.println("getCurrencyInstance en US\t" + numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(Locale.ITALY);
    System.out.println("getCurrencyInstance it IT\t" + numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(LocaleExample.
          LOCALE_ITALIAN_SWISS);
    System.out.println("getCurrencyInstance it CH\t" + numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(new Locale("zh", "HK"));
    System.out.println("getCurrencyInstance zh HK\t" + numFormat.format(value));

    // Zahlenformat stammt aus Sprache, Währungszeichen sprachabhängig aus Land
    numFormat = NumberFormat.getCurrencyInstance(new Locale("de", "HK"));
    System.out.println("getCurrencyInstance de HK\t" + numFormat.format(value));
}
```

Listing 13.3 *Ausführbar als* **'NUMBERFORMATEXAMPLE'**

Dieses Beispiel verdeutlicht, dass die lokalen Besonderheiten (Währungsinformationen, Zahlenformate etc.) vom Benutzer nicht angegeben werden müssen. Es reicht vielmehr aus, ein `Locale`-Objekt an die entsprechenden Methoden der `Format`-Klassen zu übergeben. Für den Wert `12345.67890` kommt es zu folgenden Ausgaben:

```
getInstance de DE           12.345,679
getNumberInstance en US     12,345.679
getCurrencyInstance de DE   12.345,68 €
getCurrencyInstance en US   $12,345.68
getCurrencyInstance it IT   € 12.345,68
getCurrencyInstance it CH   SFr. 12'345.68
getCurrencyInstance zh HK   HK$12,345.68
getCurrencyInstance de HK   HKD 12.345,68
```

Man erkennt diverse Unterschiede in der Formatierung der Ausgabe. Eine Besonderheit wird bei der (künstlichen) deutschsprachigen Locale für Hongkong sichtbar: Die

Sprache bestimmt das Format der Zahlen. Das Land bestimmt das bzw. die Währungszeichen, wobei diese sprachabhängig sind – im Beispiel also etwa `HK$` bzw. `HKD` für den Hongkong-Dollar.

All diese sprach- und landesabhängigen Spezialitäten von Hand zu realisieren, wäre sehr aufwendig und fehleranfällig. Insbesondere gilt das auch für das Parsing derartiger Informationen. Auch dabei unterstützt die Klasse `NumberFormat`.

Parsen von Strings – Umwandlung in `Number`-Objekte

Die Unterstützung verschiedener Formate ist bei der Auswertung von Zeichenketten und der Umwandlung in Objekte noch deutlich wichtiger als bei der Ausgabe. Dieser Abschnitt geht genauer darauf ein.

Um textuell vorliegende Informationen in korrespondierende Objekte umzuwandeln, verwendet man die Methode `parse(String)`. Dabei ist zu beachten, dass je nach gewählter Locale eine Zahl mit Komma "`1,2`" oder Punkt "`1.2`" unterschiedlich interpretiert wird! Folgendes Beispielprogramm verdeutlicht dies anhand der Auswertung der Eingaben "`123,456,789`" und "`123.456.789`" für die Zahlenformate für Deutschland, Frankreich und die USA.

```
public static void main(final String[] args) throws ParseException
{
    final NumberFormat DE_FORMAT = NumberFormat.getInstance(Locale.GERMANY);
    final NumberFormat FR_FORMAT = NumberFormat.getInstance(Locale.FRANCE);
    final NumberFormat US_FORMAT = NumberFormat.getInstance(Locale.US);

    final String[] values = { "123,456,789", "123.456.789" };
    for (final String number : values)
    {
        System.out.println("Value " + number);
        System.out.println("NumberInstance DE  " + DE_FORMAT.parse(number));
        System.out.println("NumberInstance FR  " + FR_FORMAT.parse(number));
        System.out.println("NumberInstance US  " + US_FORMAT.parse(number));
    }
}
```

Listing 13.4 *Ausführbar als* **'NUMBERFORMATPARSEEXAMPLE'**

Führt man das Programm NUMBERFORMATPARSEEXAMPLE aus, so erscheinen folgende Ausgaben auf der Konsole:

```
Value 123,456,789
NumberInstance DE  123.456
NumberInstance FR  123.456
NumberInstance US  123456789
Value 123.456.789
NumberInstance DE  123456789
NumberInstance FR  123
NumberInstance US  123.456
```

Erwartete Unterschiede findet man bei der amerikanischen und den beiden europäischen Formaten. Unerwartet ist jedoch die unterschiedliche Auswertung der punktseparierten Zahl für die deutsche und die französische Locale – das liegt vermutlich daran,

dass im Französischen keine Tausenderpunkte, sondern Leerzeichen genutzt werden. Das konnte ich aber nicht eindeutig durch Tests bestätigen.

13.1.7 Textmeldungen und die Klasse `MessageFormat`

Sind Textmeldungen formatiert auszugeben, hilft dabei die Klasse `MessageFormat`. Deren Objekte werden im Unterschied zu den beiden Klassen `NumberFormat` und `DateFormat` nicht über Fabrikmethoden, sondern über einen Konstruktoraufruf erzeugt. Diesem wird eine Vorlage der auszugebenden Nachricht übergeben:

```
MessageFormat(String pattern)
```

Im Parameter `pattern` lassen sich Platzhalter angegeben, die dann zur Laufzeit geeignet mit den Werten aus dem `Object[]`, umgewandelt mithilfe von `toString()`, beim Aufruf der Methode `format(Object)` ersetzt werden. Folgendes Listing zeigt ein Beispiel mit vier Platzhaltern:

```
public static void main(final String[] args) throws ParseException
{
    final String textTemplate = "Am {1} ist {0}. {2} Sekt bitte {4}!";
    final MessageFormat messageFormat = new MessageFormat(textTemplate);

    final Object[] arguments = { "Karneval",
                                 LocalDate.of(2019, 11, 11), 3 };
    final String formattedText = messageFormat.format(arguments);

    System.out.println("Eingabe: " + textTemplate);
    System.out.println("Formatiert: " + formattedText);
}
```

Listing 13.5 *Ausführbar als* **'MESSAGEFORMATEXAMPLE'**

Die Werte aus dem Array werden über Platzhalter wie `{0}`, `{1}` etc. in die Nachricht eingefügt. Die Nummern entsprechen den Indizes des `Object`-Arrays der Argumente. In der Textvorlage können diese Indizes mehrfach und in beliebiger Reihenfolge angegeben werden oder auch ungenutzt bleiben. Wird ein Index angegeben, für den es keinen Eintrag im `Object`-Array der Werte gibt, so erfolgt keine Ersetzung des Platzhalters, was hier für den Platzhalter 4 deutlich wird.

Startet man das Programm MESSAGEFORMATEXAMPLE, so kommt es zu folgender Ausgabe:

```
Eingabe: Am {1} ist {0}. {2} Sekt bitte {4}!
Formatiert: Am 2019-11-11 ist Karneval. 3 Sekt bitte {4}!
```

Die Klasse `MessageFormat` bietet auch eine statische Methode `format(String, Object...)` für die Ausgabe, wodurch man kein `MessageFormat`-Objekt konstruieren und dessen Objektmethode `format(Object)` aufrufen muss. Im eigenen Sourcecode ist daher folgende Variante kürzer und besser lesbar:

```
final String formattedText = MessageFormat.format(textTemplate, arguments);
```

Erweiterungen der Formatspezifikation

Platzhalter können nicht nur den Index des Arguments enthalten, sondern auch weitere Informationen zur Formatierung. Mit der Syntax

```
{ArgumentIndex, FormatType}
```

kann die Ausgabe angepasst werden. Dabei wird ein Formatierer des entsprechenden Formattyps `number`, `date`, `time` oder `choice` erzeugt. Für `number` entspricht dies beispielsweise `NumberFormat.getInstance(getLocale())`. Bei Bedarf kann sogar der Formatstil angegeben werden:

```
{ArgumentIndex, FormatType, FormatStyle}
```

Für das Format `date` sind hier die Werte `short`, `medium`, `long` und `full` vordefiniert. Für das Format `number` können die Stile `integer`, `currency` und `percent` gewählt werden, die denen der Klasse `NumberFormat` entsprechen. Verdeutlichen wir uns dies anhand eines Beispiels, das auch noch mal die Widrigkeiten das alten Date-API und der Klasse `Date` verdeutlicht:

```
public static void main(final String[] args) throws ParseException
{
    final Date birthday = new Date(117, 1, 7, 21, 54);

    final Object[] arguments = { "MessageFormat", 2, 3.1415, birthday };
    final String textTemplate = "Beispiel für die Klasse {0}. " +
                                "\nWährung: {1, number, currency} " +
                                "\nZahl: {2, number} " +
                                "\nDatum und Uhrzeit: {3, date} um {3, time}";

    final String formattedText = MessageFormat.format(textTemplate, arguments);

    System.out.println("Eingabe: " + textTemplate);
    System.out.println();
    System.out.println("Formatiert: " + formattedText);
}
```

Listing 13.6 *Ausführbar als* **'MESSAGEFORMATEXAMPLE2'**

Führt man das Programm MESSAGEFORMATEXAMPLE2 aus, so kommt es zu folgender, aufs Wesentliche gekürzten Ausgabe:

```
Formatiert: Beispiel für die Klasse MessageFormat.
Währung: 2,00 €
Zahl: 3,142
Datum und Uhrzeit: 07.02.2017 um 21:54:00
```

Fazit

Dieser Abschnitt hat einen kurzen Einstieg in das Thema formatierte Ein- und Ausgabe gegeben und soll als Anregung für eigene Experimente dienen. Zudem wurden Grundkenntnisse zur Internationalisierung aufgebaut, die das Verständnis der im Fol-

genden vorgestellten Themen erleichtern. Abschließend sei noch darauf hingewiesen, dass die Formate auf das alte Datums-API ausgelegt sind und dass Typen des neuen Datums-APIs nur als Strings ausgegeben werden können.

13.1.8 Stringvergleiche mit der Klasse `Collator`

Neben der Darstellung von Informationen mithilfe verschiedener `Format`-Klassen ist es insbesondere für Listen oder Tabellen von Interesse, Texte in einer bestimmten Reihenfolge anzuordnen. Der Vergleich von Strings scheint auf den ersten Blick eine einfache Aufgabe zu sein, da die Klasse `String` das Interface `Comparable<String>` erfüllt und damit auch eine Methode `compareTo(String)` bietet. Damit lassen sich Strings in eine gewünschte Reihenfolge bringen – man kann etwa eine Liste von Namen alphabetisch sortieren. Allerdings beruht der Vergleich der einzelnen Strings auf den Unicode-Werten der enthaltenen Zeichen. In einigen Anwendungsfällen reicht dies bereits aus. Es gibt aber viele Anwendungsfälle, in denen diese vorgegebene Art des Vergleichs nicht zweckmäßig ist – das gilt insbesondere dann, wenn landes- oder sprachspezifische Besonderheiten beachtet werden müssen. Schon ein Vergleich deutscher Wörter gemäß `Comparable<String>` kann unerwartete Ergebnisse liefern. Das liegt daran, dass in der Unicode-Codierung Kleinbuchstaben hinter dem Großbuchstaben '`Z`' und Umlaute hinter dem Kleinbuchstaben '`z`' angeordnet sind.

Wenn Strings landes- bzw. sprachspezifisch entsprechend einer gewählten Locale zu vergleichen sind, wäre es aufwendig und fehleranfällig, diese Funktionalität selbst zu programmieren. Für sprachabhängige Vergleiche sollte man stattdessen die Klasse `java.text.Collator` einsetzen, die wir nun kennenlernen wollen.

Die Klasse `Collator`

Sprachspezifische Vergleiche von Strings kann man mithilfe der Klasse `Collator` auf elegante Weise realisieren. Eine Instanz davon erhält man über verschiedene Fabrikmethoden, die in der Klasse `Collator` definiert sind. Die Methode `getInstance()` liefert ein `Collator`-Objekt für die Default-Locale, wohingegen die Methode `getInstance(Locale)` ein `Collator`-Objekt für das als Parameter übergebene `Locale`-Objekt zurückgibt.

Die Klasse `Collator` implementiert das Interface `Comparator<Object>` und bietet daher eine Methode `compare(Object, Object)`.[4] Damit lassen sich zwei beliebige Strings miteinander vergleichen und ordnen. Insbesondere fällt damit der häufige Anwendungsfall des Sortierens von Strings leicht. Dabei helfen die Utility-Methoden `sort(T[], Comparator<? super T>)` und `sort(List<T>, Comparator<? super T>)` aus der Utility-Klasse `Arrays` bzw. `Collections`, die

[4] In der Methode `compare(Object, Object)` werden beide Parameter direkt auf `String` gecastet. Korrekterweise müsste die Klasse `Collator` daher das auf die Klasse `String` typisierte Interface `Comparator<String>` erfüllen. Allgemeiner könnte man den Typ `CharSequence` statt `String` verwenden.

beide aus dem Collections-Framework stammen. Details zum Interface `Comparator<T>` beschreibt Abschnitt 6.1.8.

Zur Verdeutlichung der Funktionalität und Arbeitsweise eines `Collator`s soll ein Vergleich von Objekten einer Klasse `Person` realisiert werden.

Beispiel

Die im Listing gezeigte Klasse `Person` erfüllt das Interface `Comparable<Person>` und realisiert daher die Methode `compareTo(Person)`. Damit lassen sich `Person`-Objekte miteinander vergleichen und ordnen. In diesem Beispiel basiert der Vergleich auf dem Attribut `lastname`. Die eigentliche Vergleichsfunktionalität wird hier durch einen `Collator` für die `Locale.GERMANY` gemäß deutschen Regeln bereitgestellt:

```
public class Person implements Comparable<Person>
{
    private static final Collator collator =
                                 Collator.getInstance(Locale.GERMANY);

    private final String lastname;

    Person(final String lastname)
    {
        this.lastname = lastname;
    }

    public int compareTo(final Person other)
    {
        return collator.compare(getLastname(), other.getLastname());
    }

    // ...
}
```

Vergleich im Detail steuern

Das einführende Beispiel zeigt einen einfachen Anwendungsfall eines `Collator`s. Der Vergleich von Strings lässt sich bei Bedarf im Detail beeinflussen. Dazu kann man durch Aufruf der Methode `setStrength(int)` eine sogenannte »Strenge« vorgeben, die bestimmt, wann Zeichen als unterschiedlich gelten sollen. Dazu übergibt man folgende Konstanten:

- `Collator.PRIMARY` – Es werden lediglich verschiedene Buchstaben als unterschiedlich erkannt. Beim Vergleich werden demnach Groß- und Kleinschreibung sowie eventuell vorhandene Akzente und Umlaute außer Acht gelassen, d. h., ein 'Ä' wird beim Sortieren wie ein 'a' behandelt.[5]

[5] In vielen Sprachen wird durch einen Akzent oder Umlaut kein wirklich eigenständiger Buchstabe beschrieben. In einigen Sprachen, etwa Dänisch, ist das anders. Dann wird ein Unterschied im Umlaut, z. B. 'ø' statt 'o', auch als primärer Unterschied gewertet.

- `Collator.SECONDARY` – Diese Vergleichsart berücksichtigt zusätzlich zur Einstellung `PRIMARY` auch Akzente und Umlaute. Unterschiede in der Groß- und Kleinschreibung beeinflussen die Sortierung jedoch nicht.
- `Collator.TERTIARY` – Diese Vergleichsart ist »strenger« als die Einstellung `SECONDARY`, denn sie beachtet auch Unterschiede in der Groß- und Kleinschreibung. Es gilt etwa '`a`' < '`A`'. ***Das ist die Standardeinstellung.***
- `Collator.IDENTICAL` – Hierbei werden alle Unicode-Zeichen als unterschiedlich angesehen. Das gilt auch für verschiedene nicht darstellbare Unicode-Zeichen, die bei den anderen Einstellungen teilweise als gleich angesehen werden.

Zur Veranschaulichung zeigt folgende Tabelle die Sortierreihenfolge bei einem Vergleich der Strings `Abc` und `Äbc` für Umlaute sowie der Strings `Hallo` und `hallo` für Groß- bzw. Kleinschreibung für verschiedene `Collator`-Strengegrade. Im Anschluss daran erstelle ich ein Beispielprogramm zur weiteren Verdeutlichung.

Tabelle 13-1 *Einflüsse der* `Collator`*-Strengegrade*

Sortierungsvariante	Reihenfolge der Strings
`Comparable<String>`	Abc < Hallo < hallo < Äbc
`PRIMARY`	(Abc = Äbc) < (Hallo = hallo)
`SECONDARY`	Abc < Äbc < (Hallo = hallo)
`TERTIARY`	Abc < Äbc < hallo < Hallo

Das folgende Listing demonstriert die Auswirkungen verschiedener Einstellungen für die »Strenge« auf den Vergleich von Strings. Dazu wird ein `String[]` mit Werten initialisiert, die besonders geeignet sind, die Auswirkungen unterschiedlicher Strengegrade beobachten zu können. Dabei werden auch spezielle Unicode-Zeichen genutzt, damit insbesondere die Einstellung `IDENTICAL` verständlich wird:

```
public final class CollatorStrengthExample
{
    // Diese Werte werden verglichen, inklusive spezieller Unicode-Zeichen
    private static final String[] EXAMPLE_VALUES = { "\u0001BC", "\u0002BC",
                                                     "ABC", "ÄBC", "abc",
                                                     "Maße", "Masse" };

    public static void main(final String[] args)
    {
        System.out.println("CollatorStrengthExample");
        System.out.println("Values: " + Arrays.toString(EXAMPLE_VALUES) + "\n");

        final Collator collator = Collator.getInstance(Locale.GERMANY);

        for (final CollatorStrength strength : CollatorStrength.values())
        {
            // Achtung: Kopie des Arrays ist wichtig, da der sort-Befehl
            // ansonsten das Ausgangsarray ändern würde!
            sortByCollator(collator, strength, EXAMPLE_VALUES.clone());
```

```
        }
    }


    private static void sortByCollator(final Collator collator,
                                       final CollatorStrength collatorStrength,
                                       final String[] exampleValues)
    {
        // Gemäß Strenge sortieren und ausgeben
        collator.setStrength(collatorStrength.collatorStrengthValue);
        Arrays.sort(exampleValues, collator);
        System.out.println("Collator-Strength: " + collatorStrength + " => " +
                           Arrays.toString(exampleValues));

        // Aufbereiten der Vergleichsordnung
        final String orderingInfo = buildOrderingInfo(collator, exampleValues);
        System.out.println("Using ordering: [" + orderingInfo + "]" + "\n");
    }

    enum CollatorStrength
    {
        PRIMARY(Collator.PRIMARY), SECONDARY(Collator.SECONDARY),
        TERTIARY(Collator.TERTIARY), IDENTICAL(Collator.IDENTICAL);

        final int collatorStrengthValue;

        CollatorStrength(final int collatorStrengthValue)
        {
            this.collatorStrengthValue = collatorStrengthValue;
        }
    }

    // ...
```

Listing 13.7 *Ausführbar als* **'COLLATORSTRENGTHEXAMPLE'**

Startet man das Programm COLLATORSTRENGTHEXAMPLE, kommt es zu folgender, für die bessere Lesbarkeit leicht modifizierten Ausgabe, wobei die zwei besonderen Unicode-Zeichen nicht druckbar sind und daher hier als '?' dargestellt sind:

```
Values: [?BC, ?BC, ABC, ÄBC, abc, Maße, Masse]

Collator-Strength: PRIMARY => [ÄBC, ABC, abc, ?BC, ?BC, Maße, Masse]
Using ordering: ['?BC' = '?BC', '?BC' > 'ÄBC', 'ÄBC' = 'ABC',
                 'ABC' = 'abc', 'abc' < 'Maße', 'Maße' = 'Masse']

Collator-Strength: SECONDARY => [ABC, abc, ÄBC, ?BC, ?BC, Maße, Masse]
Using ordering: ['?BC' = '?BC', '?BC' > 'ÄBC', 'ÄBC' > 'ABC',
                 'ABC' = 'abc', 'abc' < 'Maße', 'Maße' = 'Masse']

Collator-Strength: TERTIARY => [abc, ABC, ÄBC, ?BC, ?BC, Masse, Maße]
Using ordering: ['?BC' = '?BC', '?BC' > 'ÄBC', 'ÄBC' > 'ABC',
                 'ABC' > 'abc', 'abc' < 'Maße', 'Maße' > 'Masse']

Collator-Strength: IDENTICAL => [abc, ABC, ÄBC, ?BC, ?BC, Masse, Maße]
Using ordering: ['?BC' < '?BC', '?BC' > 'ÄBC', 'ÄBC' > 'ABC',
                 'ABC' > 'abc', 'abc' < 'Maße', 'Maße' > 'Masse']
```

Anhand der Ausgaben erkennen wir den Einfluss des gewählten Strengegrads sehr schön. Zum besseren Verständnis und zum leichteren Nachvollziehen werden die Er-

gebnisse der Vergleiche textuell dargestellt, etwa `'ABC' = 'abc'`. Diese Informationen werden durch die Methode `buildOrderingInfo(Collator, String[])` aufbereitet. Für den allgemeinen Teil der Darstellung des Vergleichs wird vom konkreten Typ `Collator` abstrahiert und eine generische Hilfsmethode `buildComparatorInfo(Comparator<T>, T, T)` für den allgemeineren Typ `Comparator<T>` definiert. Dadurch ist diese bei Bedarf auch in anderen Anwendungsfällen einsetzbar.

```
private static String buildOrderingInfo(final Collator collator,
                                        final String[] exampleValues)
{
    String orderingInfo = "";
    for (int i = 0; i < exampleValues.length - 1; i++)
    {
        orderingInfo += buildComparatorInfo(collator, exampleValues[i],
                                                      exampleValues[i + 1]);

        // ', ' für alle Einträge hinzufügen, außer letztem
        if (i < exampleValues.length - 2)
            orderingInfo += ", ";
    }
    return orderingInfo;
}

public static <T> String buildComparatorInfo(final Comparator<T> comparator,
                                             final T obj1, final T obj2)
{
    final String order;
    if (comparator.compare(obj1, obj2) < 0)
        order = " < ";
    else if (comparator.compare(obj1, obj2) > 0)
        order = " > ";
    else
        order = " = ";

    return "'" + obj1 + "'" + order + "'" + obj2 + "'";
}
```

Das war ein eher künstliches Beispiel, das dazu diente, ein erstes Verständnis für die verschiedenen Strengegrade aufzubauen. In der Praxis wird man einen `Collator` dazu nutzen, Attribute einer Klasse zu vergleichen, wie wir dies bereits für die Klasse `Person` getan haben. Kommen wir zur Betrachtung eines Details zur Optimierung noch einmal auf diese zurück.

Optimierung des Vergleichs

Die durch `Collator`-Instanzen realisierten Vergleiche sind langsamer als »normale« Stringvergleiche, die auf dem Interface `Comparable<String>` und der Methode `compareTo(String)` basieren. Für Performance-kritische Applikationen kann man die Klasse `java.text.CollationKey` zur Optimierung der Ausführungsgeschwindigkeit nutzen. Diese Klasse stellt vereinfachend gesagt eine Ordnungszahl dar, die im Voraus berechnet wird. Im nachfolgenden Listing ist der Einsatz eines `CollationKeys` für das Attribut `lastname` einer Klasse `Person` gezeigt. Instanzen der Klasse

`CollationKey` erzeugt man nicht per Konstruktor, sondern über die Methode `getCollationKey(String)` der Klasse `Collator`:

```
public class Person implements Comparable<Person>
{
    private static final Collator collator =
                                 Collator.getInstance(Locale.GERMANY);

    private final String lastname;

    // zur Performance-Optimierung
    private final CollationKey key;

    Person(final String lastname)
    {
        this.lastname = lastname;
        this.key = collator.getCollationKey(lastname);
    }

    @Override
    public int compareTo(final Person otherPerson)
    {
        return this.key.compareTo(otherPerson.key);
    }
}
```

Während der Konstruktion einer Instanz der Klasse `Person` entsteht zwar ein geringer Aufwand für die Erzeugung der `CollationKey`-Instanz, jedoch fällt dieser einmalig an und ist vergleichsweise gering zu dem Aufwand, der durch wiederholte »normale« `Collator`-Vergleiche ohne diese `CollationKey`-Optimierung entsteht.

Sind Vergleiche eher selten, so kann man zusätzlich Lazy Initialization (vgl. Abschnitt 22.3) einsetzen, sodass die Berechnungsaufwände für die `CollationKey`s nur anfallen, wenn tatsächlich Vergleiche erfolgen.

Achtung: Vergleiche von `CollationKey`s

Instanzen der Klasse `CollationKey` dürfen nur miteinander verglichen werden, wenn sie vom selben `Collator`-Objekt stammen, ansonsten kommt es zu unerwarteten und möglicherweise auch falschen Vergleichsergebnissen.

13.2 Programmbausteine zur Internationalisierung

Nachdem wir zuvor bereits einige Grundlagen zur Internationalisierung und dazu nützliche Java-Klassen kennengelernt haben, stellt dieser Abschnitt komplexere Programmbausteine vor, die bei der Internationalisierung eigener Anwendungen helfen. In Abschnitt 13.2.1 betrachten wir die Unterstützung mehrerer Datumsformate.

13.2.1 Unterstützung mehrerer Datumsformate

Häufig muss man mehrere Datumsformate unterstützen. Dies ist in der Regel bei der Eingabe von Datumswerten in einer Benutzeroberfläche der Fall. Hier ist es für einen Nutzer angenehm, Datums- oder Zeitangaben nicht *exakt* in dem Format eingeben zu müssen, in dem sie vom Programm später gespeichert und weiterverarbeitet werden. Machen wir es konkret: Würde beispielsweise ein Programm Datumswerte mit vierstelligen Jahreszahlen und Sekundenangaben im Format '`dd.MM.yyyy HH:mm:ss`' verarbeiten, so ist ein derart starres Format doch unhandlich für einen Anwender. Zum Teil möchte dieser lediglich eine zweistellige Jahreszahl angeben. Auch die Uhrzeit soll häufig nicht sekundengenau spezifiziert werden. In beiden Fällen können Eingaben verarbeitet werden, wenn das Parsing fehlertolerant arbeitet und für nicht angegebene Werte sinnvolle Standardwerte vorsieht.

Im Folgenden zeige ich, wie man durch den Einsatz einer Utility-Klasse fehlertolerant parsen kann. Zum einen befreit dies den Entwickler der Applikation von den Details des Parsings. Zum anderen erlaubt dies dem Benutzer mehr Flexibilität bei der Eingabe von Datumswerten. Die Applikation kann zudem mit der gewünschten und für die Anwendungsfälle geeigneten Datumsrepräsentation arbeiten, ohne dass ein Nutzer dieses Format kennen und bei einer Eingabe beachten muss.

Auswertung von Datumsformaten

Nehmen wir an, wir wollten Datumsangaben mit zwei- oder vierstelligen Jahresangaben sowie optionaler Angabe von Sekunden unterstützen. Eine erste Idee wäre, nacheinander ein Parsing mit verschiedenen Datumsformaten durchzuführen. Man definiert dazu verschiedene erlaubte Datumsformate vom Typ `SimpleDateFormat`, etwa:

```
DateFormat df1 = new SimpleDateFormat("dd.MM.yy HH:mm:ss");
DateFormat df2 = new SimpleDateFormat("dd.MM.yy HH:mm");
DateFormat df3 = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
DateFormat df4 = new SimpleDateFormat("dd.MM.yyyy HH:mm");
```

Sofern man mit Java 8 arbeitet, bietet sich dafür die Klasse `DateTimeFormatter` an:

```
DateTimeFormatter df1 = DateTimeFormatter.ofPattern("dd.MM.yy HH:mm:ss");
DateTimeFormatter df2 = DateTimeFormatter.ofPattern("dd.MM.yy HH:mm");
DateTimeFormatter df3 = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm:ss");
DateTimeFormatter df4 = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm");
```

Definition einer Utility-Klasse Zur Umwandlung von textuellen Eingaben in `Date`-Objekte erstellen wir eine Utility-Klasse `DateParseUtils` und definieren die obigen Variablen dort `static`. Einen ersten Baustein bildet die im Folgenden vorgestellte Methode `parseDate(String, DateFormat...)`. Dieser werden sowohl eine Datumseingabe als auch eine variable Anzahl erlaubter Datumsformate als Parameter übergeben. Um Klienten den Aufruf zu erleichtern, empfiehlt sich hier der Einsatz

des Sprachfeatures Varargs. Somit können ein einzelnes Datumsformat, eine kommaseparierte Aufzählung oder ein Array als Eingabe genutzt werden.

```
public static Date parseDate(final String dateAndTime,
                             final DateFormat... supportedDateFormats)
                             throws ParseException
{
    for (final DateFormat currentFormat : supportedDateFormats)
    {
        try
        {
            // Rückgabe des Datums bei erfolgreichem Parsing
            return currentFormat.parse(dateAndTime);
        }
        catch (final ParseException e)
        {
            // Ignorieren und mit nächstem Format versuchen
        }
    }
    // Kein Format erlaubt eine Umwandlung
    throw new ParseException(dateAndTime, 0);
}
```

Für jedes `DateFormat`-Objekt wird mit dessen `parse(String)`-Methode versucht, die Eingabe in ein `Date`-Objekt umzuwandeln. Ist dies nicht möglich, so wird eine `ParseException` ausgelöst. Diese wird hier zwar abgefangen, aber nicht weiter behandelt, da die Parsingaufgabe an ein nachfolgendes `DateFormat`-Objekt übertragen wird. Dieser Vorgang wiederholt sich, bis entweder das Parsing erfolgreich war oder aber kein weiteres Datumsformat verbleibt. In diesem Fall wird kein gültiges `Date`-Objekt zurückgeliefert, sondern explizit eine `ParseException` ausgelöst.

Anpassungen für Java 8 Für Java 8 und das dort eingeführte Date and Time API ändert sich die Verarbeitung nur in Details: Zum einen nutzen wir statt `DateFormat` die Klasse `DateTimeFormatter`. Zum anderen erfolgt das Parsing nicht auf dem Format, sondern durch die jeweilige Klasse unter Angabe des Formats wie folgt:

```
public static LocalDateTime parseDate(final String dateAndTime,
                              final DateTimeFormatter... supportedDateFormats)
                              throws DateTimeParseException
{
    for (final DateTimeFormatter currentFormat : supportedDateFormats)
    {
        try
        {
            return LocalDateTime.parse(dateAndTime, currentFormat);
        }
        catch (final DateTimeParseException e)
        {
            // Ignorieren und mit nächstem Format versuchen
        }
    }
    // Kein Format erlaubt eine Umwandlung
    throw new DateTimeParseException("unparsable input", dateAndTime, 0);
}
```

Einsatz der Utility-Klasse Betrachten wir nun eine `main()`-Methode, die die eben definierte Hilfsmethode nutzt, um den Datumswert Silvester 2019 zu parsen:

```
private static final String SILVESTER = "31.12.2019";

public static void main(final String[] args)
{
    final DateFormat[] supportedFormats = { df1, df2, df3, df4 };

    try
    {
        final Date date = DateParseUtils.parseDate(SILVESTER, supportedFormats);
        System.out.println("Parsed '" + SILVESTER + "' into date " + date);
    }
    catch (final ParseException ex)
    {
        System.out.println("Parsing failed: value='" + SILVESTER +"'");
    }
}
```

Listing 13.8 *Ausführbar als* **'MULTIPLEDATEFORMATPARSINGEXAMPLE1'**

Führt man das Programm MULTIPLEDATEFORMATPARSINGEXAMPLE1 aus, so wird kein Datumswert ausgegeben, sondern die Fehlermeldung »`Parsing failed: value='31.12.2019'`«. Es fehlt jedoch ein Hinweis auf die Fehlerursache.

Aufbereitung von aussagekräftigen Fehlermeldungen Beim Auftreten von Parsingfehlern helfen informative Fehlermeldungen, mögliche Ursachen besser erkennen zu können. Würde jede Applikation selbst wieder die Aufbereitung von Fehlermeldungen implementieren, so wäre dies aufwendig. Es bietet sich an, die gerade erstellte Utility-Klasse `DateParseUtils` um genau diese Funktionalität zu erweitern.

Die im folgenden Listing gezeigte Methode `buildErrorMessage(String, DateFormat...)` bereitet eine aussagekräftige Fehlermeldung auf, die alle beim Parsing akzeptierten Datumsformate aufzählt:

```
public static String buildErrorMessage(final String dateAndTime,
                                       final DateFormat... supportedDateFormats)
{
    // Aufbereiten der Muster
    final String[] patterns = new String[supportedDateFormats.length];
    for (int i = 0; i < supportedDateFormats.length; i++)
    {
        if (supportedDateFormats[i] instanceof SimpleDateFormat)
        {
            patterns[i] = "'" + ((SimpleDateFormat)
                                 supportedDateFormats[i]).toPattern() + "'";
        }
        else
        {
            patterns[i] = "'" + supportedDateFormats[i] + "'";
        }
    }

    return "Parsing error: value='" + dateAndTime + "'\nSupported formats: " +
           Arrays.toString(patterns);
}
```

Erweitern wir die `main()`-Methode um einen Aufruf dieser Fehlerbehandlung:

```
public static void main(final String[] args)
{
    final DateFormat[] supportedFormats = { df1, df2, df3, df4 };

    try
    {
        final Date date = DateParseUtils.parseDate(SILVESTER, supportedFormats);
        System.out.println("Parsed '" + SILVESTER + "' into date " + date);
    }
    catch (final ParseException ex)
    {
        System.out.println(DateParseUtils.buildErrorMessage(SILVESTER,
                                                            supportedFormats));
    }
}
```

Listing 13.9 *Ausführbar als* **'MULTIPLEDATEFORMATPARSINGEXAMPLE2'**

Das Programm MULTIPLEDATEFORMATPARSINGEXAMPLE2 gibt Folgendes aus:

```
Parsing error: value='31.12.2019'
Supported formats: ['dd.MM.yy HH:mm:ss', 'dd.MM.yy HH:mm',
                    'dd.MM.yyyy HH:mm:ss', 'dd.MM.yyyy HH:mm']
```

Offensichtlich ist diese Reaktion auf einen Fehler beim Parsing sehr hilfreich, um die Fehlerursache zu erkennen: Es wird neben der Angabe eines Datums auch die Angabe einer Uhrzeit erwartet. Letztere erfolgt hier jedoch nicht und löst so den Fehler aus.

Anpassungen für Java 8 Mithilfe der Klasse `DateTimeFormatter` kann man leider keine Angaben zum Muster erzeugen. Stattdessen kann man aber die Verarbeitung ein wenig modifizieren. Als Eingabe nutzt man dann einfach eine Menge von Strings als Muster und erzeugt die jeweilige `DateTimeFormatter`-Instanz on the fly. Zum Aufbereiten der Fehlermeldung nutzt man dann die Streams aus Java 8 wie folgt:

```
public static String buildErrorMessage(final String dateAndTime,
                                       final List<String> supportedDateFormats)
{
    final List<String> formats = supportedDateFormats.stream().
                                        map(pattern -> "'" + pattern + "'").
                                        collect(Collectors.toList());

    return "Parsing error: value='" + dateAndTime +
           "'\nSupported formats: " + formats;
}
```

Erweiterung: Einlesen von Formatstrings aus einer Property-Datei

Wir haben erste Bausteine für eine flexible Datumsprüfung geschrieben, die wir nun sinnvoll erweitern wollen, um flexibel weitere Datumsformate unterstützen zu können. Weil sich Anforderungen oftmals im Laufe der Zeit wandeln, wäre es unpraktisch, für jedes weitere Datumsformat jeweils neue Konstanten im Sourcecode definieren zu müs-

sen. Das würde außerdem eine Neukompilierung erfordern, um die neue Funktionalität in das Programm zu integrieren. Eine flexible Konfigurierbarkeit erreicht man, wenn eine Darstellung und Speicherung in Form von Schlüssel-Wert-Paaren in einer Datei erfolgt. Eine Menge gültiger Datumsformate könnte man in etwa wie folgt definieren:

```
dateformat1=dd.MM.yy HH:mm:ss
dateformat2=dd.MM.yy HH:mm
dateformat3=dd.MM.yyyy HH:mm:ss
```

Folgende Methode `readDateFormatsFromFile(String)` nutzt die bereits bekannte JDK-Bibliotheksklasse `Properties` (vgl. Abschnitt 9.5.2), um diese Properties einzulesen und das Muster des Datumsformats zu extrahieren:

```
public static List<String> readDateFormatsFromStream(final InputStream resIs)
                throws IOException, FileNotFoundException
{
    try (final InputStream is = new BufferedInputStream(resIs))
    {
        final Properties dateFormatProperties = new Properties();
        dateFormatProperties.load(is);

        return dateFormatProperties.values().stream().
                                          map(obj -> (String)obj).
                                          collect(Collectors.toList());
    }
}
```

Wir setzen in folgenden Beispiel alle drei Methoden wie folgt ein, um die Datumsformate aus einer Datei zu lesen und einen nun gültigen String mit Uhrzeitangabe zu parsen:

```
public static void main(final String[] args)
{
   final String dateFormatFileName = "DateFormat.properties";
   final InputStream dateFormatIs = MultipleDateFormatParsingExample3Jdk8.class.
                                    getResourceAsStream(dateFormatFileName);
   try
   {
      final List<String> supportedFormats =
                         readDateFormatsFromStream(dateFormatIs);
      final String silvester = "31.12.2019 18:00";
      try
      {
         final LocalDateTime date = parseDate(silvester, supportedFormats);
         System.out.println("Parsed '" + silvester + "' into " + date);
      }
      catch (final DateTimeParseException ex)
      {
         System.out.println(buildErrorMessage(silvester, supportedFormats));
      }
   }
   catch (final IOException ex)
   {
      System.out.println("No DateFormat-File: " + dateFormatFileName);
   }
}
```

Listing 13.10 *Ausführbar als* **'MULTIPLEDATEFORMATPARSINGEXAMPLE3JDK8'**

Als Ausgabe des Programms MULTIPLEDATEFORMATPARSINGEXAMPLE3JDK8 erhalten wir nach dieser Korrektur folgende Ausgabe:

```
Parsed '31.12.2019 18:00' into 2019-12-31T18:00
```

Fazit

Das war ein kurzer Einstieg in das Thema Auswertung mehrerer Datumsformate. Sie besitzen damit das notwendige Wissen, um eigene Erweiterungen zu realisieren. Folgen Sie dem Rat, nützliche Funktionalität in Utility-Klassen auszulagern. Ihre Applikationen werden dadurch schlanker und eleganter.

13.2.2 Fazit und Ausblick

In diesem Abschnitt wurden verschiedene nützliche Bausteine zur Internationalisierung vorgestellt. Dabei wurde zunächst Basisfunktionalität entworfen und diese mit bereits bekannten Programmbausteinen kombiniert. Dadurch konnte der Aufwand zur Internationalisierung in der Applikation selbst gering gehalten werden.

Es gibt diverse weitere Einflussfaktoren auf die Internationalisierung, die hier nicht betrachtet wurden, da diese nicht im Fokus dieses Buchs liegen. Dazu gehören unter anderem Themen wie der Einfluss von Zeitzonen und die Schreibrichtung von Text.

III Wichtige Neuerungen in Java 12 bis 15

14 Neues und Änderungen in den Java-Versionen 12 bis 15

Dieses Kapitel beschreibt die Neuerungen aus Java 12 bis 15 als Einheit, weil alles aus Java 12, 13 und 14 in Java 15 enthalten ist. Das Release von Java 15 bietet somit kumulativ diverse interessante Neuerungen. Leider sind nur einige davon schon in einer finalen Version bereitgestellt, während viele noch als Preview[1] in das JDK integriert wurden – nur die ersten beiden sind final, die restlichen sind Preview-Features.

- Mit Java 13 wurden **Text Blocks**, damals unter dem Namen Multi Line Strings, als Preview vorgestellt, um mehrzeilige Texte als Literal und ohne viel Mühe definieren zu können. Insbesondere kann auf das Escaping von Zeilenumbrüchen, Anführungszeichen usw. verzichtet werden. Mit Java 14 wurden zwei neue Escape-Sequenzen hinzugefügt. In Java 15 hat das Feature den finalen Stand erreicht.
- **Switch Expressions** sind eine großartige Erweiterung, die viele der Schwachstellen der alten Syntax adressieren und lesbare, kürze und präzisere `switch`-Ausdrücke ermöglichen. Das ist in Java 15 ein finales Feature.
- Sogenannte **Records** bieten eine extrem kompakte Schreibweise zur Definition von Klassen mit unveränderlichen Daten, die automatisch das passende API in Form von nur lesenden Accessor-Methoden für alle implizit definierten Attribute sowie weitere zentrale Methoden realisieren,
- **Pattern Matching** wurde im Kontext von `instanceof` in die Sprache integriert. Damit lassen sich künstliche Hilfsvariablen und unschöne Casts vermeiden, wodurch der Sourcecode an Klarheit und Verständlichkeit gewinnt.
- Als Syntaxneuerungen finden wir **Sealed Types** und **lokale Enums und Interfaces**. Erstes erlauben die Kontrolle über Vererbungshierarchien. Letzteres ermöglicht Enums und Interfaces innerhalb von Methoden definieren zu können, was bisher nur für Klassen möglich war. Beide Features sind vermutlich eher selten nötig, dann aber sehr nützlich.

Darüber hinaus finden sich noch in diversen APIs kleinere Ergänzungen, etwa bei Strings oder in der Utility-Klasse `Files`. Besonders erwähnenswert ist sicher auch die Integration des Microbenchmark-Frameworks JMH.

[1] Preview-Features sind noch nicht finale Funktionalitäten, die veröffentlicht werden, um Erfahrungen damit zu sammeln und Rückmeldungen aus der Java-Community zu erhalten.

14.1 Syntaxneuerungen

In diesem Abschnitt schauen wir uns verschiedene Syntaxneuerungen an. Wir starten mit Text Blocks und Switch-Expressions. Danach thematisiere ich Records. Zum Abschluss beschäftigen wir uns mit zwei in Java 15 hinzugekommenen Syntaxerweiterungen, nämlich Sealed Types und die lokalen Enums und Interfaces.

14.1.1 Text Blocks

Mit Java 13 wurde eine Syntaxerweiterung eingeführt, die eine lang ersehnte Vereinfachung in Form mehrzeiliger Strings bringt. Diese lassen sich nun ohne mühselige Verknüpfungen definieren. Besser noch, man kann auch auf ein fehlerträchtiges Escaping verzichten, wodurch unter anderem der Umgang mit SQL-Befehlen sowie die Definition von JavaScript und JSON in Java-Sourcecode erleichtert wird. Intern werden Text Blocks wie normale Strings gehandhabt, weshalb man alle bekannten Stringmethoden aufrufen kann.

Syntax

Text Blocks beschreiben mehrzeilige Strings. Diese beginnen und enden mit je drei Anführungszeichen, wobei der eigentliche Inhalt auf einer neuen Zeile angegeben werden muss:

```
System.out.println("""
          -Ich bin ein-
          -Text Block-
        """);
```

Schauen wir uns die Ausgabe und deren Einrückung an:

```
  -Ich bin ein-
  -Text Block-
```

Wie kommt es zu der Einrückung von lediglich zwei Leerzeichen, wo doch im `println()`-Aufruf einige initiale Leerzeichen vor den beiden Textzeilen vorhanden sind? Die Antwort ist ganz einfach: Die unteren drei Anführungszeichen bestimmen den Start der Einrückung. Darüber hinaus wird auch die letzte Zeile mit einem Zeilenumbruch versehen, wodurch es zu einer zusätzlichen Leerzeile kommt.

Sofern keine Einrückung benötigt wird, ist es oftmals besser, den Abschluss des Text Blocks mit in die letzte Zeile aufzunehmen:

```
var firstTextBlockNoIndentation = """
        -Ich bin ein-
        -Text Block-""";
System.out.println(firstTextBlockNoIndentation);
```

So erhält man das etwas intuitivere Ergebnis eines zweizeiligen Strings ohne nachfolgende Leerzeile, allerdings ohne Einrückung zu Beginn der Zeilen:

```
-Ich bin ein-
-Text Block-
```

Es sei nochmals kurz erwähnt: In dem Fall kann man keine Einrückung mehr vorgeben.

Besonderheit Escaping Neben der Mehrzeiligkeit lassen sich in Text Blocks auch doppelte Anführungszeichen direkt im String verwenden – ohne Escaping, mit den folgenden Ausnahmen und Besonderheiten:

- Hinter dem einführenden `"""` dürfen nur noch Leerzeichen stehen (werden ignoriert), aber sonst nichts – nicht einmal ein Kommentar!
- Soll der Text Block drei Anführungszeichen enthalten, so müssen diese maskiert werden.
- Dies gilt ebenfalls für den Backslash (außer ganz am Ende einer Zeile, später dazu mehr bei der Beschreibung der Erweiterungen in Java 14).

Schauen wir uns dies nun anhand eines Beispiels an:

```
System.out.println("""
                   First 'line' simple quotes
                   Second "line" double quotes
                   Third line \""" three quotes
                   Fourth line no quotes, just \\ :-)""");
```

Die Ausgabe sieht folgendermaßen aus:

```
First 'line' simple quotes
Second "line" double quotes
Third line """ three quotes
Fourth line no quotes, just \ :-)
```

Besonderheit Platzhalter In Text Blocks kann man Platzhalter definieren und durch Aufruf von `formatted()` mit Werten befüllen – diese Methode entspricht in ihrer Handhabung exakt der statischen Methode `String.format()`:

```
String placeholders = """
       Michael %s hat am "%tF"
       %d Bücher in '%s' gekauft.
       """.formatted("Inden", LocalDate.of(2020, 1, 20), 7, "Bremen");
```

Durch diese Anweisungen wird folgende Ausgabe produziert:

```
Michael Inden hat am "2020-01-20"
7 Bücher in 'Bremen' gekauft.
```

Beispiel 1: JavaScript

Nachfolgend sehen wir die Definition eines JavaScript-Code-Schnipsels mithilfe von Text Blocks:

```
String javaScriptCode = """
       function hello() {
           print("Hello World");
       }
       hello();
       """;
```

Welch eine elegante Schreibweise! Im Gegensatz dazu muss man mit Java 11 noch ziemlich viel Escaping verwenden und Zeilenumbrüche in den String integrieren:

```
String javaScriptCodeOld = "function hello() {\n" +
       "    print(\"Hello World\");\n" +
       "}\n" +
       "hello();\n";
```

Beispiel 2: HTML-Code

Auch wenn man HTML-Code in Java aufbereiten möchte, sind die mehrzeiligen Strings sehr hilfreich. Bisher musste man umständlich beispielsweise Folgendes schreiben:

```
var helloWorldHtmlOld = "<html>\n" +
                        "    <body>\n" +
                        "            <p>Hello World</p>\n" +
                        "    </body>\n" +
                        "</html>\n";
```

In der Tat ist es viel angenehmer, dies wie folgt anzugeben:

```
var helloWorldHtml = """
                     <html>
                         <body>
                             <p>Hello World</p>
                         </body>
                     </html>
                     """;
```

Beispiel 3: Definition von JSON / XML

Das Datenformat JSON (JavaScript Object Notation) ist heutzutage aus dem Programmieralltag, insbesondere im Kontext von REST-Services[2], kaum mehr wegzudenken. Zwar wäre eine native Unterstützung von JSON in Java SE sehr wünschenswert, jedoch ist diese nicht in Sicht.[3] Allerdings wird mit Text Blocks zumindest eine textuelle Beschreibung deutlich erleichtert:

```
String jsonObj = """
                 {
                     "name" : "Mike",
                     "birthday" : "1971-02-07",
                     "comment" : "Text blocks are nice!"
                 }
                 """;
```

Auch einfache XML-Strukturen kann man mithilfe von Text Blocks und Platzhaltern wie folgt definieren und befüllen:

```
String xmlString = """
                   <customer>
                      <firstname>%s</firstname>
                      <lastname>%s</lastname>
                      <birthday>%s</birthday>
                   </customer>
                   """.formatted("Michael", "Inden", "07.02.1971");
```

Beispiel 4: Definition von SQL und Queries in Repositories

Kommen wir zum letzten Beispiel für Text Blocks. Bei Abfragen mit SQL müssen oftmals doppelte Anführungszeichen in die Kommandos eingefügt werden, was ganz natürlich erfolgen kann, wenn man einen Text Block nutzt:

```
String multiLineSQL = """
                      SELECT LAST_NAME FROM CUSTOMER
                      WHERE CITY = "ZÜRICH" AND FIRST_NAME LIKE 'MI%'
                      """;
```

Ebenso lassen sich auch komplexere Queries in Spring Data Repositories recht lesbar gestalten, etwa die Gehaltserhöhung um einen gewissen Wert:

```
@Modifying
@Query("""
       UPDATE Employee emp
       SET emp.salary = emp.salary + :amount
       WHERE emp.deptId = :deptId
       """)
void increaseSalary(@Param("deptId") long deptId, @Param("amount") int amount);
```

[2] Für Details zu REST-Services verweise ich Sie auf mein Buch »Der Java-Profi: Persistenzlösungen und REST-Services« [39].

[3] In Java EE sieht die Situation deutlich besser aus. Dort gibt es beispielsweise den Typ `javax.json.JsonObject`.

Erweiterungen in Java 14

Ergänzend zum Preview in Java 13 bringt Java 14 zwei neue Escape-Sequenzen. Diese ermöglichen eine feingranulare Kontrolle der Verarbeitung von Zeilenumbrüchen und Whitespaces: `\` (gefolgt von einem Zeilenumbruch) und `\s`. Erstere kann genutzt werden, um zu verhindern, dass in einem Text Block automatisch Zeilenumbrüche am Zeilenende eingefügt werden. Letztere hilft dabei, Leerzeichen am Ende eines Text Blocks erhalten zu können. Schauen wir uns für beides Beispiele an.

Escape-Sequenz `\` Manchmal trennt man Strings in mehrere kleinere Bestandteile auf, sodass man den längeren String einfach besser im Editor lesen kann:

```
String literal = "This is a string splitted " +
                 "in several smaller " +
                 "strings.";
```

Würde man dies in einen Text Block konvertieren, so würde automatisch an jedem Zeilenende ein Zeilenumbruch eingefügt. Das kann man durch Angabe von `\` verhindern und folgenden Text Block nutzen:

```
String text = """
              This is a string splitted \
              in several smaller \
              strings.\
              """;
```

Damit wird folgender einzeiliger String erzeugt:

```
This is a string splitted in several smaller strings.
```

Escape-Sequenz `\s` Die zweite Escape-Sequenz `\s` wird in ein Leerzeichen umgewandelt und verhindert, dass Leerzeichen in einem Text Block am Ende abgeschnitten werden. Das Standardverhalten entfernt überschüssige Leerzeichen am Ende eines Strings und fügt direkt den Zeilenumbruch hinzu. Mit der neuen Escape-Sequenz kann man beispielsweise erreichen, dass alle Zeichenketten die gleiche Länge besitzen. Nehmen wir dazu an, wir wollten die Namen alle gleich lang begrenzen:

```
System.out.println("""
       Tim  \s
       Peter\s
       Mike \s""");
```

Das produziert folgende Ausgabe mit drei jeweils 6 Zeichen langen Strings (ohne `\s` wären die Strings 3, 5 und 4 Zeichen lang):

```
Tim
Peter
Mike
```

Allerdings wäre es wohl besser, wenn die Escape-Sequenz `\s` in kein Zeichen transformiert werden würde. Derzeit enthalten alle Angaben ein künstliches Leerzeichen am Ende.

Möchte man die Zeichen auf die Länge des längsten Strings begrenzen, so kann man dies wie folgt erreichen, indem man beim längsten String die Escape-Sequenz nicht angibt:

```
String names2 = """
Tim   \s
Michael
Mike  \s""";
System.out.println(names2);
```

Dadurch verändert sich die Ausgabe so, dass alle Texte die Länge des längsten Namens haben.

14.1.2 Switch Expressions

Java verharrte bislang beim `switch` bei den uralten Wurzeln aus der Anfangszeit der Programmiersprache und den Kompromissen im Sprachdesign, die C++-Entwicklern den Umstieg erleichtern sollten. Relikte, wie das `break` und bei dessen Fehlen das Fall Through[4], waren wenig intuitiv und luden zu Fehlern ein. Zudem war man beim `case` recht eingeschränkt bei der Notation der Werte.

Das alles hat sich glücklicherweise mit Java 12 und 13 in Form eines Previews auf eine neue, leicht modifizierte Syntax für `switch` geändert, die nun mit Java 14 final ist. Damit lassen sich Fallunterscheidungen deutlich eleganter als zuvor formulieren.

Einführendes Beispiel

Die Switch Expressions sind in JEP 361 (`https://openjdk.java.net/jeps/361`) beschrieben. Hier greife ich auf das dortige Beispiel der Abbildung von Wochentagen auf deren textuelle Länge zurück, ergänze dieses aber an geeigneter Stelle.

Um den Bedarf der Syntaxanpassung besser nachvollziehen zu können, betrachten wir zunächst, wie man die oben genannte Abbildung vor Java 12 formuliert hätte. Danach sehen wir uns die Vorteile durch die neue Syntax von `switch` an.

[4] Zur Erinnerung: Enden die Anweisungen des `case`-Zweigs nicht mit einem `break`, so werden unabhängig von der Übereinstimmung der nachfolgenden Bedingungen beim folgenden `case` die dortigen Anweisungen ausgeführt.

Analyse: Was waren einige bisherige Schwachstellen beim `switch`?

Beginnen wir mit der Implementierung der Abbildung von Wochentagen auf deren textuelle Länge mit der bisherigen Syntax, die `switch` vor Java 12 bietet:

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters;
switch (day)
{
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numOfLetters = 6;
        break;
    case TUESDAY:
        numOfLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numOfLetters = 8;
        break;
    case WEDNESDAY:
        numOfLetters = 9;
        break;
    default:
        numOfLetters = -1;
}
```

Betrachten wir den Sourcecode kritisch. Zunächst einmal wirkt das gezeigte Konstrukt nicht wirklich elegant und ist zudem ziemlich lang. Auch die Mehrfachangabe von Werten ist gewöhnungsbedürftig. Schlimmer noch: Es wird ein `break` benötigt, damit die Verarbeitung überraschungsfrei abläuft und es zu keinem Fall Through kommt. Zudem müssen wir die (künstliche) Hilfsvariable `numOfLetters` in jedem Zweig korrekt setzen. Insbesondere ist trotz der eigentlich vollständigen Abdeckung der Enum-Werte das `default` nötig, da sonst der Compiler moniert, dass die Variable `numOfLetters` möglicherweise nicht initialisiert ist – außer man hat dieser initial schon einen Wert zugewiesen. Wie geht es also besser?

Syntaxerweiterungen in Java 14

Mit den nun in Java 14 finalen »Switch Expressions« wird die Formulierung von Fallunterscheidungen deutlich erleichtert und bietet eine intuitive Schreibweise:

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

An diesem Beispiel erkennen wir einige syntaktische Neuerungen: Neben dem offensichtlichen Pfeil statt des Doppelpunkts können nun auch mehrere Werte hinter dem `case` angegeben werden. Praktischerweise wird kein `break` mehr benötigt: Die hinter dem Pfeil notierten Anweisungen werden jeweils nur spezifisch für das `case` ausgeführt und es existiert bei dieser Syntax kein Fall Through. Außerdem kann das `switch` nun einen Wert zurückgeben, wodurch sich die Definition von Hilfsvariablen vermeiden lässt. Statt nach dem Pfeil lediglich einen Wert anzugeben, ist hier auch problemlos die Angabe von Ausdrücken wie Zuweisungen oder Methodenaufrufen möglich, selbstverständlich nach wie vor ohne die Notwendigkeit eines `break`s: Mehr noch, dies ist in der neuen Syntax nach dem Pfeil nicht mehr erlaubt.

Wie bisher gilt: Übergibt man `null` an ein `switch`, so löst dies eine `NullPointerException` aus.

Besonderheit – Vollständigkeitsprüfung

Bei der alten Variante von `switch` konnte man das `default` weglassen oder auch die Angabe eines `case` für einzelne Werte, beispielsweise `WEDNESDAY`. Der Compiler hätte das im `switch` selbst nicht bemängelt und erst später beim Zugriff auf die im Beispiel genutzte Variable darauf hingewiesen, dass diese nicht in jedem Fall initialisiert ist.

Das hat sich praktischerweise mit den neuen Switch Expressions verbessert: Nehmen wir an, wir hätten im obigen Beispiel keine Angabe von `WEDNESDAY` gemacht. Dann wird dies direkt vom Compiler im `switch` bemängelt: »`A Switch expression should cover all possible values.`« Die IDEs bieten dann die Wahl, entweder das passende `case` oder ein `default` hinzuzufügen, aber nicht beides: Es wird automatisch die vollständige Abdeckung der Enum-Werte erkannt. Diese Neuerung ist deshalb nötig, weil das `switch` in jedem Anwendungsfall einen Wert zurückgeben muss.

Während es für `enum`s noch recht klar ist, stellt sich die Frage, wie geht das für andere Typen, etwa `int`s? Hier kann der Compiler nur feststellen, dass möglicherweise. nicht alle Werte abgedeckt sind, und bemängelt: »`A Switch expression should cover all possible values.`« Dann schlagen die IDEs vor, ein `default` hinzuzufügen – hier durch Auskommentieren angedeutet:

```
String numericString = switch (value)
{
    case 1 -> "one";
    case 2 -> "two";
    case 3 -> "three";
    // default -> "N/A";
};
```

Weitere Gründe für die Neuerung

Auf die Vollständigkeitsprüfung der Überdeckung der Werte in den `case`s möchte ich nochmals etwas genauer eingehen.

Fallstrick 1: Unvollständige Wertangaben Nachfolgend wollen wir einen Wert aus der Aufzählung `java.time.Month` auf den korrespondierenden Monatsnamen abbilden. Das kann man konventionell in etwa wie folgt lösen:[5]

```
Month month = Month.JULY;

String monthAsString;
switch (month)
{
    case JANUARY:
          monthAsString = "January";
          break;
    case FEBRUARY:
          monthAsString = "February";
          break;
    case MARCH:
          monthAsString = "March";
          break;
     //...
     default:
          monthAsString = "N/A";
}
```

Erneut ist das gezeigte Konstrukt nicht wirklich elegant. Je nachdem, ob für den Wert `Month.JULY` ein `case` definiert ist oder nicht, erhält man den Wert »July« oder »N/A«. Außerdem wird ein `break` benötigt, damit die Verarbeitung überraschungsfrei abläuft.

Fallstrick 2: `default` zwischen den `breaks` Betrachten wir etwas ziemlich Ungeschicktes, nämlich das `default` zwischen den `case`s und ohne ein `break`:

```
// ACHTUNG: Mitunter ganz übler Fehler: default mitten zwischen den cases
Month month = Month.JULY;

String monthAsString;
switch (month)
{
    case JANUARY:
        monthAsString = "January";
        break;
    default:
         monthAsString = "N/A";  // hier auch noch Fall Through
    case FEBRUARY:
         monthAsString = "February";
         break;
    case MARCH:
         monthAsString = "March";
         break;
}
```

[5] Mit dem Date and Time API aus Java 8 ließe sich das komplizierte Konstrukt natürlich deutlich vereinfachen: `month.getDisplayName(TextStyle.FULL, Locale.UK)`.

Die Eingabe `Month.FEBRUARY` führt erwartungskonform zum Wert »`February`«. So weit, so gut: Doch diesen Wert erhält man unerwartet auch für die Eingabe `Month.JULY`. Wie kommt das? Zunächst wird aufgrund des in den `case`s nicht aufgeführten Werts `JULY` der Defaultzweig angesprungen und wegen des fehlenden `break`s auch noch (unerwartet) ein Fall Through ausgelöst. Dadurch wird der Code für `case FEBRUARY` ausgeführt, was schon ziemlich verwirrend sein kann.

Abhilfe durch Umsetzung mit Java 14 Mithilfe der neuen Syntax könnte man das Ganze wie folgt schreiben:

```
static String monthToName(final Month month)
{
    return switch (month)
    {
        case JANUARY -> "January";
        default -> "N/A";  // hier KEIN Fall Through
        case FEBRUARY -> "February";
        case MARCH -> "March";
    };
}
```

Besonders erwähnenswert ist, dass man direkt den vom `switch`-Konstrukt berechneten Wert zurückgeben kann. Außerdem kommt es nicht zum Fall Through. Dadurch liefert die Eingabe `Month.FEBRUARY` erwartungskonform »`February`«, und zudem ist das `default` inmitten der `case`s nicht ganz so dramatisch, allerdings sicher auch nicht schön. Anders als bei der alten Syntax kommt es durch die Eingabe `Month.JULY` eben nicht mehr zu einer unerwarteten Ausgabe »`February`«, sondern wie durch `default` spezifiziert zu »`N/A`«. Darüber hinaus gilt: Gäbe es ein `case JULY`, so würde dieses immer angesprungen werden, unabhängig von der Position des `default`, also analog zum Verhalten der bisherigen `switch`-Anweisung.

Rückwärtskompatible Angabe bei `case` mit `yield`

Beim Umstieg auf die neue Syntax für `switch` könnte es eher zögerliche Entwickler geben, die sich den Veränderungen in der Syntax gerne schrittweise nähern möchten. Dann kann man die Angabe bei `case` recht ähnlich gestalten und statt einer Wertzuweisung `yield` mit Wert nutzen.

Unelegante alte Syntax Kommen wir nochmals auf die bisherige Syntax zurück und bilden Namen von Farben aus der folgenden Aufzählung

```
enum Color { RED, GREEN, BLUE, YELLOW, ORANGE }
```

auf deren Anzahl an Buchstaben ab – hier bewusst mit einem kleinen Flüchtigkeitsfehler zur Demonstration von Fall Through versehen:

```
Color color = Color.GREEN;
int numOfChars;

switch (color)
{
    case RED: numOfChars = 3; break;
    case BLUE: numOfChars = 4; break;
    case GREEN: numOfChars = 5; /* break; UPS: FALL THROUGH */
    case YELLOW:
    case ORANGE: numOfChars = 6; break;
    default: numOfChars = -1;
}

System.out.println("Color " + color.name() + " chars: " + numOfChars);
```

Es kommt durch das Fall Through zur (überraschenden) Ausgabe von

```
Color GREEN chars: 6
```

Dieses Beispiel zeigt erneut die Nachteile der alten Syntax, die doch etwas sperrig in der Handhabung ist und ohne explizite Kennzeichnung das Fall Through kaum ersichtlich macht. Da ist der Zwang zur Definition der künstlichen Hilfsvariablen eher nur ein Schönheitsmakel. Zudem können wir leider `default` nicht weglassen, da der Compiler die Vollständigkeit hier nicht erkennt. Schauen wir uns erneut an, wie es besser geht.

Abhilfe durch Umsetzung mit Java 14 Die nachfolgend gezeigte Variante nutzt statt `break` das Schlüsselwort `yield` und einen Rückgabewert. Das ist recht nah an der alten Syntax, jedoch mit dedizierten Verbesserungen. Insgesamt ergibt sich damit folgende besser lesbare Schreibweise:

```
int numOfChars = switch (color)
{
    case RED: yield 3;
    case BLUE: yield 4;
    case GREEN: yield 5;
    case YELLOW, ORANGE: yield 6;
};

System.out.println("Color " + color.name() + " chars: " + numOfChars);
```

Nun kommt es zu der erwarteten (und korrekten) Ausgabe von

```
Color GREEN chars: 5
```

In diesem Beispiel werden die Vorteile der neuen Syntax deutlich. Zum einen benötigt man keine künstliche Hilfsvariable mehr, sondern kann mit `yield` einen Wert zurückgeben. Dadurch wird die Abarbeitung dort gestoppt, also ähnlich zu dem `return` einer Methode. Zum anderen lassen sich nun mehrere Werte in einem `case` angeben, wodurch eine kompaktere Schreibweise erreicht wird. Und wiederum ist kein `default` nötig, da alle Werte des Enums durch `case`-Anweisungen abgedeckt sind.

Rückgabe mit `yield`

Neben dem gerade beschriebenen Anwendungsfall gibt es eine weitere syntaktische Besonderheit, für die `yield` benötigt wird.

Kommen wir dazu auf die Abbildung von Wochentagen auf die Länge ihrer Namen zurück. Einleitend hatte ich erwähnt, dass nach einem Pfeil Methodenaufrufe oder mehrere Anweisungen folgen können, doch wie sollen diese einen Wert zurückgeben? Dazu bedarf es des Einsatzes von `yield` wie folgt:

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> {
        if (day == DayOfWeek.SUNDAY)
            System.out.println("SUNDAY is FUN DAY");

        yield 6;
    }
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
System.out.println(numOfLetters);
```

Merkwürdigkeit: Wechselnder Typ

Ich möchte Sie noch auf eine Besonderheit hinweisen, nämlich den flexiblen Typ. Was meine ich damit?

Sie kennen sicher den ?-Operator, aber sind Sie auch mit seiner (tückischen) Besonderheit vertraut? In der Regel arbeitet der ?-Operator wie ein `if`. Allerdings gibt es dabei bezüglich des Rückgabetyps Unterschiede. Es ist nämlich möglich, in den beiden Alternativen nicht kompatible Typen als Rückgabe zu verwenden, hier `String` und `int`:

```
var result = value == 13 ? "Pleiten, Pech und Pannen" : 42;
```

Je nach Wertebelegung von `value` erhält man einen `String` oder einen `int`. Das ist nun bei Switch Expressions auch möglich:

```
var result = switch (value)
{
    case 13 -> "Pleiten, Pech und Pannen";
    default -> 42;
};
```

14.1.3 Records (Preview)

In Java 14 und 15 werden als Preview sogenannte Records unterstützt. Dabei handelt es sich um eine simplifizierte Form von Klassen, deren API sich implizit aus den als Konstruktorparametern definierten Attributen ergibt. Ein Record stellt damit einen einfachen Datencontainer dar und ist somit eine Werteklasse, die nur genau einen unveränderlichen Zustand modelliert und die dazu notwendigen Implementierungen von rein lesenden Accessor-Methoden sowie `equals()` und `hashCode()` automatisch und vor allem kontraktkonform erzeugt sowie einen Konstruktor und `toString()`. Allerdings ist Unveränderlichkeit nur dann garantiert, wenn man für die Konstruktorparameter lediglich primitive Datentypen nutzt oder nur unveränderliche Typen angegeben werden.

Als weitere Analogie kann man noch das mathematische Konzept der Tupel hinzuziehen, also eine Zusammenfassung mehrerer Werte potenziell unterschiedlichen Typs.

Insgesamt lassen sich Records in diversen Anwendungsfällen gewinnbringend einsetzen. Beispiele sind etwa Methoden, die mehrere Werte zurückgeben sollen, Compound Keys für Maps, DTOs, Parameter Value Objects usw. Schauen wir uns zunächst ein einführendes Beispiel und danach weiter gehende Möglichkeiten an.

Einführendes Beispiel

Die einführenden Erklärungen mögen noch etwas kompliziert klingen, das Ganze wird aber nach einem Blick auf den folgenden Sourcecode klarer:

```
record MyPoint(int x, int y) { }
```

Betrachten wir einmal, wie viel Sourcecode man schreiben müsste, um eine äquivalente Funktionalität mit den bisherigen Java-Sprachmitteln zu erreichen:

```
public final class MyPoint
{
    private final int x;
    private final int y;

    public MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o)
    {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        MyPoint point = (MyPoint) o;
        return x == point.x && y == point.y;
    }
```

```
    @Override
    public int hashCode()
    {
        return Objects.hash(x, y);
    }

    @Override public String toString()
    {
        return "MyPoint[x=" + x + ", y=" + y + "]";
    }

    // lesende Zugriffsmethoden auf x und y
}
```

Recht interessant ist auch, einmal mit dem Tool `javap` den generierten Bytecode in Form einer Übersicht anzuschauen:

```
Michaels-iMac:java14 michaeli$ javap MyPoint
Compiled from "MyPoint.java"
final class java14.MyPoint extends java.lang.Record {
  public java14.MyPoint(int, int);
  public int x();
  public int y();
  public java.lang.String toString();
  public int hashCode();
  public boolean equals(java.lang.Object);
}
```

Man erkennt, dass die Records den Basistyp `java.lang.Record` besitzen und für jedes Konstruktorargument eine passende Zugriffsmethode erstellt wird. Deren Namensgebung folgt jedoch nicht dem JavaBeans-Standard und besitzt kein Präfix `get`, sondern besteht lediglich aus dem Attributnamen.

Zugriff auf Attribute – implizit erzeugte Methoden

Nachdem wir einführend den Record `MyPoint` kennengelernt haben, könnten wir uns noch fragen, wie der Zugriff auf die Attribute möglich ist. Tatsächlich wird für jeden Konstruktorparameter ein Attribut sowie eine zugehörige lesende Zugriffsmethode erzeugt:

```
public class MyPointClient
{
    public static void main(final String[] args)
    {
        final MyPoint myPoint = new MyPoint(47, 11);

        System.out.println(myPoint);
        System.out.println(myPoint.x());
        // System.out.println(myPoint.x);   // nicht zugreifbar
    }
}
```

Diese Zeilen führen zu folgender Ausgabe:

```
MyPoint[x=47, y=11]
47
```

Records für DTOs und Parameter Value Objects

Beim Austausch von Daten hat sich der Einsatz von DTOs (Data Transfer Objects) bewährt. Das sind (oftmals unveränderliche) Datencontainerklassen, die zum Transport von Daten an Schnittstellen oder Komponentengrenzen eingesetzt werden. Allerdings erfordert deren Definition ohne Records einiges an Schreibarbeit – schlimmer noch, mitunter erfolgt die Implementierung einer der Methoden `equals()` und `hashCode()` falsch oder gar nicht. Gerade haben wir schon ein Positivbeispiel in Form des Records `MyPoint` kennengelernt.

Ähnlich zu DTOs, aber mit einer anderen Intention, gibt es sogenannte Parameter Value Objects, die einige Methodenparameter zu einer neuen Einheit bündeln und dadurch Methodensignaturen vereinfachen können.

Betrachten wir ein paar Beispiele:

```
record Point(int x, int y) { }
record TopLeftWidthAndHeight(Point topLeft, int width, int height) { }

record ColorAndRgbDTO(String name, int red, int green, int blue) { }
record PersonDTO(String firstname, String lastname, LocalDate birthday) { }
```

Mit den ersten beiden Records könnte man dann ein Rechteck statt mit vier `int`-Werten mit sprechenden Abmessungen wie folgt definieren:

```
var oldStyle = new Rectangle(10, 10, 70, 20);

var topLeft = new Point(10, 10);
var newStyle = new Rectangle(new TopLeftWidthAndHeight(topLeft, 70, 20));
```

Records für komplexere Rückgabewerte und Compound Keys

In Java ist es bisher ziemlich umständlich, eine Methode mehrere Werte zurückgeben zu lassen, etwa einen Fehlercode samt Mitteilungstext. Das wird mit Records zu einem Kinderspiel. Darüber hinaus kann man Records auch zur Definition von sogenannten Compound Keys nutzen. Das sind Klassen, die mehrere Werte bündeln, die als Einheit beim Zugriff auf eine Map als Schlüssel verwendet werden sollen.

Mithilfe von Records ist auch die Definition leicht komplexerer Datenstrukturen denkbar, beispielsweise wie folgt:

```
record IntStringReturnValue(int code, String info) { }

record IntListReturnValue(int code, List<String> values) { }

record ReturnTuple(String first, String last, int amount) { }

record CompoundKey(String name, int age) { }
```

Erweiterungsmöglichkeiten – Definition eigener Methoden

Bisher sind Records sehr intuitiv und es bleibt auch so – selbst für den Fall, dass wir dort eigene Methoden ergänzen möchten. Nehmen wir als Beispiel einen Record `PersonDTO` und den Wunsch, dort die Ausgabe des vollen Namens aus der Konkatenation von Vor- und Nachname in Form einer Methode `asFullname()` zu realisieren:

```
record PersonDTO(String firstname, String lastname, LocalDate birthday)
{
    public String asFullname()
    {
        return firstname + " " + lastname;
    }
}
```

Schauen wir uns dies einmal für die obige Definition an:

```
final PersonDTO micha = new PersonDTO("Michael", "Inden",
                                      LocalDate.of(1971, 2, 7));
System.out.println(micha);
System.out.println(micha.asFullname());
```

Diese Zeilen führen zu folgender Ausgabe:

```
PersonDTO[firstname=Michael, lastname=Inden, birthday=1971-02-07]
Michael Inden
```

Sofern in seltenen Fällen wirklich nötig, lassen sich sogar die automatisch kontraktkonform erzeugten drei Methoden `equals()`, `hashCode()` und `toString()` auch überschreiben, etwa für das initiale Beispiel wie folgt – dabei sehen wir noch eine Besonderheit, weil wir im Record selbst die Attribute natürlich direkt per Namen ansprechen können:

```
record MyPoint(int x, int y)
{
    @Override
    public String toString()
    {
        return String.format("(x: %d,y: %d)", x, y);
    }
}
```

Erweiterungsmöglichkeiten – Definition eigener Konstruktoren

Es lassen sich nicht nur eigene Methoden, sondern auch Konstruktoren definieren. Nehmen wir an, wir wollten unseren Record `MyPoint` auch aus einer textuellen Repräsentation im Format `x,y` erzeugen. Dann schreiben wir folgenden Konstruktor:

```
record MyPoint(int x, int y)
{
    public MyPoint(String values)
    {
        this(Integer.parseInt(values.split(",")[0]),
             Integer.parseInt(values.split(",")[1]));
    }
}
```

Das erlaubt dann folgende Konstruktion:

```
MyPoint myPoint = new MyPoint("72,71");
```

Spezialbehandlungen – Gültigkeitsprüfungen der Parameter

Beim Definieren eines Records ist es manchmal wichtig, gewisse Randbedingungen oder Wertebereiche für einige Parameter zu prüfen. Für solche Fälle kann man explizit einen Konstruktor in der Definition des Records angeben.

Betrachten wir das Ganze zur Definition von Wertebereichen eines geschlossenen Intervalls, wobei die untere Grenze immer kleiner oder gleich dem oberen Grenzwert sein muss. Setzen wir dies mit einem Record um:

```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval(int lower, int upper)
    {
        if (lower > upper)
        {
            String errorMsg = String.format("invalid: %d (lower) > %d (upper)",
                                            lower, upper);
            throw new IllegalArgumentException(errorMsg);
        }

        this.lower = lower;
        this.upper = upper;
    }
}
```

Interessanterweise gibt es eine syntaktische Besonderheit: Man kann anstelle eines Konstruktors auch folgendes Konstrukt verwenden – dadurch erfolgen die gleichen Prüfungen, aber man muss keine Zuweisungen der Variablen von Hand vornehmen:

```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval
    {
        if (lower > upper)
        {
            String errorMsg = String.format("invalid: %d (lower) > %d (upper)",
                                            lower, upper);
            throw new IllegalArgumentException(errorMsg);
        }
    }
}
```

Besonderheit – Generics in Records

Nicht nur für Interfaces und Klassen ist es mitunter wünschenswert, diese generisch für verschiedene Typen anbieten zu können. Tatsächlich lassen sich auch typisierte Records erstellen.

Das wollen wir am Beispiel von `Pair` für Wertepaare sowie von `MultiTypes` mit einer Map und einem weiteren generischen Parameter betrachten. Schließlich kombinieren wir die Generics bei Records mit der gerade kennengelernten Gültigkeitsprüfung und erstellen einen Record namens `ListRestrictions`, der die hereingereichte Liste auf eine ebenfalls als Parameter übergebene Maximalgröße testet und bei Überschreiten die Konstruktion verweigert:

```
record Pair<T1, T2> (T1 first, T2 second)
{}

record MultiTypes<K,V,T> (Map<K,V> mapping, T info)
{}

record ListRestrictions<T> (List<T> values, int maxSize)
{
    public ListRestrictions
    {
        if (values.size() > maxSize)
            throw new IllegalArgumentException("too many entries! got: "
                      + values.size() + ", but restricted to " + maxSize);
    }
}
```

Das Ganze wollen wir jetzt in einer `main()`-Methode in Aktion erleben. Dort rufen wir drei Methoden auf. In `pairExamples()` definieren wir einige Paare mit unterschiedlichen Typen, geben diese auf der Konsole aus und prüfen, ob Paare mit identischen Werten auch mit `equals()` als gleich angesehen werden. In `multiTypesExamples()` erzeugen wir einen Record mit einer Map und einem String als Eingabe und ermitteln die jeweiligen Typen. Zum Abschluss definieren wir in `listRestrictionsExamples()` zwei Listen und prüfen, ob deren Längenbeschränkung eingehalten wird. Diese Aktionen implementieren wir wie folgt:

```
public static void main(final String[] args)
{
    // Paare mit teilweise verschiedene Typisierungen
    pairExamples();

    //
    multiTypesExamples();

    // Prüfe die Restriktion auf n Elemente (hier 5)
    listRestrictionsExamples();
}

protected static void pairExamples()
{
    var pair1 = new Pair<>("INFO", 4711);
    var pair1b = new Pair<>("INFO", 4711);
    var pair2 = new Pair<>("FIRST", "SECOND");
    var pair3 = new Pair<>("BLUE", Color.BLUE);
    System.out.println(pair1);
    System.out.println(pair2);
    System.out.println(pair3);
    System.out.println(pair1.equals(pair1b));
    System.out.println();
}

protected static void multiTypesExamples()
{
    var mapping = Map.of("tim", 42, "tom", 77, "mike", 49);
    var multiGenericTypes = new MultiTypes<>(mapping, "INFO");
    System.out.println(multiGenericTypes);
    System.out.println("mapping type: " + multiGenericTypes.mapping.getClass());
    System.out.println("info type: " + multiGenericTypes.info.getClass());
    System.out.println();
}

protected static void listRestrictionsExamples()
{
    var timAndMike = List.of("Tim", "Mike");
    var initialRestricted = new ListRestrictions<>(timAndMike, 5);
    System.out.println("values: " + initialRestricted.values());

    var names = List.of("Tim", "Tom", "Peter", "Franz", "Fritz", "Mike");
    var restrictionViolated = new ListRestrictions<>(names, 5);
}
```

Führen wir die obige `main()`-Methode aus, so erhalten wir die folgenden, erwartungskonformen Ausgaben:

```
Pair[first=INFO, second=4711]
Pair[first=FIRST, second=SECOND]
Pair[first=BLUE, second=java.awt.Color[r=0,g=0,b=255]]
true

MultiTypes[mapping={mike=49, tom=77, tim=42}, info=INFO]
mapping type: class java.util.ImmutableCollections$MapN
info type: class java.lang.String

values: [Tim, Mike]
Exception in thread "main" java.lang.IllegalArgumentException: too many entries!
       got: 6, but restricted to 5
```

Zusammenfassung

Weil Records ein tolles neues Sprachfeature mit vielen Besonderheiten darstellen, möchte ich noch einmal ein paar Dinge rekapitulieren oder erwähnen:

- Records besitzen den Basistyp `java.lang.Record` und sind implizit `final`, sodass man davon keine eigenen Klassen ableiten kann.
- Records können Interfaces implementieren, was sehr praktisch im Kontext von Vererbungshierarchien und Polymorphie ist.
- Die Attribute werden aus den Konstruktorparametern hergeleitet und man kann selbst keine weiteren Attribute definieren, statische Attribute sind dagegen möglich.
- Die Attribute dürfen nicht `hashCode` und `toString` heißen, ansonsten kommt es zu Kompilierfehlern (`Illegal component name hashCode in record`). Zwar ist `equals` als Name syntaktisch erlaubt, jedoch wenig sinnvoll.
- Da die Attribute direkt im Konstruktor belegt werden, ist ein Initializer-Block nicht erlaubt (`instance initializers not allowed in records`).
- Die automatisch erzeugten Zugriffsmethoden weichen vom JavaBeans-Namensschema ab, statt `getXyz()` kommt einfach `xyz()` zum Einsatz.
- Die Methoden `equals()`, `hashCode()` und `toString()` können überschrieben werden – bitte mit der gebotenen Vorsicht bei `equals()` und `hashCode()`. Dabei ist ein Aufruf der Basisimplementierung verboten (`abstract method hashCode() in Record cannot be accessed directly`).

Tipp: Implikationen für Reflection

Um die Eigenschaften zu Records abfragen zu können, bietet das Reflection-API folgende neue Methoden in `java.lang.Class<T>`:

```
public boolean isRecord()
public RecordComponent[] getRecordComponents()
```

14.1.4 Pattern Matching bei `instanceof` (Preview)

Bei `instanceof` gibt es in Form von Pattern Matching eine hilfreiche Erweiterung, die es erlaubt, den Sourcecode klarer zu halten und weniger Boilerplate nutzen zu müssen. Was meine ich damit? Gerade im Kontext von `instanceof` sieht man immer wieder nach der Typprüfung eine Definition einer Hilfsvariablen inklusive eines Casts etwa wie folgt:

```
if (obj instanceof Person)
{
    final Person person = (Person) obj;
    // ... Zugriff auf person...
}
```

Dieses verbreitete Idiom (Muster auf Sourcecode-Ebene) wird verwendet, um mit der Variablen sinnvoll und typspezifisch weiterarbeiten zu können. Vor allem bei mehreren Prüfungen auf verschiedene Typen kann das jedoch eher unübersichtlich werden und ist auch oft nicht schön zu lesen.

Seit Java 14 kann man das Ganze folgendermaßen präziser und kürzer schreiben, indem man nach dem Typ bei `instanceof` noch eine sogenannte Binding-Variable angibt, auf die man dann im Block darunter typisierten Zugriff hat:

```
if (obj instanceof Person person)
{
    // Hier kann man auf die Variable person direkt zugreifen
}
```

Betrachten wir zum besseren Verständnis ein konkretes Beispiel:

```
Object obj = "Hallo Java 14";

if (obj instanceof String str)
{
    // Hier kann man str nutzen
    System.out.println("Länge: " + str.length());
}
else
{
    // Hier kein Zugriff auf str
    System.out.println(obj.getClass());
}
```

Praktische Besonderheit

Obwohl das gerade Gezeigte schon recht nett ist, geht es noch ein wenig besser. Man kann nämlich auf die Variable bereits im `if`-Ausdruck zugreifen:

```
if (obj instanceof String str2 && str2.length() > 5)
{
    System.out.println("Länge: " + str2.length());
}
```

Fallstrick

Allerdings sollten wir auch auf einen möglichen Fallstrick eingehen: Man darf die definierte Variable nicht genauso benennen wie die getestete, d. h., Folgendes ist nicht erlaubt:

```
// Achtung: Compile-Error: Duplicate local variable obj
if (obj instanceof String obj && obj.length() > 5)
{
    System.out.println("Länge: " + obj.length());
}
```

Wie im Listing angedeutet, kommt es sonst zum Kompilierfehler: »`Duplicate local variable obj`«.

14.1.5 Sealed Types (Preview)

Ebenfalls als Preview-Neuerung finden wir sogenannte **Sealed Types**. Diese ermöglichen es, Vererbung zu steuern und zu spezifizieren, welche Klassen eine Basisklasse erweitern können. Damit existiert eine restriktive Sichtbarkeitssteuerung, die jedoch vor allem in speziellen Kontexten sinnvoll einsetzbar ist (generell sollten Basisklassen keine Verweise auf Subklassen besitzen). Um was geht es genau?

Teilweise möchte man eine Vererbungshierarchie aus Interfaces und / oder abstrakten Basisklassen mit einer vordefinierten Menge an Klassen implementieren. Wird der Basistyp nun `public` definiert, um auch für Nutzer aus anderen Packages zugreif- und verwendbar zu sein, dann ist es potenziell möglich, dass Spezialisierungen des Basistyps entstehen, die man zum Kompilierzeitpunkt noch gar nicht im Sinn hatte. Genau diese durch Polymorphie und dynamisches Binden (vgl. Abschnitt 3.1.1) ermöglichte Flexibilität stellt eine Stärke der objektorientierten Programmierung dar.

Warum sollte man das Einschränken wollen? Eine gute Frage! Tatsächlich sind die Anwendungsfälle vermutlich eher selten. Einige davon sind im Bereich der abstrakten Datentypen zu finden. Ein weiteres Beispiel wäre eine Security-Prüfung, die über ein öffentliches Interface bereitgestellt wird. Nun möchte man mögliche Implementierungen auf die im Framework vorgegebenen konkreten Realisierungen beschränken. Oder stellen wir uns einen Taschenrechner vor, der nur die vier Grundrechenarten in Form von Klassen anbieten soll.

Mit `sealed` können wir nun eine Vererbungshierarchie versiegeln und nur die explizit angegebenen Typen erlauben. Deren Definition muss `sealed`, `non-sealed` oder `final` sein. Das hat folgende Konsequenzen:

- Eine als `sealed` markierte Klasse *muss* Subklassen besitzen, die wiederum hinter dem neuen Schlüsselwort `permits` aufgeführt werden müssen.
- Eine als `non-sealed` markierte Klasse kann als Basisklasse fungieren und von dieser können weitere Klassen abgeleitet werden.
- Eine als `final` markierte Klasse bildet den Endpunkt einer Ableitungshierarchie.

Schauen wir uns ein Beispiel an, das diese Kombinationen verdeutlicht und in den Kommentaren mögliche Restriktionen beschreibt:

```
// Eine sealed Klasse muss Subklassen besitzen
sealed interface MathOp permits BaseOp, Add, Sub
{
    int calc(int x, int y);
}

// direkte Implementierung muss final sein
final class Add implements MathOp
{
    @Override
    public int calc(int x, int y)
    {
        return x + y;
    }
}
```

```
final class Sub implements MathOp
{
    @Override
    public int calc(int x, int y)
    {
            return x - y;
    }
}

    // Mit non-sealed kann man innerhalb der Vererbungshierarchie
    // Basisklassen bereitstellen
    non-sealed class BaseOp implements MathOp
    {
        @Override
        public int calc(int x, int y)
        {
            return 0;
        }
    }

    // Ableitung aus Basisklasse muss final sein
    final class Mult extends BaseOp
    {
    }

    final class Div extends BaseOp
    {
    }

}
```

Dagegen sind aus den oben schon genannten Gründen folgende Konstrukte nicht möglich und produzieren die gezeigten Kompilierfehler:

```
// SpecialMathOp is not allowed in the sealed hierarchy
// final class SpecialMathOp implements MathOp
// {
//     ...
// }

// UnsealedMathOp is not allowed in the sealed hierarchy
// public non-sealed class UnsealedMathOp implements MathOp
// {
// }
```

Wissenswertes

Weil Sealed Types ein doch eher spezielles Konstrukt sind, möchte ich die wichtigsten Dinge noch einmal stichpunktartig zusammenfassen:

- Mit Sealed Types kann man mithilfe von `sealed` und `permits` festlegen, welche andere Klassen oder Interfaces davon Subtypen bilden dürfen.
- Sealed Types können bei der Entwicklung von Bibliotheken sinnvoll sein, die ihr Verhalten über Interfaces exponieren, aber Kontrolle über mögliche Implementierungen behalten wollen.

- Sealed Types ermöglichen intern über `non-sealed` die Definition von Basisklassen, wobei die konkreten Implementierungen der als `sealed` markierten Klassen dann schließlich alle `final` sein müssen.
- Sealed Types schränken bezüglich der Erweiterbarkeit von Klassenhierarchien ein und sollten daher mit Bedacht verwendet werden. In Abschnitt 20.5.3 zeige ich die Technik »Extract And Override«, die Funktionalität eines bestehenden Typs umdefiniert, um das Testen zu erleichtern. Solche bei der Implementierung nicht vorhergesehenen Flexibilitäten können nachträglich störend sein.

14.1.6 Lokale Enums und Interfaces (Preview)

Zum Abschluss der Syntaxneuerungen lernen wir noch lokale Enums und Interfaces kennen. Bis einschließlich Java 14 war es von der Syntax nicht erlaubt, innerhalb von Methoden Enums oder Interfaces zu definieren, sondern lediglich Klassen. Diese Restriktion wurde mit Java 15 aufgehoben. Allerdings dürfte es eher wenige Anwendungsfälle geben, für die das wirklich entscheidend ist. Bedenken Sie, dass Methoden kurz und präzise und gemäß dem SRP (vgl. Abschnitt 3.5.3) auch nur eine Funktionalität bereitstellen sollten – lokale Enums und Interfaces verleiten eher zum Gegenteil.

Nach dieser Vorbetrachtung lernen wir die Syntaxneuerung an folgendem Beispiel in der `main()`-Methode und dem lokalen Enum `LocalEnumState` sowie dem lokalen Interface `Evaluationable` mit einer realisierenden lokalen Klasse `AlwaysBad` kennen, die auch noch das lokale Enum nutzt.

```
public class LocalEnumsAndInterfacesExamples
{
    public static void main(final String[] args)
    {
        // Erst ab Java 15, vorher Compile-Error:
        // Local enums are not supported at language level '14'
        enum LocalEnumState
        {
            BAD, GOOD, UNKNOWN
        }

        // Erst ab Java 15, vorher Compile-Error:
        // Local interfaces are not supported at language level '14'
        interface Evaluationable
        {
            LocalEnumState evaluate(String info);
        }

        class AlwaysBad implements Evaluationable
        {
            @Override
            public LocalEnumState evaluate(String info)
            {
                return LocalEnumState.BAD;
            }
        }
        System.out.println(new AlwaysBad().evaluate("DOES NOT MATTER"));
    }
}
```

14.2 API-Neuerungen

Zunächst gehe ich auf die Ergänzungen in der Klasse `String` ein. Danach stelle ich eine Convenience-Funktionalität in der Utility-Klasse `Files` vor. Abschließend widmen wir uns dem sogenannten Teeing-Kollektor, einer Erweiterung im Bereich des Stream-APIs.

14.2.1 Neue Methoden in der Klasse `String`

Die Klasse `java.lang.String` wurde zuletzt in Java 11 erweitert und mit Java 12 nochmals leicht ergänzt.

Die Methode `indent()`

Für Strings war es mit JDK-Bordmittel bisher mühsam bzw. nur mithilfe von externen Bibliotheken möglich, Texte um einige Leerzeichen einzurücken oder eine solche Einrückung wieder zu entfernen. Dazu bietet Java 12 die Methode `indent()`.

Folgendes Beispiel demonstriert deren Arbeitsweise, indem der Text `Test` um vier Leerzeichen eingerückt wird. Danach zeige ich, wie sich die Angabe einer negativen Einrückung auswirkt:

```
private static void indentExample()
{
    var test = "Test".indent(4);
    printTextAndLength(test);

    var removeTest = "     6789".indent(-5);
    printTextAndLength(removeTest);
}

private static void printTextAndLength(final String str)
{
    System.out.println("'" + str + "'");
    System.out.println(str.length());
}
```

Diese Methode gibt Folgendes aus:

```
'    Test
'
9
'6789
'
5
```

Die Ausgabe der Länge erfolgt, um auf eine Merkwürdigkeit bzw. Besonderheit hinweisen zu können – nämlich das automatische Hinzufügen eines Zeilenumbruchs beim Aufruf von `indent()`. In meinen Augen ist das eine eher unerwartete und negative Verhaltensweise, die sich leider nicht deaktivieren lässt.

Besonderheit Zeilenumbruch Nachfolgend möchte ich den automatisch eingefügten Zeilenumbruch noch einmal thematisieren. Nehmen wir an, wir wollten eine Ausgabe erzeugen, die mehrere Einrückebenen besäße, etwa Level 0, 1, 2 usw. Dazu erstellen wir folgendes Beispiel:

```
private static void indentProblem()
{
    var header = "Level 0: Report";
    var infoMessage = "Level 1: This is a message".indent(4);
    var infoMessage2 = "Level 2: This is a line 2".indent(8);

    System.out.println(header);
    System.out.println(infoMessage);
    System.out.println(infoMessage2);
}
```

Diese Methode gibt Folgendes aus:

```
Level 0:Report
    Level 1: This is a message

        Level 2: This is a line 2
```

Man erkennt jetzt noch deutlicher, dass (leider) jeweils ein Zeilenumbruch eingefügt wird. Somit lässt sich eine Multi-Einrückung für unterschiedliche Ebenen nur schwierig bzw. aufwendig erreichen.

Eine gleichmäßige Einrückung kann man für mehrzeilige Strings problemlos folgendermaßen realisieren:

```
private static void indentHowItShouldBeUsed()
{
    var header = "Report";
    var infoMessage = "This is a message\nThis is line 2\nLine3".indent(4);

    System.out.println(header);
    System.out.println(infoMessage);
}
```

Diese Methode gibt Folgendes aus:

```
Report
    This is a message
    This is line 2
    Line3
```

Die Methode `transform()`

Strings sind in Java bekanntermaßen unveränderlich und Modifikationen erfordern oftmals diverse temporäre Stringobjekte. Mit Java 12 wurde die Methode `transform()` eingeführt, die ein wenig wie die Methode `map()` im Stream-API arbeitet und Modifikationen eines Strings ohne benutzerdefinierte temporäre Hilfsobjekte ermöglicht.

Nehmen wir an, wir wollten zunächst alle Zeichen eines Strings in Großbuchstaben wandeln, dann alle Zeichen '`T`' entfernen und schließlich den String in einzelne Bestandteile aufspalten. Bisher realisiert man das beispielsweise in mehreren Schritten mit jeweils neu entstehenden temporären Stringobjekten für die Zwischenschritte wie folgt:

```
private static void transformationOldStyle()
{
    var text = "This is a Test";

    var upperCase = text.toUpperCase();
    var noTs = upperCase.replaceAll("T", "");
    var result =  noTs.split(" ");

    System.out.println(Arrays.toString(result));
}
```

Mit Java 12 kann man diese Umwandlung etwas eleganter durch die Hintereinanderschaltung von `transform()` schreiben:

```
private static void transformationJdk12Style()
{
    var text = "This is a Test";

    // Hintereinanderschaltung von Operationen
    var result = text.transform(String::toUpperCase).
                      transform(str -> str.replaceAll("T", "")).
                      transform(str -> str.split(" "));

    System.out.println(Arrays.toString(result));
}
```

Beide Methoden geben Folgendes aus:

```
[HIS, IS, A, ES]
```

14.2.2 Neue Hilfsmethode in der Utility-Klasse `Files`

Wie in Abschnitt 5.5.6 beschrieben, hat die Utility-Klasse `java.nio.file.Files` in Java 11 ein paar Erweiterungen in Form der Methoden `writeString()` und `readString()` erhalten. Mit Java 12 wurde zum Vergleich von Dateien die Methode `mismatch()` ergänzt. Diese liefert die Position der ersten Abweichung zweier über ein `Path`-Objekt referenzierter Dateien. Wenn die Dateien die gleiche Länge besitzen sowie inhaltlich (auf Byte-Ebene) übereinstimmen, liefert `mismatch()` den Wert -1 für Gleichheit.

Beispiel

Betrachten wir eine kleine Beispielapplikation zum besseren Verständnis: Nachfolgend wollen wir mit Funktionalitäten aus Java 11 textuelle Informationen in temporäre Dateien schreiben und deren Inhalte dann mithilfe von `mismatch()` vergleichen:

```
private static void compareSameContent() throws IOException
{
    Path filePath1 = Files.createTempFile("test1", ".txt");
    Path filePath2 = Files.createTempFile("test2", ".txt");

    Files.writeString(filePath1, "Same Content");
    Files.writeString(filePath2, "Same Content");

    long pos = Files.mismatch(filePath1, filePath2);
    System.out.println("Same content: mismatch(File1, File2) = " + pos);
}

private static void compareDifferentContent() throws IOException
{
    Path filePath1 = Files.createTempFile("test1", ".txt");
    Path filePath2 = Files.createTempFile("test2", ".txt");

    Files.writeString(filePath1, "Same Content");
    Files.writeString(filePath2, "Same Start / Different Content");

    long pos = Files.mismatch(filePath1, filePath2);
    System.out.println("Different content: mismatch(File1, File2) = " + pos);
}
```

Listing 14.1 *Ausführbar als* '**FILESMISMATCHEXAMPLE**'

Beim Ausführen der obigen Programmzeilen als FILESMISMATCHEXAMPLE kommt es zu folgenden Ausgaben:

```
Same content: mismatch(File1, File2) = -1
Different content: mismatch(File1, File2)  = 5
```

Erwartungskonform liefert `mismatch()` bei gleichem Zeichensatz (Encoding) den Wert `-1` für Gleichheit bzw. die Position der ersten Abweichung im zweiten Fall.

Einfluss des Zeichensatzes Als Besonderheit möchte ich auf die Auswirkungen unterschiedlicher Zeichensätze (Charsets) eingehen. Dazu erzeugen wir zwei vom textuellen Inhalt zwar gleiche Dateien, die aber in ihrem Zeichensatz und somit auch in ihrer Byte-Repräsentation voneinander abweichen:

```
private static void compareDifferentEncodings() throws IOException
{
    var fileLatin = Files.createTempFile("enc1", ".latin");
    var fileUtf8 = Files.createTempFile("enc2", ".utf8");

    var msg = "Zürich is beautiful. Mainz too";
    Files.writeString(fileLatin, msg, StandardCharsets.ISO_8859_1);
    Files.writeString(fileUtf8, msg, StandardCharsets.UTF_8);

    var pos = Files.mismatch(fileLatin, fileUtf8);
    System.out.println("Different encoding: mismatch(File1, File2)  = " + pos);
}
```

Die beiden Texte sind bezogen auf die Buchstaben identisch, sie weichen jedoch durch die Codierung des Umlauts `ü` an Position 1 auf Byte-Ebene voneinander ab. Daher erhalten wir folgende Ausgabe (unter Windows erhält man eine 2 – aufgrund der zu Unix-Systemen abweichenden Byte-Reihenfolge):

```
Different encoding: mismatch(File1, File2)  = 1
```

14.2.3 Der `teeing()`-Kollektor

Das umfangreiche Stream-API besitzt eine Menge an vordefinierten Kollektoren, unter anderem `toCollection()`, `toList()`, `toSet()` und `toMap()`, die die Elemente des Streams in die entsprechende Collection überführen. Mit Java 10 kamen bekanntlich diverse Unmodifiable-Kollektoren `toUnmodifiableXyz()` hinzu (vgl. Abschnitt 7.1.8).

Man könnte sich fragen, was denn noch fehlt. Ein spezieller Anwendungsfall ist beispielsweise das Zusammenfassen von Streams oder auch das Verarbeiten von Werten eines Streams mit mehreren Kollektoren. Ersteres wird Zipping genannt und leider (noch) nicht im JDK angeboten. Letzteres wird als Teeing bezeichnet und ist ein Bestandteil von JDK 12. Der zunächst vielleicht etwas merkwürdige Name stammt vom englischen Wort für T-Stück / Verbindungsstück. Die Arbeitsweise wird durch die nachfolgende Grafik veranschaulicht.

Abbildung 14-1 *Schematische Arbeitsweise des Teeing-Kollektors*

Beispiel 1

Nehmen wir an, es sollen die Zahlenwerte eines Streams in einem Rutsch sowohl summiert als auch deren Anzahl ermittelt werden. Das lässt sich mithilfe des Teeing-Kollektors und zweier weiterer Kollektoren recht einfach formulieren. Als Ausgangsbasis dienen zwei Streams aus Zahlen. Die gerade beschriebene Funktionalität wird in der Methode `calcCountAndSum()` realisiert – deren Lesbarkeit erhöht sich durch den statischen Import der fett markierten Kollektoren:

```
public static void main(final String[] args)
{
    System.out.println(calcCountAndSum(Stream.of(1, 2, 3, 4, 5, 6)));

    System.out.println(calcCountAndSum(Stream.of(1, 3, 5, 7, 11, 13, 17)));
}

private static LongPair calcCountAndSum(Stream<Integer> numbers)
{
    return numbers.collect(teeing(
                                  counting(),
                                  summingLong(n -> n),
                                  (count, sum) -> new LongPair(count, sum)));
}

record LongPair(long count, long sum)
{
}
```

Listing 14.2 *Ausführbar als* **'TEEINGCOLLECTOREXAMPLE'**

Zur Definition des Ergebnis-Tupels nutzen wir die zuvor schon vorgestellten Records (vgl. Abschnitt 14.1.3). Beim Ausführen des Programms TEEINGCOLLECTOREXAMPLE erhält man als Ergebnis folgende Ausgaben:

```
LongPair[count=6, sum=21]
LongPair[count=7, sum=57]
```

Bitte beachten Sie, dass es sich hierbei lediglich um ein einführendes Beispiel handelt, dessen Ergebnisse sich einfacher mithilfe des `summarizingInt()`-Kollektors hätten ermitteln lassen – auf einem `IntStream` sogar noch eleganter durch direkten Aufruf von `summaryStatistics()`.

Beispiel 2

Machen wir das Ganze ein wenig anspruchsvoller: Aus einem Stream von Strings sollen nun unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden. Hierbei helfen der `toList()`-Kollektor aus Java 8, der `filtering()`-Kollektor aus Java 9 sowie `teeing()` aus Java 12.

Zunächst definieren wir einen `Stream<String>` mit einigen Namen sowie zwei Prädikate für die Filterung. Das erste prüft auf die Startzeichenfolge '`Mi`'. Das zweite testet, ob ein String mit '`m`' endet. Schließlich sollen die so ermittelten Ergebnislisten kombiniert werden, wozu wir eine `BiFunction` nutzen, die eine Liste von Listen erzeugt und auf die Collection-Factory-Methoden aus Java 9 (vgl. Abschnitt 6.1.13) zurückgreift. Das Ganze ergibt dann folgendes Programmfragment:

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd", "Carsten");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");

// Wäre var hier nicht wünschenswert? => siehe Text
final BiFunction<List<String>, List<String>, List<List<String>>> combineLists =
                (list1, list2) -> List.of(list1, list2);

var result = names.collect(teeing(filtering(startsWithMi, toList()),
                                  filtering(endsWithM, toList()),
                                  combineLists));
```

Bevor wir uns das Ergebnis anschauen, möchte ich noch auf ein Detail eingehen: Die recht umfangreiche, mit diversen Generics durchzogene Definition der Merge-Funktion `combineLists` scheint ein Paradebeispiel zu sein, bei dem `var` deutlich zur Schreibweisenabkürzung beitragen könnte. Leider ist das – wie schon in Abschnitt 5.7.2 erwähnt – aufgrund der fehlenden Typermittlung bei Lambdas (noch) nicht möglich.

Kommen wir nun zum Ergebnis. Man erhält folgende Ausgaben, die die jeweiligen Teilergebnisse als Listen und dann die Kombination sehr schön zeigen, wenn man die Zeilen ausführt:

```
[[Michael, Mike], [Tim, Tom]]
```

Zum Nachvollziehen benötigen Sie noch den Import der genutzten Kollektoren:

```
import static java.util.stream.Collectors.*
```

14.3 JVM-Neuerungen

Bezüglich Neuerungen in der JVM schauen wir uns eine Verbesserung bei der Nachvollziehbarkeit von `NullPointerException`s sowie die (schmerzliche) Entfernung der JavaScript-Engine an.

14.3.1 Verbesserung bei `NullPointerExceptions`

In Ihrer Karriere als Java-Entwickler ist Ihnen bestimmt schon die eine oder andere `NullPointerException` untergekommen. Mitunter machen sich so (versehentlich) nicht initialisierte Referenzen bemerkbar. Dadurch lässt sich ein Fehler recht gut lokalisieren, allerdings nur durch den Namen der Java-Datei inklusive der Zeilennummer:

```
Exception in thread "main" java.lang.NullPointerException
    at java14.NPE_Example.main(NPE_Example.java:8)
```

Hilfreich ist diese Angabe vor allem für einfache Konstrukte wie im folgenden Beispiel für eine nicht initialisierte Variable:

```
public class NPE_Example
{
    public static void main(final String[] args)
    {
        A a = null;
        a.value = "ERROR";
    }
}
```

Schöner ist allerdings folgende seit Java 14 erscheinende Nachricht:

```
Exception in thread "main" java.lang.NullPointerException:
        Cannot assign field 'value' because 'a' is null.
    at java14.NPE_Example.main(NPE_Example.java:8)
```

Damit diese Informationen aufbereitet werden, bedarf es der Aktivierung durch den JVM-Parameter `-XX:+ShowCodeDetailsInExceptionMessages`.

Hinweis: Aus dem Nähkästchen

Um diese hilfreichen Informationen bereitstellen zu können, war eine Erweiterung in der JVM nötig. Zur Aufbereitung einer dynamische Fehlermeldung wird der Bytecode analysiert und basierend darauf werden die Hinweise generiert. Das Ganze funktioniert sogar bei geschachtelten Methodenaufrufen.

Beispiele

Neben dem einleitenden Beispiel wollen wir zwei weitere Fälle betrachten:

```
public class NPE_Second_Example
{
    public static void main(final String[] args)
    {
        try
        {
            final String[] stringArray = { null, null, null };
            final int errorPos = stringArray[2].lastIndexOf("ERROR");
        }
        // In der Praxis bitte keine NullPointerException fangen,
        // hier nur damit die folgenden Anweisungen ausgeführt werden
        catch (final NullPointerException e)
        {
            e.printStackTrace();
        }

        try
        {
            final Integer value = null;
            final int sum = value + 3;
        }
        catch (final NullPointerException e)
        {
            e.printStackTrace();
        }
    }
}
```

Auch hier wird wieder ein hilfreicher Hinweis beim Zugriff auf ein nicht korrekt initialisiertes Array bzw. beim Auto-Unboxing einer `null`-Referenz erzeugt:

```
java.lang.NullPointerException: Cannot invoke "String.lastIndexOf(String)"
      because "stringArray[2]" is null
    at java14.NPE_Second_Example.main(NPE_Second_Example.java:28)
java.lang.NullPointerException: Cannot invoke "java.lang.Integer.intValue()"
      because "value" is null
    at java14.NPE_Second_Example.main(NPE_Second_Example.java:38)
```

Fehlersuche bei komplexen Ausdrücken

Bei komplexen Ausdrücken kann eine Analyse der tatsächlichen Fehlerursache manchmal schwierig werden, weil nicht offensichtlich ist, welche Referenz in einem Ausdruck die Ursache der `NullPointerException` war. Denken Sie an Sourcecode wie den folgenden, bei dem nicht sofort beim Betrachten klar ist, welche Variable in dem `NullPointerException`-auslösenden Konstrukt denn nun `null` war:

```
a.b.c.d.e = "ERROR";
```

In solchen Fällen wäre es wünschenswert, wenn man ein paar zusätzliche Informationen bereitgestellt bekommen würde. Dies wird durch die Neuerung in Java 14 erreicht.

Schauen wir dazu ein praxisrelevanteres Beispiel an.

```
int width = getWindowManager().getWindow(5).size().width();
```

Solche Verkettungen mit der .-Notation zum Zugriff auf referenzierte Objekte – bitte verwechseln Sie dies nicht mit empfehlenswerten Fluent-APIs, die auf dem eigenen oder einem direkt assoziierten Objekt arbeiten – verstoßen gegen gutes Design, vor allem das sogenannte Law of Demeter.[6] Trotzdem sieht man sie in der Praxis (leider) häufiger. Mitunter kommt es durch uninitialisierte Attribute in Kombination mit diesen Konstrukten dann zu `NullPointerException`s, deren Ursachen sich nur mühselig ermitteln lassen. Mit der Neuerung aus Java 14 wird dies wesentlich einfacher:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java14
      .NPE_Third_Example$Window.size()" because the return value of "java14.
      NPE_Third_Example$WindowManager.getWindow(int)" is null
    at java14.NPE_Third_Example.main(NPE_Third_Example.java:24)
```

Betrachten wir abschließend noch den Applikationscode, in dem die im oben gezeigten Aufruf verwendeten Methoden definiert werden:

```
public static void main(final String[] args)
{
    System.out.println(getWindowManager().getWindow(5).size().width());
}

public static WindowManager getWindowManager()
{
    return new WindowManager();
}

public static class WindowManager
{
    public Window getWindow(final int i)
    {
        return null;
    }
}

public static record Window(Size size) {};
public static record Size(int width, int height) {};
```

Hinweis: Aktivierung der Java-15-Preview-Syntaxerweiterungen

Bitte denken Sie daran, dass sowohl beim Kompilieren als auch beim Ausführen jeweils der Kommandozeilenparameter `--enable-preview` angegeben bzw. die korrespondierende Einstellung in der IDE aktiviert werden muss (vgl. Abschnitt 14.5).

[6] Dieses besagt, dass Objekte nur mit anderen Objekten interagieren sollen, die sie direkt kennen oder selbst erzeugt haben. Für Details schauen Sie doch bitte in Abschnitt 3.5.2 nach.

14.3.2 Entfernung der JavaScript-Engine

Mit Java 15 wird die bisher in das JDK integrierte JavaScript-Engine Nashorn aus dem JDK entfernt. Begründet wird dieser Schritt mit den Aufwänden zum Schritthalten mit den Neuerungen im JavaScript-Sprachstandard sowie möglichen Alternativen wie der GraalVM.

Dieser Schritt ist zum einen nachvollziehbar, aber zum anderen für all diejenigen Anwendungsfälle auch schmerzlich, die nicht die topaktuellste JavaScript-Version benötigen, sondern nur dynamische Berechnungen ausführen wollen, etwa einen textuellen mathematischen Ausdruck wie $7 + 2 * 71$ oder $sin(x) + 5 * cos(x)$ ausrechnen wollen. So etwas konnte man beispielsweise für Taschenrechner oder Funktionsplotter mit benutzerdefinierten Formeln gut einsetzen.

14.4 Microbenchmark Suite

Mitunter sind einige Teile der eigenen Software nicht so performant, wie man es sich erhofft hat bzw. wie es vom Nutzer erwartet wird. Zur Optimierung der Performance gibt es verschiedene Hilfsmittel. Generell sollte man zunächst gründlich messen und nur mit Bedacht und vor allem nicht direkt optimieren. Das gilt unter anderem deshalb, weil zum einen die JVM bereits diverse Optimierungen eingebaut hat und zum anderen man rein auf Basis von Vermutungen häufig falsch liegt und so nur Programmteile schlechter les- und wartbar macht, jedoch die Gesamtperformance kaum oder überhaupt nicht verbessert. Als Abhilfe gibt es oftmals verschiedene Möglichkeiten. Manchmal reicht ein anderer Algorithmus und manchmal kann Caching helfen. Immer jedoch empfiehlt es sich, die für die Performance kritischen Stellen zu ermitteln und auch nur dort zu optimieren. Diese Aussage gilt für die Architektur und das Design bis hin zur Low-Level-Implementierung. Das Thema Optimierung wird später in Kapitel 22 detailliert behandelt.

Wie schon erwähnt, sollte man die kritischen Stellen keinesfalls nur aufgrund von Vermutungen identifizieren, sondern basierend auf Messungen. Dazu kann man beispielsweise einfache Start-Stopp-Messungen einsetzen, empfehlenswerter sind ausgeklügeltere Verfahren mit mehreren Durchläufen, um temporäre Störeffekte, verursacht durch höhere Systemlast oder von Garbage Collections, zu vermeiden. Dabei hilft die mit Java 12 in das JDK aufgenommene Microbenchmark Suite, die auf dem Toolkit Java Microbenchmark Harness (JMH) basiert.

14.4.1 Eigene Microbenchmarks und Varianten davon

Nachfolgend schauen wir uns die Optimierung auf der Ebene von einzelnen bzw. wenigen Anweisungen an. Dazu nutzt man sogenannte Microbenchmarks. Bedenken Sie bitte aber vorab, dass derartige Optimierungen nur dann sinnvoll sind, wenn auf den höheren Ebenen wie Architektur, Design und Algorithmus keine Verbesserungen mehr

erreicht werden können oder die zu optimierenden Anweisungen extrem häufig ausgeführt werden und somit die Gesamtperformance maßgeblich beeinflussen.

Nehmen wir an, wir wollten Aussagen über die Performance für einige Varianten einer Implementierung treffen. Grundsätzlich ist das nicht so einfach, wie man zunächst annehmen könnte, da die Messungen unter möglichst gleichen Bedingungen (CPU-Last, Speicherverbrauch usw.) geschehen sollten, damit man vergleichbare Resultate erhält. Aber auch Optimierungsmechanismen in der JVM wie Loop-Unrolling, Hotspot-Kompilierung usw. führen potenziell dazu, dass sich Messwerte signifikant voneinander unterscheiden.

Einfache Start-Stopp-Messungen

Im einfachsten Fall nutzt man eine Art Stoppuhr, indem man den zu messenden Programmteil mit Aufrufen von `System.currentTimeMillis()` wie folgt umklammert und die Differenz zwischen den Werten ermittelt:

```
// ACHTUNG: keine gute Variante
final long startTimeMs = System.currentTimeMillis();

someCodeToMeasure();

final long stopTimeMs = System.currentTimeMillis();
final long duration = stopTimeMs - startTimeMs;
```

Das scheint vernünftig. Jedoch nutzt `System.currentTimeMillis()` – zumindest unter Windows – ungenaue, zu grobgranulare Zeitgeber des Betriebssystems. Verlässlichere Werte erhält man mithilfe der folgenden Methode:

```
System.nanoTime()
```

Wiederholte Start-Stopp-Messungen

Unabhängig vom Zeitgeber liefern einfache Start-Stopp-Messungen eher unzuverlässige Ergebnisse, da die Messungen durch unterschiedlichste Effekte verfälscht werden können. Deswegen ist es oftmals sinnvoller, eine gewisse Anzahl an Iterationen zu nutzen und dann den Durchschnittswert als Referenzmesswert zu errechnen:

```
// bessere Variante
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
   someCodeToMeasure();
}

final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

Diese Variante mit mehreren Durchläufen und Durchschnittsbildung ist weniger anfällig für Systemlastschwankungen oder sonstige Störeinflüsse. Zudem ist es recht einfach möglich, auch Minimal- und Maximaldauer oder die Standardabweichung zu ermitteln. Bei diesen erweiterten Anforderungen lagert man diese Funktionalität aber sinnvollerweise in eine oder mehrere separate Klassen aus.

Wiederholte Start-Stopp-Messungen mit Warm-up

Neben kleineren externen Störungen beobachtet man immer mal wieder Einschwing-Effekte: Erst nach einer gewissen Anzahl an Durchläufen zeigt eine Funktionalität ihre optimale Laufzeit: Das kann etwa dadurch verursacht werden, dass Caches zunächst gefüllt werden oder aber der Hotspot-Optimierer verschiedene Codestellen in Maschinensprache übersetzt. Für diese Fälle empfiehlt sich dann ein Warm-up vor der eigentlichen Messung.

Nachfolgend zeige ich exemplarisch ein der eigentlichen Performance-Messung vorangestelltes Warm-up in Form einiger Schleifendurchläufe mit Aufrufen an die später zu messende Funktionalität:

```
// noch bessere Variante der Messung durch Warm-up
for (int i = 0; i < MAX_WARMUP_ITERATIONS; i++)
{
   someCodeToMeasure();
}

// eigentliche Messung nach Warm-up
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
   someCodeToMeasure();
}
final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

Man erkennt gut, dass die Implementierung der Messung immer unhandlicher wird. Das ist oftmals ein Zeichen dafür, dass man nach einem unterstützenden Framework Ausschau halten sollte. Glücklicherweise gibt es das JMH, das wir nun genauer kennenlernen.

Tipp: Hotspot-Optimierer

Wenn man ein Java-Programm startet, so liegt dieses zunächst in Bytecode vor. Dieser wird von der JVM interpretiert. Obwohl dies in den letzten Jahren immer performanter geworden ist, reicht die Ausführungsgeschwindigkeit interpretierten Bytecodes natürlich nicht ganz an die von Maschinensprache heran. In die JVM ist der Hotspot-Optimierer inklusive eines JIT-Compilers (JIT = Just In Time) integriert. Beide sorgen dafür, dass besonders häufig durchlaufene Teile des Bytecodes erkannt und on the fly in Maschinensprache transferiert werden.

Aufgrund dieser Besonderheit betrachtet man bei naiven Messungen der Performance eventuell nur die interpretierte Ausführung, gegebenenfalls aber auch eine Mischform, wenn bereits ein Teil des Bytecodes in Maschinensprache übersetzt wurde.

Mithilfe des Aufrufparameters `-XX:+PrintCompilation` kann man die Kompilierung protokollieren lassen und `-XX:CompileThreshold` erlaubt es, die Anzahl der benötigten Ausführungen eines Programmstücks bis zu dessen Kompilierung festzulegen. Standardmäßig benötigt es in der seit Java 11 nur noch vorhandenen Servervariante 10.000 Abarbeitungen – die früher verfügbare Client-JVM hat bereits nach 1.500 Abarbeitungen den Bytecode kompiliert.

14.4.2 Microbenchmarks mit JMH

Seit Java 12 ist das Open-Source-Framework Java Microbenchmark Harness (JMH) ins JDK aufgenommen worden und adressiert die oben angedeuteten sowie weitere Schwierigkeiten bei Performance-Messungen.

Einstieg in JMH

JMH arbeitet mit Annotations und integriert basierend darauf verschiedene Messungen. Hierzu bietet sich die Nutzung eines Maven-Archetypes an. Folgender Aufruf erzeugt ein neues Verzeichnis mit einem vollständig initialisierten JMH-Benchmark:

```
mvn archetype:generate \
    -DinteractiveMode=false \
    -DarchetypeGroupId=org.openjdk.jmh \
    -DarchetypeArtifactId=jmh-java-benchmark-archetype \
    -DgroupId=org.sample \
    -DartifactId=jmh-test \
    -Dversion=1.0-SNAPSHOT
```

Schauen Sie doch kurz in Abschnitt 2.7.1, um etwas mit Maven vertraut zu werden.

Dadurch entstehen folgende Verzeichnisse und Dateien:

```
.
`-- jmh-test
    |-- pom.xml
    `-- src
        `-- main
            `-- java
                `-- org
                    `-- sample
                        `-- MyBenchmark.java
```

Als Grundgerüst wird eine Klasse `MyBenchmark` erzeugt, weshalb sich dort auch ein englischer Kommentar findet. In der generierten Klasse können wir nun die zu beobachtenden Funktionalitäten aufrufen. Ähnlich zu JUnit lassen sich die gewünschten Einstellungen zu den Messungen per Annotations vorgeben. Die wichtigste ist `@Benchmark`, die eine zu untersuchende Methode kennzeichnet:

```
public class MyBenchmark
{
    @Benchmark
    public void testMethod()
    {
        // This is a demo/sample template for building your JMH benchmarks.
        // Edit as needed. Put your benchmark code here.
    }
}
```

Um dann den Benchmark auszuführen, muss man in das durch den obigen Aufruf erzeugte Verzeichnis wechseln und dort – wie von Maven gewohnt – kompilieren und paketieren:

```
mvn package
```

Dadurch entsteht ein entsprechendes Artefakt `benchmarks.jar` im Verzeichnis `target`. Weil dort insbesondere alle benötigten Klassen und Bibliotheken bereits enthalten sind, kann man das JAR ohne weitere Angaben folgendermaßen starten:

```
java -jar target/benchmarks.jar
```

Da wir bislang noch keine sinnvolle Funktionalität, sondern nur eine leere Methode messen würden, heben wir uns den Start für das nächste Beispiel auf.

Der erste Benchmark in JMH

Wir haben gerade gesehen, wie man eine Performance-Suite mit JMH erzeugt. Allerdings weiß das Framework ja noch nicht, was gemessen werden soll. Deshalb wird auch nur eine Klasse `MyBenchmark` als Grundgerüst erzeugt. Dort können wir die zu beobachtenden Funktionalitäten aufrufen und die gewünschten Messeinstellungen per Annotations vorgeben. Generell empfiehlt es sich aber, einen sprechenderen Namen für

die Benchmark-Klasse zu nutzen, nachfolgend etwa `FirstIntegerBenchmark`, der die Performance der Methoden `toHexString()` und `toBinaryString()` misst.

Insgesamt wird das Schreiben von Microbenchmarks durch JMH deutlich erleichtert: Praktischerweise kann man zwar eine recht feingranulare Steuerung vornehmen, aber dem Gedanken *Convention over Configuration* folgend ist es für erste Messungen oftmals ausreichend, wenn man lediglich die Annotation `@Benchmark` wie folgt nutzt – die Zahlen sind mithilfe des Unterstrichs an den Positionen der Tausenderpunkte etwas besser lesbar:

```
public class FirstIntegerBenchmark
{
    @Benchmark
    public String numberAsHexString()
    {
        final int number = 1_234_567;
        return Integer.toHexString(number);
    }

    @Benchmark
    public String numberAsBinaryString()
    {
        final int number = 123_456_789;
        return Integer.toBinaryString(number);
    }
}
```

Nachdem wir beide Klassen kompiliert und paketiert haben, starten wir das ausführbare JAR. Es ergeben sich in etwa folgende gekürzten Konsolenausgaben:

```
# JMH version: 1.23
# VM version: JDK 14, OpenJDK 64-Bit Server VM, 14+36-1461
# VM invoker: /Library/Java/JavaVirtualMachines/jdk-14.jdk/Contents/Home/bin/
      java
# VM options: <none>
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.FirstIntegerBenchmark.numberAsBinaryString

# Run progress: 0.00% complete, ETA 00:16:40
# Fork: 1 of 5
# Warmup Iteration   1: 56326650.457 ops/s
# Warmup Iteration   2: 59780600.908 ops/s
# Warmup Iteration   3: 64051264.973 ops/s
# Warmup Iteration   4: 64008812.168 ops/s
# Warmup Iteration   5: 64186426.089 ops/s
Iteration   1: 61304141.790 ops/s
Iteration   2: 61953075.846 ops/s
Iteration   3: 63036185.662 ops/s
Iteration   4: 64322247.337 ops/s
Iteration   5: 64872604.082 ops/s
...
```

Nach längerer Ausführung erhält man dann ungefähr folgende Ergebnisse:

```
Result "org.sample.FirstIntegerBenchmark.numberAsBinaryString":
  64224040.592 ±(99.9%) 978073.744 ops/s [Average]
  (min, avg, max) = (61298595.526, 64224040.592, 65767851.332), stdev =
        1305700.466
  CI (99.9%): [63245966.848, 65202114.336] (assumes normal distribution)

Result "org.sample.FirstIntegerBenchmark.numberAsHexString":
  145802842.871 ±(99.9%) 1765144.419 ops/s [Average]
  (min, avg, max) = (140986090.679, 145802842.871, 149109246.667), stdev =
        2356417.299
  CI (99.9%): [144037698.452, 147567987.290] (assumes normal distribution)
```

Möchte man den Benchmark noch ein wenig feintunen, beispielsweise die Anzahl an Iterationen oder das Warm-up und die Anzahl an Messdurchgängen konfigurieren, so könnte man dies folgendermaßen mithilfe der Annotations `@Warmup` sowie `@Measurement` tun:

```
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.annotations.Fork;

@Measurement(iterations = 3, time = 1000, timeUnit = TimeUnit.MILLISECONDS)
@Warmup(iterations = 7, time = 1000, timeUnit = TimeUnit.MICROSECONDS)
@Fork(4)
public class MyBenchmark
{
     ....
}
```

Wendet man diese Werte auf den ersten Benchmark an, dann ändert sich die Ausgabe wie folgt:

```
# Benchmark: org.sample.FirstIntegerBenchmark.numberAsBinaryString

# Run progress: 0.00% complete, ETA 00:33:44
# Fork: 1 of 4
# Warmup Iteration   1: 2612321.666 ops/s
# Warmup Iteration   2: 6316193.097 ops/s
# Warmup Iteration   3: 7323288.665 ops/s
# Warmup Iteration   4: 4430635.983 ops/s
# Warmup Iteration   5: 6554600.103 ops/s
# Warmup Iteration   6: 4660414.309 ops/s
# Warmup Iteration   7: 5267248.079 ops/s
Iteration   1: 32894213.089 ops/s
Iteration   2: 34024432.225 ops/s
Iteration   3: 46131122.469 ops/s
```

Man erkennt sehr schön den zuvor beschriebenen Einfluss der Annotations: Es gibt erwartungsgemäß vier Durchläufe mit jeweils sieben Warm-ups und drei Iterationen.

Der zweite Benchmark in JMH

Um noch ein wenig vertrauter mit JMH zu werden, erstellen wir eine Messung für die arithmetischen Operationen * und / sowie für das Potenzieren von `int`-Werten:

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Scope;

import java.util.concurrent.TimeUnit;

@State(Scope.Benchmark)
public class SimpleMathBenchmark
{
    int x = 7271;
    int y = 1234;

    @Benchmark
    @BenchmarkMode({Mode.Throughput, Mode.AverageTime})
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public long mult()
    {
        return x * y;
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public long div()
    {
        return x / y;
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.MILLISECONDS)
    public double pow()
    {
        return Math.pow(x, y);
    }
}
```

Im Listing lernen wir einige neue Annotations kennen:

- `@State` – Wie eingangs erwähnt, finden zur Performance-Messung mehrere Durchläufe statt. Es lässt sich spezifizieren, wie der Zustand zwischen den Messungen und den Threads geteilt werden soll. Gibt es einen eher globalen Zustand, so lässt sich die Benchmark-Klasse annotieren. Man kann aber auch innere Klassen nutzen und dort den Zustand definieren. Über den Scope wird die Lebensdauer eines State-Objekts festgelegt. Im Thread-Scope erhält jeder Thread seinen unabhängigen Zustand. Hier nutzen wir den Zustand pro Benchmark: Damit ist dieser für alle Threads gleich.

- `@BenchmarkMode` – Oftmals ist zur Performance-Beurteilung die Anzahl an Operationen pro Zeiteinheit (ops/time = Default) oder die durchschnittliche Ausführungszeit (time/op) von Interesse. Beides verwenden wir in diesem Beispiel.
- `@OutputTimeUnit` – Mit JMH werden die Laufzeiten normalerweise in Nanosekunden berechnet. Mitunter ist man aber an anderen Zeitskalen interessiert, wie oben an Millisekunden für die Potenzierung.

Nachdem wir beide Klassen kompiliert und paketiert haben, starten wir das Executable JAR. Das produziert Konsolenausgaben etwa wie folgt:

```
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.SimpleMathBenchmark.mult

Result "org.sample.SimpleMathBenchmark.mult":
  0.401 ±(99.9%) 0.012 ops/ns [Average]
  (min, avg, max) = (0.348, 0.401, 0.414), stdev = 0.017
  CI (99.9%): [0.388, 0.413] (assumes normal distribution)

# Benchmark mode: Average time, time/op
# Benchmark: org.sample.SimpleMathBenchmark.mult

Result "org.sample.SimpleMathBenchmark.mult":
  2.568 ±(99.9%) 0.108 ns/op [Average]
  (min, avg, max) = (2.414, 2.568, 2.931), stdev = 0.144
  CI (99.9%): [2.460, 2.676] (assumes normal distribution)
```

Zunächst werden zwischen den einzelnen Iterationen die Ergebnisse präsentiert und zum Abschuss erfolgt die Ausgabe einer Zusammenfassung.

```
Benchmark                                    Mode  Cnt         Score
      Error   Units
FirstIntegerBenchmark.numberAsBinaryString  thrpt   12  36490099.791 ±
      7938126.104   ops/s
FirstIntegerBenchmark.numberAsHexString     thrpt   12  78930550.252 ±
      18442012.239  ops/s
SimpleMathBenchmark.mult                    thrpt   25         0.401 ±
      0.012  ops/ns
SimpleMathBenchmark.div                      avgt   25         4.445 ±
      0.115   ns/op
SimpleMathBenchmark.mult                     avgt   25         2.568 ±
      0.108   ns/op
SimpleMathBenchmark.pow                      avgt   25        ≈ 10^5
                    ms/op
```

Mit deren Hilfe kann man verschiedene Aussagen zur Performance ableiten. So sieht man beispielsweise, dass die Multiplikation rund 2,5 ns benötigt, eine Division jedoch rund 4,4 ns. Das Potenzieren hat eine Laufzeit von 0,00005 ms (50 ns) und ist damit um Größenordnungen langsamer als die einfachen Operationen * und /. Das war zu erwarten, lässt sich aber so auch gut nachvollziehen.

Der dritte Benchmark in JMH

Obwohl die Ergebnisse des letzten Benchmarks relativ erwartungskonform waren, könnten diese doch verfälscht sein. Die JVM ist in der Lage, die Anweisungen so umfassend zu analysieren, dass nicht benutzter Code erkannt wird. Um zu vermeiden, dass die JVM dann ganze Codeteile wegoptimiert, gibt es sogenannte `Blackhole`s. Diese konsumieren Berechnungsergebnisse und verhindern somit, dass die JVM Berechnungen nicht ausführt, weil das Ergebnis nicht weiterverarbeitet wird.

Im nachfolgenden Beispiel sehen wir zum einen die Definition und Übergabe von Zustand per `@State` sowie den Parameter namens `state` an die Benchmarking-Methoden als auch den Einsatz von `Blackhole`:

```
@OutputTimeUnit(TimeUnit.MICROSECONDS)
public class BooleanPoolingOptimizationJMH
{
   @State(Scope.Benchmark)
   public static class MyBenchmarkState
   {
      final Boolean[] testArray = new Boolean[10_000_000];
   }

   @Benchmark
   @BenchmarkMode(Mode.AverageTime)
   public void newlyCreatedBooleans(MyBenchmarkState state, Blackhole blackhole)
   {
      for (int i = 0; i <  state.testArray.length; i++)
      {
         state.testArray[i] = new Boolean(true);
      }

      blackhole.consume(state.testArray);
   }

    // ...
}
```

14.4.3 Fazit zu JMH

Mitunter ist es wichtig, Aussagen über die Laufzeit spezieller Programmteile zu erhalten, um Optimierungspotenziale zu ermitteln. Neben einem Profiler, der Messungen zur Laufzeit ermöglicht, ist es manchmal auch wünschenswert, die Performance spezieller Anweisungen oder Methoden feingranular zu messen. Implementiert man dies jedoch von Hand, so wird man mit einigen Schwierigkeiten und Tücken konfrontiert. Sinnvoller ist es, hierzu JMH zu nutzen, das das Erstellen korrekter Microbenchmarks deutlich erleichtert. In diesem Unterkapitel konnte ich Ihnen lediglich einen ersten Eindruck von den Möglichkeiten von JMH vermitteln, es gibt noch sehr viel mehr zu entdecken.

14.5 Java 15 – notwendige Anpassungen für Build-Tools und IDEs

Zur Demonstration der Auswirkungen von Java 15 auf Build-Tools und IDEs nutzen wir ein paar einfache Beispielapplikationen mit folgendem Verzeichnisaufbau:

```
Java15Examples
|-- build.gradle
|-- pom.xml
`-- src
    `-- main
        `-- java
            `-- java15
                |-- InstanceOfExamples.java
                |-- LocalEnumAndInterfaceExamples.java
                |-- RecordExamples.java
                |-- SealedTypesExamples.java
                `-- TextBlockExamples.java
```

Weil wir einige Aktionen auch auf der Kommandozeile ausführen, sei noch mal darauf hingewiesen, dass Sie zum Nachvollziehen

- das aktuelle Java 15 heruntergeladen und installiert haben müssen sowie
- die Umgebungsvariablen wie `JAVA_HOME` und `PATH` passend auf die Java-Installation verweisen lassen. Für Mac OS geschieht dies etwa wie folgt:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-15.jdk/Contents/Home
export PATH=$JAVA_HOME/bin:$PATH
```

14.5.1 Java 15 mit Gradle

Für Java 15 benötigt man ein aktuelles Gradle 6.6.1 (oder neuer) und ein paar passende Angaben in der Datei `build.gradle`:

```
plugins {
    id 'java'
}

sourceCompatibility=15
targetCompatibility=15

repositories
{
    jcenter();
}

// Aktivierung von Preview-Features
tasks.withType(JavaCompile) {
    options.compilerArgs += ["--enable-preview"]
}
```

Danach können wir wie gewohnt mit

```
gradle clean assemble
```

ein korrespondierendes JAR bauen und die Klasse `TextBlockExamples` wie folgt daraus starten:

```
java --enable-preview -cp build/libs/Java15Examples.jar \
     java15.TextBlockExamples
```

Dabei ist es wichtig, den Kommandozeilenparameter `--enable-preview` anzugeben. Denn nur dadurch lässt sich das Programm auch starten.

14.5.2 Java 15 mit Maven

Mit der Anfang Februar 2020 veröffentlichten Version von Maven 3.6.1_1 (dabei ist genau dieses _1 wichtig, weil erst diese Unterversion den Support für Java 14 bringt) und dem Maven-Compiler-Plugin in der Version 3.8.1 lassen sich Java-15-Projekte ohne Probleme übersetzen. Mittlerweile ist im September 2020 eine weitere offizielle Version erschienen, die nur noch die Versionsnummer 3.6.3 trägt. Achten Sie also auf den Download einer möglichst aktuellen Maven-Version.

Um Java 15 mit Maven zu nutzen, muss man folgende Angaben in der Datei `pom.xml` vornehmen:

```
<plugins>
    <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
            <release>15</release>
            <!-- Aktivierung von Preview-Features -->
            <compilerArgs>--enable-preview</compilerArgs>
        </configuration>
    </plugin>
</plugins>
```

Mit diesen Modifikationen lässt sich ein Maven-Build mit

```
mvn clean package
```

ausführen und es wird das korrespondierende JAR erzeugt.

14.5.3 Java 15 mit Eclipse

Eclipse bietet in Version 2020-09 Support für Java 15. Durch die kurze Zeit zwischen der Veröffentlichung von Java 15 und Eclipse 2020-09 ist der Support noch nicht fix integriert, sondern muss als Plugin über den Eclipse Marketplace nachinstalliert werden, wie es in Abbildung 14-2 angedeutet ist.

Abbildung 14-2 *Installation Java-15-Plugin im Eclipse Marketplace*

Selbstverständlich müssen Sie noch den Compiler auf Java 15 einstellen und die Preview-Features aktivieren, um alle Beispiele ausprobieren zu können.

Abbildung 14-3 *Compiler-Einstellungen in Eclipse für Java 15*

14.5.4 Java 15 mit IntelliJ

Um mit Java 15 zu experimentieren, nutzen Sie bitte IntelliJ IDEA 2020.2 oder aktueller. Allerdings sind noch ein paar Einstellungen vorzunehmen und folgende Dinge zu beachten:

- Im Dialog »Project Structure« (Abbildung 14-4) muss man im Bereich »`Project SDK`« den Wert 15 auswählen und im Bereich »`Project language level`« den Wert »`15 (Preview) - Sealed types, records, patterns, text blocks, local enums and interfaces`« einstellen.
- Zum Ausführen des Beispiels ist es erforderlich, in der jeweiligen Run Configuration der zu startenden Klassen noch `--enable-preview` anzugeben.

Abbildung 14-4 *Einstellungen in Project Structure für Java 15*

14.5.5 Java 15 mit JShell oder der Kommandozeile

Durch die recht eng getakteten Releasezyklen von 6 Monaten ist die Tool-Landschaft mitunter noch nicht immer parat, wenn man schon mal mit Vorabversionen experimentieren möchte. Dann bieten sich folgende Alternativen an:

1. Kommandozeile – Selbstverständlich ist es immer möglich, mit `javac` und `java` zu arbeiten, wenn auch manchmal etwas mühsamer.
2. Kommandozeile – Seit Java 11 gibt es das Feature, Java-Dateien direkt durch Aufruf von `java`, also ohne explizites vorheriges Kompilieren, starten zu können. Das Editieren kann dann mit jedem beliebigen Texteditor oder einer IDE (gegebenenfalls mit der Markierung von Syntaxfehlern) erfolgen.
3. JShell – Die JShell ist eine Alternative für kleine Experimente, weil das REPL-Tool (REPL = Read Evaluate Print Loop) Bestandteil der JVM ist. Somit kann man immer auch die neuesten Features ausprobieren.

14.6 Fazit

Die Java-Releases 10, 12 und 13, aber auch 15 sind bezüglich ihrer Neuerungen doch eher überschaubar. Das gilt selbst für Java 11 als LTS-Version, die recht wenig Innovatives bietet. Glücklicherweise bringt Java 14 und damit indirekt auch Java 15 einen guten Schwung von sinnvollen Erweiterungen: Zum einen sind das die praktischen Syntaxänderungen bei `switch` und bei `instanceof`. Zum anderen gehören dazu auch die mehrzeiligen Strings, Text Blocks genannt. All diese Features sind schon ziemlich nützlich. Tatsächlich gibt es noch eine alles überragende Neuerung als Preview: die Records. Diese erlauben die Implementierung von Immutable-Klassen[7] mit minimalem Schreibaufwand. Records vereinfachen beispielsweise die Definition von DTO-Klassen, von Parameter Value Objects oder von Tupeln für kombinierte Rückgabetypen. Abgerundet wird Java 15 durch Verbesserungen bei `NullPointerException`s, die nun detaillierte Fehlermeldungen generieren können, wodurch sich eine nachfolgende Problemanalyse oftmals deutlich vereinfachen sollte.

Hinweis: Kurze Laufzeit von Java 15 durch die neue Releasestrategie

Bei Java 15 handelt es sich wie bei Java 9, 10, 12, 13 und 14 wieder um ein nur kurzzeitig gültiges Release, das dann bereits im März 2021 von Java 16 abgelöst werden wird.

Wenn Sie auf längerfristige Stabilität und die Verfügbarkeit von Security Updates setzen müssen, dann bleibt Java 11 als Alternative. Allerdings kommen Sie damit leider nicht in den Genuss der Detailänderungen aus Java 15 und den Folgeversionen bis zum LTS-Release von Java 17, das voraussichtlich im September 2021 veröffentlicht werden wird.

[7] Sofern alle Attribute entweder von primitivem Typ oder selbst wieder immutable sind.

IV Modularisierung

15 Modularisierung mit Project Jigsaw

Dieses Kapitel behandelt das Thema Modularisierung und insbesondere die im Project Jigsaw vorangetriebene Modularisierungslösung von Java und des JDKs. Dabei umfasst die Modularisierung zum einen das JDK an sich und zum anderen die Modularisierung von Anwendungen und Bibliotheken: Neben dem Aufbrechen des monolithischen JDKs in einzelne Module mit der Möglichkeit zur Beschreibung und Kontrolle von Abhängigkeiten sollte vor allem die fehlerträchtige Abhängigkeitsverwaltung basierend auf dem `CLASSPATH` durch eine verlässliche Konfiguration (***Reliable Configuration***) ersetzt werden. Diese verlangt, dass Module ihre Abhängigkeiten untereinander explizit und vollständig beschreiben müssen. Dadurch können Zyklen, nicht eindeutige Modulnamen und nicht aufgelöste Abhängigkeiten sowohl beim Kompilieren als auch zur Laufzeit geprüft und verhindert werden.

In den folgenden Abschnitten gebe ich einen Einstieg in die Modularisierung und lege damit den Grundstein für das Verständnis der Thematik. Zudem schauen wir uns konkret an, welche Änderungen in Bezug auf die Modularisierung in JDK 9 enthalten sind. Dabei lernen wir unter anderem, wie sich durch die Modularisierung die Wartbarkeit verbessert. Zudem werden ergänzend fortgeschrittene Themen wie Migrationsszenarien vorgestellt.

Verzeichnisaufbau

Weil einige IDEs noch keinen komfortablen Support für Module bieten und weil wir das Ganze von der Pike auf lernen wollen, werden wir in diesem Kapitel vor allem mit der Konsole arbeiten. Die Beispiele werden in jeweils eigenen Verzeichnissen entwickelt, wobei das Verzeichnis `jigsaw_ch15` parallel zum Verzeichnis `src/main/java` liegt. Dadurch ergibt sich folgendes High-Level-Verzeichnislayout:

```
jigsaw_ch15
+-- ch15_2_2_modules_example
+-- ch15_2_3_packaging_module_example
+-- ch15_2_4_linking_module_example
+-- ch15_2_5_dependencies_module_example
+-- ch15_2_6_include_jdk_modules_example
+-- ch15_3_2_accessibility
+-- ch15_4_1_migration_compatibility_mode
+-- ch15_4_3_pitfall_bottomup_migration
+-- ch15_4_4_migration_automatic_modules
+-- ch15_4_5_migration_automatic_and_unnamed_modules
```

15.1 Grundlagen

Für den Begriff Modularisierung hat vermutlich jeder Entwickler leicht unterschiedliche Ideen im Sinn. Deshalb möchte ich zunächst ein gemeinsames Verständnis für die Modularisierung in Java erreichen und insbesondere kurz auf mögliche Probleme und Schwierigkeiten, aber auch auf Anforderungen eingehen.

15.1.1 Begrifflichkeiten

Unter ***Modulen*** versteht man in sich abgeschlossene, idealerweise unabhängig deploybare Softwarebausteine (Komponenten), die eine möglichst klar abgegrenzte Funktionalität bereitstellen, ihre Abhängigkeiten zu anderen Modulen beschreiben und nach außen sichtbare Typen festlegen. Beachten Sie bitte, dass Jigsaw-Module in der Regel nicht unabhängig voneinander deployt werden können, wenn es Abhängigkeiten zwischen den Modulen gibt. Für optionale Funktionalität kann man aber Services nutzen. Dieses fortgeschrittene Thema wird in meinem Buch »Java - die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41] behandelt.

Modularisierung

Mit Modularisierung meint man häufig, dass ein Programm in mehrere Subsysteme oder Module unterteilt ist. Das erlaubt eine thematische Gruppierung von Klassen und Packages und hilft dabei, klare Zuständigkeiten und Abhängigkeiten zu erzielen.

Dem steht das monolithische Design gegenüber, bei dem die einzelnen Softwarekomponenten stark miteinander verknüpft sind. Eine Änderung an einer Stelle zieht oftmals diverse Folgeänderungen an ganz anderen Stellen im Programm nach sich. Erweiterbarkeit und Wartbarkeit leiden enorm. Das kann sich so dramatisch auswirken, dass man von Änderungen möglichst absieht, weil diese durch die Komplexität des Systems nahezu unbeherrschbar werden. Man sollte aber unbedingt vermeiden, in einen solchen Zustand einer »Schockstarre« zu kommen, damit man handlungsfähig bleibt. Ein sauberes Design unter Einhaltung wesentlicher OO-Designprinzipien sowie eine klare Strukturierung und Modularisierung können dabei helfen.

Anforderungen an Module

Bei der Modularisierung sollten folgende Eigenschaften für ein Modul als Softwarekomponente gelten. Jedes Modul ...

- besitzt einen eindeutigen Identifier (z. B. durch Name oder ID und eventuell eine Version),
- bietet Funktionalität über eine wohldefinierte Schnittstelle an,
- versteckt die Implementierungsdetails und veröffentlicht nur das, was explizit festgelegt wird, und
- beschreibt, was es an Abhängigkeiten besitzt.

Vorteile der Modularisierung

Neben der besseren Beherrschbarkeit, den klaren Abhängigkeiten und der reduzierten Komplexität führt die Modularisierung einer Applikation oftmals dazu, dass sich einzelne Module unabhängig von anderen testen und sogar parallel – bei Bedarf durch unterschiedliche Teams – (weiter)entwickeln lassen.

In dieser Hinsicht hilft Modularisierung insbesondere bei mittleren und größeren Projekten die Wartbarkeit und Verständlichkeit sicherzustellen. Aber auch nicht allzu umfangreiche Projekte profitieren von der besseren Nachvollziehbarkeit von Abhängigkeiten. Als Folge findet man weniger Spaghetticode und es reduzieren sich die Wartungsaufwände.

Für die Modularisierungslösung des JDKs gibt es eine weitere Sache zu bedenken: Es lassen sich spezifische minimale Runtime-Images erstellen.

15.1.2 Ziele von Project Jigsaw

Bevor wir uns eingehender mit der Modularisierung beschäftigen, möchte ich nochmals kurz die Ziele von Project Jigsaw und die dadurch adressierten Probleme rekapitulieren.

Problem 1: Typauflösung aus dem `CLASSPATH`

Wünschenswert ist es, Abhängigkeiten explizit beschreiben und überprüfen zu lassen. Eine auf dem `CLASSPATH` basierende Typauflösung macht dies fast unmöglich. Das liegt vor allem daran, dass der `CLASSPATH` aus einer Sammlung von Klassendefinitionen (auch denjenigen in JAR-Dateien) besteht. Jedoch sind die Abhängigkeiten und benötigten Typen nur indirekt über die `import`-Anweisungen beschrieben. Dadurch ist es schwierig, im Vorhinein zu sagen, ob es Probleme aufgrund fehlender Abhängigkeiten oder doppelter Definitionen geben wird. Auch die für Typen genutzte Suchstrategie trägt dazu bei: Im gesamten `CLASSPATH` wird nach einer Klassendefinition von einem benötigten Typ gesucht, bis dieser gefunden wird.

Insgesamt lässt sich feststellen, dass man durch die fehlende explizite Beschreibung von Abhängigkeiten beim Applikationsstart nicht mit Sicherheit feststellen kann, ob ein Artefakt fehlt oder ob gleiche Artefakte in verschiedenen JARs vorliegen, die unterschiedliche Versionsstände repräsentieren.[1] Hier macht sich bemerkbar, dass JAR-Dateien keine Komponenten sind, sondern nur Sammelstellen von Klassendefinitionen.

Problem 2: Monolithisches JDK

Das JDK war bis einschließlich JDK 8 ein riesiger Haufen von Klassen und Interfaces mit einer Vielzahl an Querabhängigkeiten. Dadurch wurde es ungemein erschwert, einzelne Bestandteile herauszulösen und separat bereitzustellen.

[1] Zwar helfen hier Build-Tools wie Maven oder Gradle, aber wenn in Artefakten gleiche Klassen (mit gleichem voll qualifiziertem Namen) definiert sind, so kann es Konflikte geben.

Durch den Trend zu IoT (Internet of Things) und zu kleinen Geräten, die zwar häufig leistungsfähige Prozessoren, jedoch nicht allzu viel Speicher besitzen, ist es wünschenswert und für die Verbreitung von Java wichtig, dort eine Java-Laufzeitumgebung anbieten zu können. Das wird umso einfacher, je modularer das JDK ist, weil dann spezielle, speicherplatzoptimierte Ausprägungen des JDKs genutzt werden können.

Lösung: Module in Java 9

Die genannten Probleme werden durch das Modulkonzept des JDKs und die damit eingeführten Module adressiert. Ein solches Modul hat einen eindeutigen Namen und besteht aus Packages, Klassen und Interfaces. Zudem besitzt ein Modul klar definierte Abhängigkeiten zu anderen Modulen: Dazu erfolgt eine Auflistung zu nutzender Module und exportierter Packages, die durch andere Module zugreifbar sein sollen. Somit kann explizit gesteuert werden, welche anderen Module ein Modul zum Kompilieren und bei der Ausführung benötigt. Im Gegensatz zum `CLASSPATH` wird eine stärkere Kapselung erzielt. Dabei stellen sowohl der Compiler als auch die JVM sicher, dass nur referenzierte Module und freigegebene Typen zugreifbar sind.

15.2 Modularisierung im Überblick

In diesem Unterkapitel werden wir uns die mit Project Jigsaw realisierte Modularisierungslösung des JDKs ansehen. Vorab sei angemerkt, dass damit Module definiert werden können, die eine stärkere Kapselung als JARs ermöglichen. Außerdem besagt das Schlüsselwort `public` nun nicht mehr, dass ein Typ für alle Klassen aus beliebigen anderen Modulen zugreifbar ist.

15.2.1 Grundlagen zu Project Jigsaw

Bevor wir die Modularisierung genauer betrachten, möchte ich darauf hinweisen, dass das JDK an vielen Stellen für die Modularisierung angepasst wurde. Die folgende Aufzählung nennt einige wesentliche Änderungen:

- Die JDK-Verzeichnisstruktur hat sich deutlich geändert, wie dies Abbildung 15-1 zeigt. Insbesondere wird die im JDK enthaltene JRE anders als früher nicht mehr innerhalb des JDKs in einem getrennten Verzeichnis installiert. Allerdings gibt es weiterhin eine getrennte JRE-Installation in einem eigenen Verzeichnis.
- Die Dateien `rt.jar` und `tools.jar` gibt es nicht mehr. Stattdessen existieren jetzt verschiedene Module im Unterverzeichnis `jmods`.
- Auf einige interne Packages, z. B. `sun.misc`, kann nicht mehr (ohne Weiteres) zugegriffen werden.

- Reflection beachtet nun auch die von Modulen vorgegebenen Sichtbarkeitsregeln und ist somit eingeschränkter verwendbar als früher. Unter anderem führt das für Tools und Frameworks zu Problemen, kann aber durch Tricks und Kommandozeilenparameter umgangen werden.
- Der Mechanismus zum Laden von Klassen wurde geändert. Klassen werden standardmäßig nicht mehr im `CLASSPATH` gesucht, sondern im Module-Path. Im `CLASSPATH` angegebene Klassen ermöglichen einen Kompatibilitätsmodus.

Aufgrund dieser Änderungen gab es gewisse Anlaufschwierigkeiten und diverse Applikationen und Tools waren nicht direkt lauffähig. Auf einen Kompatibilitätsmodus und mögliche Varianten einer Migration gehe ich später in Abschnitt 15.5 ein.

Abbildung 15-1 *Strukturänderungen im JDK*

Spracherweiterungen

Die Sprache selbst wurde unter anderem um folgende Schlüsselwörter erweitert:

- `module` – Definiert ein Modul.
- `requires` – Beschreibt die Abhängigkeiten von anderen Modulen.
- `exports` – Legt fest, welche eigenen Packages exportiert werden, d. h. für andere Module sichtbar sind.

Darüber hinaus gibt es noch einige weitere Schlüsselwörter, aber die obigen drei bilden die Basis einer Moduldefinition, auch ***Moduldeskriptor*** genannt. Dieser wird durch eine Datei namens `module-info.java` bereitgestellt:

```
module <ModuleName>
{
    requires <ModuleNameOfRequiredModule>;

    exports <PackageName>;
}
```

Wie zuvor erwähnt, werden Abhängigkeiten von anderen Modulen mit dem Schlüsselwort `requires` beschrieben. Darüber hinaus kann ein Modul mit dem Schlüsselwort

`exports` explizit festlegen, welche Packages nach außen für andere Module zugänglich sein sollen – alle nicht aufgeführten sind von extern nicht zugreifbar. Vereinfachend kann man ein Modul grafisch wie in Abbildung 15-2 darstellen.

Abbildung 15-2 *Schematische Darstellung eines Moduls*

Links ist eine einfache Repräsentation dargestellt. Rechts sieht man eine ausgeklügeltere Variante, die recht gut den Moduldeskriptor widerspiegelt und zudem verdeutlicht, dass der Sourcecode im Inneren des Moduls gekapselt und versteckt ist.

Tipp: Spezielle Schlüsselwörter nur im Moduldeskriptor

Die gerade aufgezählten und auch die später genannten Schlüsselwörter werden innerhalb von Moduldeskriptoren als solche interpretiert, sodass es keine Namenskonflikte mit bestehendem Sourcecode gibt.

Auch die Beziehung zwischen Modulen kann man grafisch gestalten, wie dies Abbildung 15-3 zeigt.

Abbildung 15-3 *Modul 1 greift auf Modul 2 zu.*

Im Zusammenhang mit Abhängigkeitsbeziehungen gibt es die zwei wichtigen Begriffe Readability und Accessibility, die ich nun beleuchte.

Die Begriffe Readability und Accessibility

Dass ein Modul 1 per `requires` auf ein anderes Modul 2 verweist, ist die Voraussetzung dafür, dass es dessen Typen referenzieren kann. Man spricht dann davon, dass Modul 1 das andere Modul 2 liest, oder alternativ, dass Modul 2 für Modul 1 lesbar ist. Per Definition liest jedes Modul sich selbst. Diese ***Readability*** bildet die Grundlage für eine verlässliche Konfiguration (***Reliable Configuration***). Dazu prüft und stellt das mit JDK 9 eingeführte Modulsystem sicher, dass jede Abhängigkeit genau durch ein Modul erfüllt wird und der entstehende Modulgraph azyklisch (zyklusfrei) ist. Zudem wird dafür gesorgt, dass unterschiedliche Module keine Packages gleichen Namens enthalten, sogenannte Split Packages.

Die Readability in Kombination mit den jeweiligen `exports`-Anweisungen sorgen für ***gute Kapselung***, weil wirklich nur die explizit exportierten Packages für andere Module freigegeben sind. Dabei gilt noch die ***Accessibility***. Darunter versteht man Folgendes: Eine Klasse A aus Modul 2 ist nur dann zugreifbar, wenn Modul 1 das korrespondierende Modul 2 liest und Modul 2 zusätzlich das zugehörige Package exportiert.

Benennung von eigenen Modulen

Der Name eines Moduls kann frei vergeben werden, er muss allerdings im Ausführungskontext einer JVM (besser noch global) eindeutig sein. Das lässt sich für Module am einfachsten dadurch erreichen, dass man das Reverse-Domain-Name-Pattern von Packages nutzt. Damit steht der Name jedoch an verschiedenen Stellen für Unterschiedliches: Modulname, Package-Name oder Sourcecode-Verzeichnis. Weil dies anfangs verwirrend sein kann, verzichte ich zur Vermeidung von Mehrfachdeutungen für die einführenden Beispiele des Buchs auf diese Namenskonvention und nutze einfache Namen für Module.

Duplikate in `exports` oder Packages Wie gerade erwähnt, müssen Modulnamen wie auch Package-Namen eindeutig sein. Dies wird vom Compiler sichergestellt. Bei der Definition eines gleichen Packages in unterschiedlichen Modulen kommt es zu Fehlermeldungen wie dieser:

```
DOPPELTE packages in zwei Modulen:
src/jigsaw/timeclient/module-info.java:1:
      error: module timeclient reads package com.server from both
      timeserver and timeserver2
module timeclient
```

Auch ein mehrfacher Export eines Packages (durch Mehrfachangabe eines Packages in verschiedenen Moduldeskriptor-Dateien) wird folgendermaßen angemerkt:

```
DOPPELTE exports:
src/jigsaw/timeserver/module-info.java:7:
      error: duplicate export: com.serverexports com.server to timeclient;
```

Von Oracle empfohlene (aber oftmals ungünstige) Verzeichnisstruktur

Soll eine Applikation in mehrere Module untergliedert werden, so wird von Oracle derzeit empfohlen, ein gemeinsames `src`-Verzeichnis zu nutzen. Pro Modul wird der Sourcecode dann in einem Unterverzeichnis mit dem Namen des Moduls abgelegt. Demgemäß ergibt sich für zwei Module eine Verzeichnisstruktur ähnlich zu folgender:

```
`-- src
    |-- com.inden.module1
    |   |-- com
    |   |   `-- inden
    |   |       `-- module1
    |   |           `-- Application.java
    |   `-- module-info.java
    `-- com.inden.module2
        |-- com
        |   `-- inden
        |       `-- module2
        |           `-- OtherClass.java
        `-- module-info.java
```

Man erkennt folgende wesentliche Punkte:

1. Per Konvention liegt der Moduldeskriptor auf der Hauptebene des Moduls.
2. Unterhalb des Modulverzeichnisses arbeitet man wie gewohnt mit Packages.
3. Etwas irritierend ist das Modulverzeichnis und sein Name. Nutzt man dafür – wie es die Konvention vorsieht – einen ***Reverse Domain Name***, so sind der Modulname und diejenigen der darunterliegenden Packages leicht zu verwechseln.

Beachten Sie unbedingt, dass sich diese Aufteilung für eigene Module in der Regel weniger eignet: Besteht eine Applikation aus mehreren Modulen und sollen diese von verschiedenen Teams realisiert werden, so ist ein gemeinsames Source-Verzeichnis insofern ungünstig, als dadurch erschwert wird, die Module in einzelnen JARs bereitzustellen und möglichst unabhängig voneinander weiterzuentwickeln.

Für die Sourcen des JDKs ist es dagegen praktisch, alle auf einmal kompilieren zu können. Deshalb wurde diese Ein-Source-Verzeichnis-Variante vermutlich von Oracle so eingeführt. Nachfolgend werden wir dieser Struktur folgen. Einerseits um deren Vorteile für kleinere miteinander verbundene Module zu demonstrieren und andererseits um ein paar spezielle Kommandozeilenparameter kennenzulernen.

Erweiterungen im Ökosystem

Um eigene modularisierte Applikationen kompilieren und starten zu können, wurden sowohl der Compiler `javac` als auch die JVM, also das Kommando `java`, angepasst. Beiden kann man nun neue Parameter übergeben:

```
javac --module-path <modulepath> ...

java -p <modulepath> -m <modulename/fully-qualified-class-name>
```

Neu sind folgende Parameter:

- `--module-path` oder kurz `-p` – Statt eines `CLASSPATH` legt dieser Parameter den Module-Path fest, der einem oder mehreren Verzeichnissen entspricht, die Module enthalten.
- `--module` oder kurz `-m` – Spezifiziert das Hauptmodul, ähnlich zur Main-Klasse einer normalen Java-Applikation. Die Notation beginnt mit dem Namen des Moduls gefolgt von einem Schrägstrich und dem voll qualifizierten Klassennamen. Der Name der Hauptklasse kann im Modul selbst hinterlegt werden, dann ist die Angabe beim Aufruf nicht mehr nötig.

Module-Path und `CLASSPATH`

Wir haben bislang kennengelernt, dass Module durch Moduldeskriptoren beschrieben werden und Typen in Packages bereitstellen. Damit eine modularisierte Applikation gestartet werden kann, bedarf es eines Mechanismus, um benötigte Typen zu lokalisieren und zu laden. Dazu dient der Module-Path. Dieser kann exklusiv oder ergänzend zum bis einschließlich Java 8 genutzten `CLASSPATH` angegeben werden.

Erwähnenswert ist, dass man statt des Module-Path weiterhin über `-cp` zur Rückwärtskompatibilität exklusiv nur einen `CLASSPATH` nutzen kann: Damit verhält sich die JVM genau so, wie man es bislang mit JDK 8 gewohnt war,[2] jedoch profitiert man dann nicht von den Vorteilen der Modularisierung.

Der Module-Path ist robuster als der `CLASSPATH`: Das liegt vor allem daran, dass beim `CLASSPATH` innerhalb des gesamten Pfades nach Typen gesucht wird, bis diese gefunden werden. Bei einer Suche im `CLASSPATH` ist nur durch die Import-Anweisungen im Sourcecode verzeichnet, welche Typen benötigt werden. Es ist aber nicht extern und explizit definiert, wie die Abhängigkeiten aussehen. Dadurch kann man im Vorhinein beim Applikationsstart nicht mit Sicherheit feststellen, ob ein Artefakt fehlt. Zudem können auch gleiche Klassen in verschiedenen JARs vorliegen, die unterschiedliche Versionsstände repräsentieren.

Mit dem Module-Path werden Abhängigkeiten grobgranularer auf Modulebene beschrieben anstatt auf Basis von Typen. Es wird sowohl zur Kompilierzeit als auch zur Laufzeit möglich, festzustellen, ob eine Abhängigkeit fehlt oder ob es einen Konflikt durch den Export des gleichen Packages aus verschiedenen Modulen gibt.

Modularisierung des JDKs

Im Rahmen von Project Jigsaw wurde auch das JDK in eine Vielzahl von Modulen aufgeteilt. Dazu wurde das monolithische JDK in einer groß angelegten Aufräumaktion in diverse Module untergliedert.

[2] Allerdings werden Zugriffe auf JDK-Interna geprüft und gegebenenfalls verhindert, etwa auf die Klassen `BASE64Encoder` oder `Unsafe` aus dem Package `sun.misc`.

15.2.2 Beispiel mit zwei Modulen

Nachfolgend betrachten wir zum besseren Verständnis der positiven Auswirkungen der Modularisierung ein Beispiel mit zwei Modulen. Es geht hier vor allem darum, den Einsatz einiger Schlüsselwörter und Kommandos kennenzulernen, um im Verlaufe dieses Kapitels komplexere Szenarien nachvollziehen zu können. Dabei schauen wir uns folgende Themen an:

1. Definition der Module
2. Implementierung der Klassen der Module
3. Kompilieren der Sourcen der Module
4. Anpassungen in den Moduldeskriptoren
5. Kompilieren mit dem Multi-Module Build
6. Starten des modularisierten Programms
7. Auflösen von Implementierungsabhängigkeiten

Zielverzeichnisstruktur im Überblick

Wir wollen zwei Module erzeugen: Das eine nennen wir `jigsawapp` und das andere heißt `services`. Im Modul `jigsawapp` soll sich die Applikation befinden, die für ihre Arbeit auf eine Klasse `MessageService` zugreift. Diese wird im Modul `services` implementiert. Damit ergibt sich folgende Abhängigkeitsbeziehung:

Abbildung 15-4 *Darstellung von Abhängigkeiten der Module des Beispiels*

Für das Beispiel wählen wir folgende Zielverzeichnisstruktur, wobei das Unterverzeichnis `step<N>` dem jeweiligen Schritt entspricht:

```
ch15_2_2_modules_example
`-- step<N>
    `-- src
        |-- jigsawapp
        |   |-- com
        |   |   `-- inden
        |   |       `-- javaprofi
        |   |           `-- MessageExample.java
        |   `-- module-info.java
        `-- services
            |-- com
            |   `-- services
            |       `-- MessageService.java
            `-- module-info.java
```

Nun müssen wir die einzelnen Dateien mit Inhalt füllen. Wir gehen sukzessive vor und beginnen mit den Moduldeskriptoren.

Schritt 1: Definition der Module

Beginnen wir mit der notwendigen Strukturierung im Dateisystem. Bekanntermaßen ist ein Modul ein Verzeichnis, das Packages und Klassen sowie einen Moduldeskriptor enthält. Im Beispiel sollen die Verzeichnisse `jigsawapp` und `services` genutzt werden. Dazu legen wir mit dem Kommando `mkdir` diese beiden Verzeichnisse an:

```
mkdir -p src/jigsawapp
mkdir -p src/services
```

Die Syntax des Kommandos ist betriebssystemabhängig leicht unterschiedlich: Für Linux und Mac OS muss die Option `-p` verwendet werden, damit der gesamte angegebene Verzeichnispfad bzw. die nicht existenten Teile erzeugt werden. Bei Windows ist das nicht nötig. Ich möchte noch darauf hinweisen, dass im gesamten Buch die Windows-Variante nur dann gezeigt wird, wenn diese sich in mehr als nur im Backslash unterscheidet.

Die Moduldeskriptoren Damit der Inhalt eines Verzeichnisses von der JVM (und vom Compiler) als einem Modul zugehörig erkannt wird, bedarf es eines Moduldeskriptors. Dieser besteht aus einer Datei namens `module-info.java`, die beim Kompilieren in die Datei `module-info.class` umgewandelt wird. Dieser Moduldeskriptor legt den Namen des Moduls sowie die davon referenzierten (benötigten) Module und die zu veröffentlichenden Packages fest.[3] In unserem initialen Beispiel nutzen wir eine minimale Definition.

Wir legen zwei rudimentäre Moduldeskriptor-Dateien `module-info.java` in den jeweiligen Unterverzeichnissen mit folgendem Inhalt an:

```
module jigsawapp
{
}
```

Und zudem:

```
module services
{
}
```

Bitte beachten Sie, dass der Name des Verzeichnisses, worin das Modul und dessen Inhalt gespeichert wird, mit dem Namen des Moduls übereinstimmen sollte – zwar kann man von der Konvention abweichen, verliert dadurch aber an Konsistenz und Klarheit.

[3] Diese Datei sollte im Hauptverzeichnis des Moduls hinterlegt werden. Sie kann zwar nahezu beliebig in der Verzeichnishierarchie liegen. Das erschwert jedoch die Nachvollziehbarkeit.

Schritt 2: Implementierung der Klassen der Module

Nach der minimalistischen Definition der beiden Module erzeugen wir zwei Klassen.

Die Klasse `MessageService` Zunächst legen wir das Verzeichnis für das Modul `services` mit

```
mkdir -p src/services/com/services
```

an und beginnen dort mit der Implementierung der Klasse `MessageService`:

```
package com.services;

public class MessageService
{
    public static String generateMessage()
    {
        return "Message from module services!";
    }
}
```

Die Klasse `MessageExample` Nun erstellen wir die Klasse `MessageExample` im Modul `jigsawapp`. Zuvor wird das Verzeichnis für das Modul `jigsawapp` angelegt:

```
mkdir -p src/jigsawapp/com/inden/javaprofi
```

Die Klasse `MessageExample` implementieren wir wie folgt:

```
package com.inden.javaprofi;

import com.services.MessageService;

public class MessageExample
{
   public static void main(final String[] args)
   {
      System.out.println("Generated msg: " + MessageService.generateMessage());
   }
}
```

Im Listing sehen wir, dass die Klasse `MessageService` aus dem Modul `services` verwendet wird. Diese Abhängigkeit zum Modul `services` wird jedoch nicht direkt im Sourcecode sichtbar, sondern nur indirekt über die `import`-Anweisungen und die aufgeführten Packages und Klassen. Demnach sehen wir im Sourcecode nicht, dass wir mit Modulen arbeiten.

Hinweis: Baumdarstellungen von Verzeichnissen mit `tree`

Das Kommando `tree` ist praktisch, um sich Verzeichnisse in einer Baumansicht darstellen zu lassen. Für Mac OS können Sie es wie folgt installieren:

```
brew install tree
```

Für Linux, etwa Ubuntu, nutzt man folgendes Kommando:

```
sudo apt-get install tree
```

Für Windows ist das Kommando schon im Kommandozeileninterpreter vorinstalliert. Hier muss der Aufruf `tree /F` lauten, damit auch die Dateien angezeigt werden. Unter Mac OS kann man wie folgt auch das gewünscht Charset angeben: `tree -charset ascii`

Schritt 3: Kompilieren der Sourcen der Module

Zum Kompilieren geben wir explizit alle Dateien einzeln an – in Schritt 5 lernen wir dann, wie es einfacher geht.

Kompilieren des Moduls `services` Beim Kompilieren beginnen wir mit dem Modul `services`:

```
javac -d build/services \
         src/services/*.java \
         src/services/com/services/*.java
```

Dadurch werden das Java-Programm und die Moduldeskriptor-Datei in das Verzeichnis `build/services` kompiliert. Dabei entsteht eine Spiegelung der Source-Ordner-Hierarchie ähnlich zu der, wie wir sie vom klassischen Kompilieren kennen:

```
ch15_2_2_modules_example/
`-- build
    `-- services
        |-- com
        |   `-- services
        |       `-- MessageService.class
        `-- module-info.class
```

Kompilieren des Moduls `jigsawapp` Nachdem das so reibungslos geklappt hat, versuchen wir das Kompilieren nun folgendermaßen für das Modul `jigsawapp`:

```
javac -d build/jigsawapp \
         src/jigsawapp/*.java \
         src/jigsawapp/com/inden/javaprofi/*.java
```

Durch dieses Kommando kommt es allerdings zu folgenden Fehlermeldungen:

```
src/jigsawapp/com/inden/javaprofi/MessageExample.java:3:
      error: MessageService is not
      visible because package com.services is not visible
import com.services.MessageService;
                   ^
src/jigsawapp/com/inden/javaprofi/MessageExample.java:9:
      error: cannot find symbol
      System.out.println("Generated msg: " + MessageService.msg());
                                                ^
  symbol:   variable MessageService
  location: class MessageExample
2 errors
```

Anhand dieser Ausgaben erkennt man zum ersten Mal, dass beim Einsatz des neuen Modulsystems die Sichtbarkeiten und Zugriffe explizit gesteuert werden müssen und können. Es erfolgt eine viel ***stärkere Einhaltung von Kapselung***, weil die Typen eines Moduls zunächst für die Außenwelt unsichtbar sind, selbst wenn diese `public` definiert sind. Um darauf zugreifen zu können, müssen die Moduldeskriptoren in Form der Dateien namens `module-info.java` geeignet angepasst werden.

Schritt 4: Anpassungen in den Moduldeskriptoren

Offensichtlich nutzt die Klasse `MessageExample` die Klasse `MessageService` und somit hängt das Modul `jigsawapp` vom Modul `services` ab. Um diese Abhängigkeit korrekt festzulegen, müssen wir für das Modul `jigsawapp` mithilfe des in Java 9 neu eingeführten Schlüsselworts `requires` angeben, dass wir das Modul `services` verwenden wollen:

```
module jigsawapp
{
    requires services;
}
```

Zudem müssen wir im Modul `services` die Sichtbarkeit des Packages und seiner enthaltenen Klassen folgendermaßen per `exports` für andere Module freigeben:

```
module services
{
    exports com.services;
}
```

Hinweis: Wissenswertes rund um `module-info.java`

Würden wir nur die Datei `module-info.java` für das Modul `services` wie gezeigt anpassen, so würde zwar die Klasse `MessageService` freigegeben, könnte aber immer noch nicht aus dem Modul `jigsawapp` zugegriffen werden, weil dort keine Abhängigkeit auf das Modul `services` definiert wäre. Denken Sie an die in Abschnitt 15.2.1 vorgestellten Begriffe Readability und Accessibility.

Interessanterweise spezifiziert man durch das Schlüsselwort `requires` Module und beim Export gibt man freizugebende Packages mit `exports` an. Damit ist `import` das Pendant zu `exports`. Dagegen besagt `requires`, durch welches Modul oder welche Module die Imports erfüllt werden sollen.

Schritt 5: Kompilieren mit dem Multi-Module Build

Wir wiederholen das Kompilieren für beide Module. Wiederum ist dies für das Modul `services` erfolgreich und schlägt erneut für das Modul `jigsawapp` fehl, diesmal jedoch mit folgender Fehlermeldung:

```
src/jigsawapp/module-info.java:3: error: module not found: services
    requires services;
             ^
```

Das liegt daran, dass nun das referenzierte Modul nicht gefunden wird. Das können wir einfach dadurch lösen, dass wir beim Kompilieren den Module-Path wie folgt angeben:

```
javac -d build/jigsawapp -p build \
         src/jigsawapp/*.java \
         src/jigsawapp/com/inden/javaprofi/*.java
```

Statt die Modulabhängigkeiten beim Kompilieren explizit anzugeben, ist es oftmals bequemer, die Sourcen aller Module in einem Rutsch zu kompilieren. Dazu muss allerdings das Standardverzeichnislayout genutzt werden. Dann setzt man das neue Compiler-Flag `--module-source-path` folgendermaßen ein:

```
javac -d build --module-source-path src $(find src -name '*.java')
```

Die Syntax funktioniert leider nur unter Linux und Mac OS, weil hier der `find`-Befehl des Kommandozeileninterpreters alle Java-Klassen zusammenträgt.

Für Windows ermöglicht die Powershell folgendes Kommando:

```
javac -d build --module-source-path src $(dir src -r -i '*.java')
```

Durch beide Kommandos entsteht die folgende Struktur im Verzeichnis `build`:

```
ch15_2_2_modules_example
`-- step4-6
    `-- build
        |-- jigsawapp
        |   |-- com
        |   |   `-- inden
        |   |       `-- javaprofi
        |   |           `-- MessageExample.class
        |   `-- module-info.class
        `-- services
            |-- com
            |   `-- services
            |       `-- MessageService.class
            `-- module-info.class
```

Hinweis: Multi-Module Build und Standardverzeichnislayout

Die gezeigte Art zu Kompilieren wird ***Multi-Module Build*** genannt und ermöglicht es, mehrere Module aus dem `src`-Verzeichnis und seinen Unterverzeichnissen auf einmal zu kompilieren. Bei komplexeren modularisierten Systemen wird man aber eine andere Strukturierung vornehmen, um Module unabhängig kompilieren zu können.

Schritt 6: Starten des modularisierten Programms

Abschließend wollen wir die kompilierte Klasse ausführen. Weil wir mit Modulen arbeiten, verändert sich der Aufruf. Statt einzelner Klassen bzw. eines `CLASSPATH` müssen wir nun beim Start der Applikation den Module-Path per `--module-path` sowie mit `-m` das Hauptmodul samt der zu startenden Main-Klasse ähnlich zu einer normalen Java-Applikation angeben. Anstelle des bereits bekannten `--module-path` nutzen wir den Shortcut `-p`:

```
java -p build -m jigsawapp/com.inden.javaprofi.MessageExample
```

Das derart gestartete Programm erzeugt – wie erwartet – folgende Konsolenausgabe:

```
Generated msg: Message from module services!
```

Alle Ungeduldigen, die sich fragen, ob die Main-Klasse nicht im Moduldeskriptor oder irgendwo anders vermerkt werden kann, bitte ich um ein wenig Geduld. Ich komme später darauf zurück.

Schritt 7: Auflösen von Implementierungsabhängigkeiten

Wir haben in den letzten Abschnitten gelernt, wie wir eine Applikation mithilfe von Modulen in kleinere Bestandteile untergliedern können. Das war ein erster Schritt in Richtung Modularisierung. Momentan verweist die nutzende Applikation direkt auf eine Klasse aus einem anderen Modul. ***Das stellt eine stärkere Implementierungsabhängigkeit dar, weil die Module eng miteinander gekoppelt sind***: Sie bilden logisch eigentlich ein Modul. Diese enge Verknüpfung ist in Abbildung 15-5 angedeutet.

Abbildung 15-5 *Ausgangslage*

Idealerweise verweist eine nutzende Applikation lediglich auf Interfaces und erhält die Realisierungen z. B. über Factory-Methoden. Das Zieldesign ist in Abbildung 15-6 gezeigt.

Abbildung 15-6 *Zieldesign zur Entkopplung der Module*

Umsetzung im Modul `services` Zunächst passen wir zur Sichtbarkeitssteuerung den Moduldeskriptor wie folgt an und geben nur das Subpackage `api` frei:

```
module services
{
    exports com.services.api;

    // keine Freigabe von com.services.impl
}
```

Danach implementieren wir das Interface zum Aufruf des Service:

```
package com.services.api;

public interface MessageService
{
    public String generateMessage();
}
```

Es kommt zu einem Namenskonflikt mit dem schon existierenden Service. Als Abhilfe nennen wir die Serviceklasse in `SimpleMessageService` um und verschieben diese in das Package `com.services.impl`. Außerdem implementiert die Serviceklasse das obige Interface `MessageService`:

```
package com.services.impl;

import com.services.api.MessageService;

public class SimpleMessageService implements MessageService
{
    public String generateMessage()
    {
        return "Message from module services!";
    }
}
```

Verbleibt noch ein Detail: Das Package `com.services.impl` wird durch die stärkere Kapselung nicht exportiert und man hat von extern, also anderen Modulen, nur noch auf das Interface `MessageService` Zugriff, aber keinen mehr auf die Klasse `SimpleMessageService`. Um diese zu instanziieren, benötigen wir eine Factory, die wir ebenfalls im veröffentlichten Package `com.services.api` wie folgt erstellen:

```
package com.services.api;

import com.services.api.MessageService;
import com.services.impl.SimpleMessageService;

public class MessageServiceFactory
{
    public static MessageService createMessageService()
    {
        return new SimpleMessageService();
    }
}
```

Umsetzung im Modul `jigsawapp` Kommen wir zur Klasse `MessageExample` zurück. Diese müssen wir nun leicht modifizieren, sodass sie das Interface `MessageService` sowie die Factory `MessageServiceFactory` nutzt:

```
import com.services.api.MessageService;
import com.services.api.MessageServiceFactory;

public class MessageExample
{
   public static void main(final String[] args)
   {
      final MessageService messageService =
                           MessageServiceFactory.createMessageService();
      System.out.println("Generated msg: " + messageService.generateMessage());
   }
}
```

Nach unseren Modifikationen sieht das Verzeichnis wie folgt aus:

```
ch15_2_2_modules_example
`-- step7
    `-- src
        |-- jigsawapp
        |   |-- com
        |   |   `-- inden
        |   |       `-- javaprofi
        |   |           `-- MessageExample.java
        |   `-- module-info.java
        `-- services
            |-- com
            |   `-- services
            |       |-- api
            |       |   |-- MessageService.java
            |       |   `-- MessageServiceFactory.java
            |       `-- impl
            |           `-- SimpleMessageService.java
            `-- module-info.java
```

Kompilieren und Starten Wir kompilieren mit dem Multi-Module Build unter Linux und Mac OS wiederum folgendermaßen:

```
javac -d build --module-source-path src $(find src -name '*.java')
```

Nach wie vor lässt sich das Programm mit

```
java -p build -m jigsawapp/com.inden.javaprofi.MessageExample
```

starten und wir erhalten die bereits bekannte Konsolenausgabe:

```
Generated msg: Message from module services!
```

Fazit Wir sind funktional immer noch auf demselben Stand wie zuvor, jedoch besitzt unser nutzendes Modul keine (starke) Implementierungsabhängigkeit auf wichtige konkrete Klassen mehr – es verbleibt lediglich die Abhängigkeit auf die Factory-Klasse und das Interface. Eine Variante, wie man die Kopplung weiter reduziert, sind Services, die ich in meinem Buch »Java - die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41] im Detail behandle. Eine andere besteht darin, ein Modul mit Interface-Definitionen einzuführen, auf das dann beide Module verweisen. Dies folgt dem Dependecy Inversion Principle und ist exemplarisch in Abbildung 15-7 angedeutet.

Abbildung 15-7 *Dependency Inversion am Beispiel zweier Module A und B*

15.2.3 Packaging

In diesem Abschnitt schauen wir uns das Packaging von modularisierten Applikationen an und nutzen dazu das Verzeichnis `ch15_2_3_packaging_module_example`. Bislang liegen die Module in Form von Verzeichnissen und `.class`-Dateien vor, sind so aber nur schwierig deploybar. Deshalb wollen wir daraus zunächst einmal JARs erstellen.

Einzelne JARs erzeugen

Die Bestandteile unserer Module sollen als Einheit deploybar sein. Dazu ist es ratsam, diese in JAR-Dateien zu bündeln. Dort ergänzen wir zusätzliche Metainformationen, insbesondere die Moduldeskriptoren in Form von `module-info.class`-Dateien. Diese werden im Jigsaw-Modus ausgewertet, um die jeweiligen Sichtbarkeiten und Zugriffe zu steuern. Wenn ein JAR einen Moduldeskriptor enthält, dann spricht man von einem ***modularen JAR***.

Für unsere zwei Module erzeugen wir aus den korrespondierenden Unterverzeichnissen unter `build` zwei modulare JARs namens `jigsawapp.jar` und `services.jar` mit folgenden Kommandos:

```
mkdir lib
jar --create --file lib/jigsawapp.jar -C build/jigsawapp .
jar --create --file lib/services.jar --module-verlision 1.0 -C build/services .
```

Die Parameter bedeuten Folgendes:

- `--create` – Archiv erzeugen[4]
- `--file` – Der Name der zu erzeugenden Archivdatei
- `--module-version` – Modulversionsnummer (optional und rein informativ)
- `-C` – Die Dateien aus dem angegebenen Verzeichnis verwenden

Durch den obigen Aufruf entstehen zwei JAR-Dateien im Verzeichnis `lib`:

```
ch15_2_3_packaging_module_example/
`-- lib
    |-- jigsawapp.jar
    `-- services.jar
```

Zwar haben wir zwei JARs und das Programm lässt sich wieder ähnlich wie zuvor starten, allerdings muss dazu der Pfad auf die Module mit `-p lib` spezifiziert werden:

```
java -p lib -m jigsawapp/com.inden.javaprofi.MessageExample
```

Schöner wäre es, wenn man nur den Modulnamen angeben müsste und automatisch die passende Klasse gestartet würde. Betrachten wir, was dazu notwendig ist.

Executable JAR erzeugen

Wie schon erwähnt, wollen wir ein JAR erzeugen, das ein direktes Starten erlaubt. Dazu nutzen wir folgendes Kommando mit dem Parameter `--main-class`:

```
jar --create --file lib/jigsawapp.jar \
    --main-class com.inden.javaprofi.MessageExample -C build/jigsawapp .
```

Weil dadurch die Information in der Manifest-Datei enthalten ist, können wir den Klassennamen beim Starten des Programms weglassen und lediglich den Modulnamen angeben:

```
java -p lib -m jigsawapp
```

Dieser Aufruf produziert folgende Konsolenausgabe:

```
Generated msg: Message from module services!
```

[4] Es gibt zwar die Kurzform `-c`, die jedoch leicht mit der auch verwendeten Option `-C` zu verwechseln ist. Deshalb wird hier `--create` genutzt.

Hinweis: Beliebige Klassen aus einem modularen JAR starten

Obwohl es oft praktisch ist, den voll qualifizierten Namen der zu startenden Klassen nicht angeben zu müssen, möchte man doch mitunter auch einmal eine andere Klasse aus einem modularen JAR mit einer `main()`-Methode starten. Dies ist möglich, indem man das bereits bekannte Kommando wie folgt nutzt:

```
java -p build -m <modulename>/<fully-qualified-name-of-class-to-run>
```

15.2.4 Abhängigkeiten und Modulgraphen

Nachdem wir mehrere Möglichkeiten der Modularisierung mit Project Jigsaw kennengelernt haben, wollen wir nun unseren Werkzeugkasten vervollständigen. Dazu werden wir das mit JDK 8 eingeführte und mit JDK 9 erweiterte Tool `jdeps` anschauen. Dabei finden die nachfolgend beschriebenen Aktionen im Verzeichnis `ch15_2_4_dependencies_module_example` statt.

Das Tool `jdeps`

Oftmals möchte man einen genaueren Blick auf die Abhängigkeiten werfen. Dabei kann das Tool `jdeps` unterstützen, mit dem sich Abhängigkeiten zwischen Modulen feingranular ermitteln lassen. Das wollen wir nun für die beiden zuvor analysierten modularen JARs anschauen und schreiben Folgendes:

```
jdeps lib/*.jar
```

Das Kommando analysiert die JAR-Dateien und gibt dann Informationen und Abhängigkeiten aus (es wurden zur besseren Lesbarkeit einige Abstände entfernt):

```
jigsawapp
 [file:///.../ch15_2_4_dependencies_module_example/lib/jigsawapp.jar]
   requires mandated java.base (@9-ea)
   requires services
jigsawapp -> java.base
jigsawapp -> services
   com.inden.javaprofi           -> com.services.api           services
   com.inden.javaprofi           -> java.io                    java.base
   com.inden.javaprofi           -> java.lang                  java.base
   com.inden.javaprofi           -> java.lang.invoke           java.base
services
 [file://.../ch15_2_4_dependencies_module_example/lib/services.jar]
   requires mandated java.base (@9-ea)
services -> java.base
   com.services.api              -> com.services.impl          services
   com.services.api              -> java.lang                  java.base
   com.services.impl             -> com.services.api           services
   com.services.impl             -> java.lang                  java.base
```

Anhand der Konsolenausgaben wird ersichtlich, auf welche Packages und Module im Detail zugegriffen wird. Allerdings ist die Konsolenausgabe etwas umfangreicher und

dadurch leicht unübersichtlich. Oftmals informativer ist die Summary-Funktion, die man durch den Kommandozeilenparameter `-s` aktiviert:

```
jdeps -s lib/*.jar
```

Das produziert folgende, übersichtliche Ausgabe:

```
jigsawapp -> java.base
jigsawapp -> services
services -> java.base
```

Grafische Aufbereitung von Abhängigkeitsgraphen Noch anschaulicher ist es, sich diese Informationen grafisch aufbereiten zu lassen. Als Grundlage dazu kann `jdeps` die Abhängigkeiten im DOT-Format, einer textuellen Beschreibung für Graphen, bereitstellen. Dazu nutzt man folgendes Kommando:

```
jdeps -dotoutput graphs lib/*.jar
```

Damit werden verschiedene Dateien im Verzeichnis `graphs` erzeugt:

```
ch15_2_4_dependencies_module_example
`-- graphs
    |-- jigsawapp.dot
    |-- services.dot
    `-- summary.dot
```

Für jedes Modul legt `jdeps` eine eigene `.dot`-Datei an und zudem eine Zusammenfassung in einer Datei `summary.dot`, die für das Beispiel wie folgt aussieht:

```
digraph "summary" {
  "jigsawapp"                  -> "java.base (java.base)";
  "jigsawapp"                  -> "services";
  "services"                   -> "java.base (java.base)";
}
```

Das im Internet unter `http://www.graphviz.org/` frei verfügbare Tool `graphviz` kann aus dem DOT-Format eine grafische Repräsentation z. B. als PNG erzeugen.

Nach der Installation von `graphviz` steht das Kommando `dot` zur Verfügung, mit dem man wie folgt eine Grafik aufbereitet:

```
dot -Tpng -Gdpi=300 graphs/summary.dot > summary.png
```

Der obige Aufruf erzeugt ein PNG-Bild (`-Tpng`) mit einer Auflösung von 300 DPI (`-Gdpi=300`). Das Bild öffnen Sie mit dem Kommando:

```
open summary.png
```

Das Ganze sollte in etwa wie in Abbildung 15-8 aussehen, wobei Sie unter Windows statt `open` das Kommando `start` nutzen müssen.

Abbildung 15-8 *Darstellung von Abhängigkeiten*

Die Tools des JDKs erlauben es, Abhängigkeiten zu analysieren und im DOT-Format zu exportieren. Vor allem bei umfangreichen Projekten ist die grafische Darstellung den textuellen Ausgaben um Längen überlegen.

Hinweis: Installation von `graphviz`

Installieren Sie das Tool für Mac OS folgendermaßen:

```
brew install graphviz
```

Unter Linux nutzt man etwa folgendes Kommando:

```
sudo apt-get install graphviz
```

Für Windows findet sich unter `http://www.graphviz.org/` ein Installer.

15.2.5 Module des JDKs einbinden

Die bislang vorgestellten modularisierten Applikationen waren recht einfach und beschränkten sich insbesondere auf JDK-Basisfunktionalität aus dem Modul `java.base`. In der Praxis wird man neben diesen auch diverse andere Bestandteile des JDKs sowie vor allem Fremdbibliotheken, wie beispielsweise Google Guava, einsetzen. Auf das Einbinden von Fremdbibliotheken gehe ich später ein. In diesem Abschnitt wollen wir zunächst kennenlernen, wie man Abhängigkeiten zu Modulen des JDKs beschreibt und dass dies nach exakt denselben Regeln abläuft wie bei eigenen Modulen.

Als Beispiel dient eine Applikation mit einem GUI. Es soll die aktuelle Uhrzeit angezeigt werden und eine Protokollierung von Aufrufen erfolgen. Dazu wollen wir im Anschluss die für Swing und zur Protokollierung mit einem `java.util.Logger` benötigten Module einbinden.

Abhängigkeiten und Zielverzeichnisstruktur im Überblick

In diesem Beispiel besteht die Applikation wieder aus zwei eigenen Modulen. Dabei befindet sich die Applikation selbst im Modul `timeclient`, das für die Ermittlung der Uhrzeit auf die Klasse `TimeInfo` zugreift. Diese wird durch ein separates Modul `timeserver` bereitgestellt. Damit ergeben sich die in Abbildung 15-9 dargestellten High-Level-Abhängigkeiten – später schauen wir uns das Ganze detaillierter an.

Abbildung 15-9 *Vereinfachte Darstellung von Abhängigkeiten*

Für das Beispiel ergibt sich für die einzelnen Schritte folgende Zielverzeichnisstruktur:

```
ch15_2_5_include_jdk_modules_example
`-- step<N>
    `-- src
        |-- timeclient
        |   |-- com
        |   |   `-- client
        |   |       `-- CurrentTimeSwingExample.java
        |   `-- module-info.java
        `-- timeserver
            |-- com
            |   `-- server
            |       `-- TimeInfo.java
            `-- module-info.java
```

Das Anlegen der Verzeichnisse geschieht mit folgenden Kommandos:

```
mkdir -p src/timeclient/com/client
mkdir -p src/timeserver/com/server
```

Nun müssen wir die einzelnen Dateien mit Inhalt füllen. Wir gehen sukzessive vor und beginnen mit den Moduldeskriptoren.

Schritt 1: Definition der Module

Wir legen zwei `module-info.java`-Dateien in den jeweiligen Unterverzeichnissen an. Für das Modul `timeclient` schreiben wir:

```
module timeclient
{
    requires timeserver;
}
```

Für das Modul `timeserver` notieren wir Folgendes:

```
module timeserver
{
    exports com.server;
}
```

Dadurch sind die Abhängigkeiten und Sichtbarkeiten zwischen den beiden Modulen korrekt beschrieben. Die Abhängigkeiten zu den Modulen des JDKs ergänzen wir sukzessive im Verlaufe der nächsten Abschnitte.

Schritt 2: Implementierung der Klassen der Module

Nach der initialen Definition der Moduldeskriptoren erzeugen wir zwei Klassen.

Klasse des Time-Servers Wir beginnen mit der Implementierung der Klasse `TimeInfo` im Modul `timeserver`. Diese soll das aktuelle Datum inklusive Uhrzeit mithilfe der Klasse `LocalDateTime` ermitteln sowie ihren eigenen Aufruf mit einem `Logger` protokollieren. Das Ganze implementieren wir wie folgt:

```
package com.server;

import java.time.LocalDateTime;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TimeInfo
{
    public static LocalDateTime getCurrentTime()
    {
        Logger.getGlobal().log(Level.INFO, "getCurrentTime() called");
        return LocalDateTime.now();
    }
}
```

Klasse des Time-Clients Den zugehörigen Client realisieren wir als Swing-Applikation in der Klasse `CurrentTimeSwingExample`. In der Methode `main()` erstellen wir ein einfaches GUI, das wichtigste Element ist ein Label zur Zeitanzeige. Um die aktuelle Zeit zu ermitteln, wird auf die Klasse `TimeInfo` zugegriffen.

```
package com.client;

import com.server.TimeInfo;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class CurrentTimeSwingExample
{
    public static void main(final String[] args)
    {
        final JFrame appFrame = new JFrame("CurrentTimeSwingExample");
        final JLabel label = new JLabel("Now: " +  TimeInfo.getCurrentTime());

        appFrame.getContentPane().add(label);
        appFrame.setSize(300, 50);
        appFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        appFrame.show();
    }
}
```

Die Implementierung ist damit abgeschlossen und wir machen uns ans Kompilieren.

Schritt 3: Kompilieren der Sourcen der Module

Zum Kompilieren nutzen wir einen Multi-Module Build und folgendes Kommando:

```
javac -d build --module-source-path src $(find src -name '*.java')
```

Dabei erhalten wir eine Vielzahl an Fehlern (hier leicht gekürzt):

```
src/timeserver/com/server/TimeInfo.java:4: error: package java.util.logging is
      not visible
import java.util.logging.Level;
                ^
  (package java.util.logging is declared in module java.logging, but module
        timeserver does not read it)
src/timeserver/com/server/TimeInfo.java:5: error: package java.util.logging is
      not visible
import java.util.logging.Logger;
                ^
  (package java.util.logging is declared in module java.logging, but module
        timeserver does not read it)
src/timeclient/com/client/CurrentTimeSwingExample.java:5: error: package javax.
      swing is not visible
import javax.swing.JFrame;
            ^
  (package javax.swing is declared in module java.desktop, but module timeclient
         does not read it)
src/timeclient/com/client/CurrentTimeSwingExample.java:6: error: package javax.
      swing is not visible
import javax.swing.JLabel;
            ^
```

```
  (package javax.swing is declared in module java.desktop, but module timeclient
        does not read it)
```

Wie zu erwarten, zeigen die Fehlermeldungen, dass wir die JDK-Module zu Swing (`java.desktop`) und zum Logging (`java.logging`) einbinden müssen. Praktischerweise bietet der Compiler mittlerweile eine gute Hilfestellung, indem er zu den jeweiligen Imports bzw. Typen auch direkt das benötigte Modul aufführt. Möchte man jedoch schon vorab das für eine Klasse zugehörige Modul ermitteln, so gibt es dazu momentan keine direkte Möglichkeit auf Toolebene, aber ein Blick in die Javadoc ist diesbezüglich zielführend.

Aus dem Gesagten wird auch verständlich, warum abweichend zu anderen Listings für die vorherigen Beispiele die `imports` aufgeführt sind.

Schritt 4: Anpassungen in den Moduldeskriptoren

Anhand der Fehlermeldungen kann man die zugehörigen und benötigten Module des JDKs ermitteln. Basierend darauf ergänzen wir den Moduldeskriptor des Moduls `timeclient` wie folgt:

```
module timeclient
{
    requires java.desktop;

    requires timeserver;
}
```

Im Modul `timeserver` muss auf das JDK-Modul `java.logging` verwiesen werden:

```
module timeserver
{
    requires java.logging;

    exports com.server;
}
```

Schritt 5: Erneutes Kompilieren

Nach diesen Korrekturen in den `module-info.java`-Dateien ist das Kompilieren mit demselben Kommando wie zuvor erfolgreich:

```
javac -d build --module-source-path src $(find src -name '*.java')
```

Schritt 6: Starten des modularisierten Programms

Wir wollen die modularisierte Applikation mit dem bekannten Aufruf im Unterverzeichnis `step4-6` starten:

```
java -p build -m timeclient/com.client.CurrentTimeSwingExample
```

Neben der Protokollierung des Methodenaufrufs auf der Konsole wird die Swing-Applikation gestartet, die den aktuellen Zeitpunkt ausgibt, wie es Abbildung 15-10 zeigt.

Abbildung 15-10 *Darstellung des Programms* CURRENTTIMESWINGEXAMPLE

Schritt 7: Analyse der Abhängigkeiten

Zum Abschluss wollen wir noch schauen, welcher Abhängigkeitsgraph durch unsere Erweiterungen in den Moduldeskriptoren entsteht. Dazu muss man nicht einmal JARs erzeugen, um das Tool `jdeps` zu nutzen. Es reicht, wenn die Module – wie in diesem Beispiel – in kompilierter Form vorliegen. Mit folgenden drei Befehlen im Unterverzeichnis `step7` können wir eine grafische Repräsentation aufbereiten:

```
jdeps --module-path build -dotoutput graphs build/timeclient build/timeserver
dot -Tpng -Gdpi=300 graphs/summary.dot > summary.png
open summary.png
```

Das letzte Kommando öffnet eine Darstellung wie in Abbildung 15-11 gezeigt.

Abbildung 15-11 *Darstellung der Abhängigkeiten im Detail*

Fazit

Anhand eines Beispiels wurde dargestellt, dass das Einbinden von JDK-Modulen nach exakt denselben Regeln abläuft wie bei eigenen Modulen. Um zu den gewünschten, zu importierenden Klassen das korrespondierende JDK-Modul zu ermitteln, kann man die informativen Fehlermeldungen des Compilers nutzen.

15.2.6 Arten von Modulen

Im Verlauf dieses Kapitels haben wir bereits einige Arten von Modulen kennengelernt, aber nicht explizit benannt. Das möchte ich an dieser Stelle nachholen:

- **Named Platform Modules** – Bekanntermaßen wurde im Rahmen von Project Jigsaw auch das JDK in Module unterteilt. Dort findet man rund 80 Module, die Sie sich mit dem Kommando `java --list-modules` auflisten lassen können. Diese speziellen Module des JDKs haben keinen Zugriff auf den Module-Path.
- **Named Application Modules** – Auch Named Application Modules haben wir schon in verschiedenen Beispielen selbst erstellt. Darunter werden Module verstanden, die Anwendungen oder Bibliotheken bündeln. Im Speziellen sind dies modulare JARs, also solche, die in ihrem JAR einen Moduldeskriptor in Form der Datei `module-info.class` enthalten. Diese können auf alle über den Module-Path erreichbaren Module zugreifen, nicht jedoch auf Klassen aus dem `CLASSPATH`.
- **Open Modules** – Wie die beiden ersten Module, allerdings geben Open Modules alle Packages für Reflection nach außen frei. Mit dem Schlüsselwort `opens` im Moduldeskriptor bzw. dem Kommandozeilenparameter `--add-opens` kann dies auch auf spezifische Packages eingeschränkt erfolgen. Für Details verweise ich Sie an mein Buch »Java - die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41].
- **Automatic Modules** – Für die Migration von Anwendungen ist es von Vorteil, dass man auch gewöhnliche JAR-Dateien, also ohne `module-info.class`, im Module-Path angeben kann. Diese JAR-Dateien werden zu sogenannten Automatic Modules: Dabei wird der JAR-Dateiname ohne Versionsnummer und Endung als Modulname genutzt. Named Application Modules können auf Automatic Modules per `requires` zugreifen. Insbesondere exportieren Automatic Modules alle ihre Packages und können auf sämtliche Module aus dem Module-Path sowie JARs aus dem `CLASSPATH` zugreifen.
- **Unnamed Modules** – Ergänzend zum Module-Path kann man sowohl beim Kompilieren als auch beim Programmstart einen `CLASSPATH` angeben. Alle dort vorhandenen Typen werden zu dem sogenannten Unnamed Module zusammengefasst. Enthält der `CLASSPATH` modulare JARs, so werden diese wie normale JARs ohne Modularisierung behandelt: Sie exportieren alle enthaltenen öffentlichen Typen. Somit können alle öffentlichen Typen aus einem Unnamed Module direkt aufeinander zugreifen, also ohne Sichtbarkeitsschutz. Zudem gilt Folgendes: Automatic Modules können auf das Unnamed Module zugreifen, aber Named Application Modules können dies nicht. Dieser Sachverhalt ist für die schrittweise Migration wichtig. So kann man ein benötigtes JAR als Automatic Module einbinden und dessen Abhängigkeiten aus dem `CLASSPATH` erfüllen.

Nach diesem einführenden Blick auf die verschiedenen Arten von Modulen schauen wir uns später in Abschnitt 15.5 mögliche Migrationsszenarien an und lernen dabei weitere Details zu den Modularten kennen.

15.3 Sichtbarkeiten und Zugriffsschutz

Die Themen Sichtbarkeit und Zugriffsschutz wurden zuvor nicht explizit thematisiert, sondern eher am Rande besprochen. Beiden Themen wollen wir uns nun widmen.

15.3.1 Sichtbarkeiten

Eingangs nannte ich als eines der Ziele bei der Modularisierung eine bessere Steuerung von Abhängigkeiten und Sichtbarkeiten. Die bisherigen Beispiele haben gezeigt, dass seit Java 9 die Sichtbarkeiten strenger als in Java 8 geprüft werden. Rekapitulieren wir zunächst den Status quo in Java 8, bevor ich die Erweiterungen von Java 9 vorstelle.

Sichtbarkeiten bis JDK 8

Bis Java 8 besaßen Typen eine der folgenden vier Sichtbarkeiten:

- `private` – Nur in der eigenen Klasse sichtbar
- *default* / *package private* (kein Schlüsselwort) – Nur im eigenen Package sichtbar
- `protected` – Wie *package private*, aber auch in abgeleiteten Klassen sichtbar
- `public` – Aus allen Packages zugreifbar

Tatsächlich bedeutet `public`, dass ein solcher Typ im `CLASSPATH` für alle anderen Typen zugänglich ist. Demnach kann man – zumindest für `public` – nicht wirklich von einer Sichtbarkeitssteuerung sprechen.

Sichtbarkeiten seit JDK 9

Mit der Einführung der Modularisierung lässt sich die Sichtbarkeit von Typen genauer spezifizieren. Relevant ist dies in der Regel nur für die als `public` definierten Typen. Diese untergliedern sich seit JDK 9 wie folgt:

- **Global** – `public` für alle Module – Wenn das entsprechende Package einer `public` definierten Klasse mit `exports` zum Zugriff freigegeben wurde, ist diese von allen anderen Modulen zugreifbar, die auf dieses Modul per `requires` verweisen.
- **Eingeschränkt** auch ***Qualified Export*** genannt – `public` für angegebene Module – Mithilfe von `exports to` kann eine Liste von Modulen spezifiziert werden, die Zugriff erhalten sollen. Das erfordert zudem, dass andere Module auf dieses Modul per `requires` verweisen.
- **Modul intern** – `public` im Modul selbst – Sofern Klassen in Packages liegen, die nicht in `exports` aufgeführt werden, sind diese Klassen nur aus den Packages des eigenen Moduls zugreifbar.

Den ersten und dritten Fall haben wir schon in verschiedenen Beispielen kennengelernt: Auf eine `public` definierte Klasse konnte nicht von anderen Modulen aus zugegriffen

werden. Dies war erst möglich, nachdem mit `requires` die Abhängigkeit und damit der Zugriffswunsch sowie mit `exports` die öffentliche Bereitstellung beschrieben wurden. Der zweite Fall ist ein Spezialfall des ersten. Allgemein geht es darum, dass man das Exportieren von Klassen auf eine vordefinierte Menge an Modulen beschränken kann. Das benötigt man z. B., wenn man innerhalb von Modulen eines Frameworks Zugriff erlauben möchte, diese Klassen aber von Nutzern des Frameworks nicht zugreifbar sein sollen. Auf diese Weise wurden im JDK spezielle, nicht für das öffentliche API vorgesehene Funktionalitäten nur intern bereitgestellt.

Ein Bild sagt mehr als tausend Worte. Somit sollte sich Ihr Verständnis nach einem Blick auf die folgende Abbildung 15-12 deutlich verbessern.

Abbildung 15-12 *Auswirkungen der drei Sichtbarkeiten bei* `public`

Einsatz im JDK Zuvor hatte ich angedeutet, dass es manchmal wünschenswert ist, bestimmte Typen bzw. Packages dediziert lediglich für spezielle Module zugreifbar zu machen. Im JDK selbst gilt dies auch für das Package `sun.reflect`. Dieses soll nur für diejenigen Module des JDKs verfügbar sein, die explizit in `exports` hinter dem Schlüsselwort `to` aufgeführt sind:

```
module java.base
{
     exports sun.reflect to java.corba,
                            java.logging,
                            java.sql,
                            java.sql.rowset,
                            jdk.scripting.nashorn;
}
```

15.3.2 Zugriffsschutz und Reflection

Bei den Sichtbarkeiten gilt, dass Packages explizit mit `exports` nach außen veröffentlicht und mit `requires` Module referenziert werden müssen. Nachfolgend schauen wir uns zur Vertiefung und zur Ergänzung das Thema Zugriffsschutz und Reflection an. Für das Beispiel sei das Verzeichnis wie folgt aufgebaut:

```
ch15_3_2_accessibility
`-- src
    |-- timeclient
    |   |-- com
    |   |   `-- client
    |   |       `-- CurrentTimeExample.java
    |   `-- module-info.java
    `-- timeserver
        |-- com
        |   `-- server
        |       |-- TimeInfo.java
        |       `-- internal
        |           `-- InternalUtil.java
        `-- module-info.java
```

Zugriffsschutz für interne Packages

Betrachten wir das Thema Zugriffsschutz: In der obigen Applikation soll im Modul `timeserver` die Klasse `InternalUtil` aus dem Package `com.server.internal` nach außen nicht sichtbar sein, sondern lediglich das Package `com.server`. Deswegen wird das interne Package in `exports` nicht aufgeführt:

```
module services
{
    exports com.server;
}
```

Zugriffsschutz und Reflection

Bis einschließlich Java 8 lässt sich mit Reflection so ziemlich jedes Sicherheitsmerkmal von Java umgehen. Beispielsweise konnte man sich mit `setAccessible(true)` selbst auf private Attribute und Methoden Zugriff verschaffen. Es war sogar möglich, wenn auch sicher nicht ratsam, als `final` definierte Attribute nachträglich noch mit einem anderen Wert zu versehen. Reflection war also ein wenig wie ein Zauberstab, mit dem man viel Magie betreiben konnte, insbesondere weil sich die sonst üblichen Regeln außer Kraft setzen lassen.

Deswegen sollten wir uns fragen, welche Auswirkungen die Modularisierung auf die Zugriffsmöglichkeiten per Reflection hat. Also konkret, ob sich mit Reflection auch die durch die Modularisierung eingeschränkten Zugriffe umgehen lassen.

Die Klasse `CurrentTimeExample` ändern wir so ab, dass dort mit Reflection ein Zugriff auf die Klasse `InternalUtil` und deren Methode `getCurrentTime()` erfolgt:

```
public class CurrentTimeExample
{
   public static void main(final String[] args) throws Exception
   {
      System.out.println("Trying to access getCurrentTime() via reflection");

      final ClassLoader classLoader = CurrentTimeExample.class.getClassLoader();

      final Class<?> internalClass = classLoader.
            loadClass("com.server.internal.InternalUtil");

      final Method method = internalClass.getMethod("getCurrentTime");
      final Object result = method.invoke(null, new Object[0]);

      System.out.println("getCurrentTime() returned: " + result);
   }
}
```

Das Programm CURRENTTIMEEXAMPLE lässt sich mit folgenden Kommandos kompilieren und starten:

```
javac -d build --module-source-path src $(find src -name '*.java')
java -p build -m timeclient/com.client.CurrentTimeExample
```

Dadurch kommt es in etwa zur folgenden Konsolenausgabe:

```
Trying to access getCurrentTime() via reflection
Exception in thread "main" java.lang.IllegalAccessException: class com.client.
      CurrentTimeExample (in module timeclient)
      cannot access class com.server.internal.InternalUtil (in module timeserver)
       because module timeserver does not export com.server.internal to
      module timeclient
```

Wir sehen, dass trotz des erlaubten Zugriffs auf die Methode `getCurrentTime()` per `getMethod()` und des zurückgelieferten validen `Method`-Objekts ein Aufruf per `invoke()` durch eine `IllegalAccessException` verhindert wird.

Hinweis: Zugriffsschutz mit `--add-exports` aushebeln

Mitunter benötigt man Zugriff auf Interna. In begründeten Fällen kann durch den Kommandozeilenparameter `--add-exports` eine explizite Freigabe erfolgen – das sollte aber wirklich nur in Ausnahmefällen eingesetzt werden. Durch Folgendes ist dann auch der Aufruf per Reflection erlaubt und es wird die Methode `getCurrentTime()` ausgeführt:

```
java -p build --add-exports timeserver/com.server.internal=timeclient
     -m timeclient/com.client.CurrentTimeExample
```

15.4 Empfehlenswertes Verzeichnislayout für Module

Bereits bei der Einführung zur Modularisierung habe ich in Abschnitt 15.2.1 angedeutet, dass das von Oracle für Module empfohlene Verzeichnislayout für die meisten eigenen Applikationen eher ungünstig ist.

Zur Erinnerung sei hier nochmals das Verzeichnislayout dargestellt: Unterhalb eines gemeinsamen `src`-Ordners sind die jeweiligen Module mit gleichnamigem Unterverzeichnis beheimatet – hier exemplarisch für die Module `personprocessor` und `utilities` gezeigt:

```
person_processor_application
`-- src
    |-- personprocessor
    |   |-- com
    |   |   |-- domain
    |   |   |   `-- Person.java
    |   |   `-- application
    |   |       `-- PersonProcessingApplication.java
    |   `-- module-info.java
    `-- utilities
        |-- com
        |   `-- utils
        |       `-- SomeUtil.java
        `-- module-info.java
```

Während dieses Verzeichnislayout für ganz spezifische Anwendungsfälle, etwa zum Kompilieren des JDKs, hilfreich ist, empfiehlt sich für eigene Module aus verschiedenen Gründen ein Verzeichnislayout, das sich an dem Standard von Maven und Gradle orientiert. Diese Thematik schauen wir uns nun genauer an.

Schwachstellen des Oracle-Verzeichnislayouts

Warum ist dieses Oracle-Verzeichnislayout nicht immer optimal? Das hat unter anderem folgende Gründe:

- Durch das gemeinsame Source-Verzeichnis mit jeweils Unterverzeichnissen pro Modul gestaltet es sich recht schwierig, Module unabhängig voneinander zu entwickeln, zu kompilieren und in JARs zu verpacken.
- Beim Oracle-Verzeichnislayout lässt sich durch das Unterverzeichnis pro Modul kein normaler Maven- oder Gradle-Build nutzen, da die Sourcen dann nicht wie standardmäßig erwartet in `src/main/java` liegen. Dadurch müssen die Source- und Test-Verzeichnisse für jedes Projekt bzw. Modul angepasst werden. Ähnliches gilt für IDEs. Insgesamt wird das Handling dadurch unkomfortabel.
- Ebenfalls erschwert sich die Zuordnung der Abhängigkeiten von Fremdbibliotheken zu den entsprechenden Modulen.

Für die Praxis empfohlenes Verzeichnislayout

In diversen kleineren eigenen Projekten kam ich im Laufe der letzten Jahre zur Erkenntnis, dass es sich anbietet, für jedes Modul ein separates Verzeichnis mit eigenem Source-Verzeichnis zu nutzen. IntelliJ, Maven und Gradle erlauben dazu Multi-Module-Projekte, die ein übergeordnetes Hauptverzeichnis vorsehen. In Eclipse muss man dagegen mehrere eigenständige Projekte nutzen, die aber idealerweise auf einem Multi-Module Build von Maven basieren.

Nachfolgend gezeigtes Verzeichnislayout verdeutlicht das zuvor Gesagte: Module sind in eigenen Verzeichnissen untergebracht, hier für die Module `personprocessor` und `utilities` gezeigt. Darunter existiert jeweils ein Source-Verzeichnis pro Modul. Zudem wird statt `src` der Standard `src/main/java` genutzt:

```
person_processor_application
|-- personprocessor
|   `-- src
|       `-- main
|           `-- java
|               |-- com
|               |   |-- domain
|               |   |   `-- Person.java
|               |   `-- application
|               |       `-- PersonProcessingApplication.java
|               `-- module-info.java
`-- utilities
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- utils
                |       `-- SomeUtil.java
                `-- module-info.java
```

Empfohlenes Verzeichnislayout für die nachfolgenden Beispiele

In diesem Kapitel habe ich in den Beispielen bislang ein Verzeichnislayout genutzt, das sich an die Vorgaben von Oracle hält.

Oben habe ich auf dessen Schwachstellen hingewiesen und motiviert, wie eine Alternative dazu aussehen kann. Diese werde ich für die nachfolgenden Beispiele verwenden. Daraus ergeben sich die Vorteile, dass sich dieses Verzeichnislayout einfacher und ohne Konfigurationsaufwand mit Build-Tools und IDEs verarbeiten lässt. Zudem wird dadurch die Unabhängigkeit von Modulen gefördert.

15.5 Kompatibilität und Migration

Bislang haben wir Module selbst definiert oder auf Module des JDKs zugegriffen. Es gibt dabei jedoch zwei Punkte zu bedenken:

1. **Rückwärtskompatibilität** – Applikationen, die ohne das Modulsystem erstellt wurden, sollten weiterhin lauffähig sein, um einen möglichst reibungslosen Übergang von Java 8 auf 9, 11 oder gar 15 zu ermöglichen.
2. **Einbinden von Fremdbibliotheken** – Jede etwas größere Java-Applikation verweist immer auch auf Fremdbibliotheken, die eingebunden werden müssen, die aber vermutlich noch nicht modularisiert wurden.

Den ersten Punkt behandeln wir direkt im Anschluss. Auf das Einbinden von Fremdbibliotheken gehe ich dann in verschiedenen Abschnitten ein und betrachte mehrere Migrationsszenarien. Dazu schauen wir uns die bereits kurz vorgestellten Konzepte Unnamed Module und Automatic Modules genauer an.

15.5.1 Kompatibilitätsmodus

Früher wurde bei jeder neuen Java-Version dem Erhalten der Rückwärtskompatibilität große Aufmerksamkeit geschenkt. Die Modularisierung ist erstmalig ein Feature, das zu Inkompatibilitäten führt. Zur schrittweisen Umstellung und zur Rückwärtskompatibilität bietet das Modulsystem jedoch einen ***Kompatibilitätsmodus***, bei dem Applikationen wie bisher aus JARs zusammengesetzt sein können und trotzdem die Module des JDKs nutzbar sind.

Wie bereits beschrieben, werden Typen durch das Modulsystem über den Module-Path aufgelöst. Doch was passiert, wenn dies nicht erfolgreich ist? Dann kommt es zu einem Fehler beim Applikationsstart. Jedoch erlaubt es das Modulsystem, ergänzend zum Module-Path einen `CLASSPATH` anzugeben, in dem die JVM zusätzlich nach Typen suchen kann. Durch die reine Angabe von Abhängigkeiten im `CLASSPATH` lassen sich Applikationen exakt so starten wie zuvor mit JDK 8 auch.

Wie schon aus den bisherigen Beispielen bekannt, finden die Aktionen diesmal im Verzeichnis namens `ch15_5_1_migration_compatibility_mode` statt – nachfolgend markiere ich das jeweilige Verzeichnis fett und verzichte in der Regel auf eine explizite Nennung im Text.

Beispiel: Applikation nur im `CLASSPATH`

Oftmals werden Sie neben Klassen des JDKs auch externe Bibliotheken einbinden, etwa die verbreitete Bibliothek Google Guava in Form der Datei `guava-29.0-jre.jar`, die unter `https://mvnrepository.com/artifact/com.google.guava/guava` heruntergeladen werden kann.

Exemplarisch betrachten wir eine Klasse `UseGuavaFromClassPathExample`, die die Klasse `com.google.common.base.Joiner` aus Google Guava zum Verketten von Strings nutzt:

```
package com.inden.javaprofi;

import com.google.common.base.Joiner;

public class UseGuavaFromClassPathExample
{
    public static void main(final String[] args)
    {
        joinWithGuava("Guava", null, "From", "ClassPath");
    }

    public static void joinWithGuava(final String... strings)
    {
        final Joiner joiner = Joiner.on(", " ).skipNulls();

        System.out.println(joiner.join(strings));
    }
}
```

Die obige Klasse ist in nachfolgend gezeigter Verzeichnishierarchie definiert, wobei wir auch erkennen, dass die Datei `guava-29.0-jre.jar` in einem Verzeichnis `externallibs` parallel zum Hauptverzeichnis des Moduls abgelegt ist:

```
jigsaw_ch15
|-- externallibs
|    `-- guava-29.0-jre.jar
|
`-- ch15_5_1_migration_compatibility_mode
    `-- src
        `-- main
            `-- java
                `-- com
                    `-- inden
                        `-- javaprofi
                            `-- UseGuavaFromClassPathExample.java
```

Kompilieren und Starten Versuchen wir das Ganze zu kompilieren und geben dabei über `-cp` einen `CLASSPATH` an:

```
javac -d build -cp ../externallibs/guava-29.0-jre.jar \
      $(find src -name '*.java')
```

Es entsteht die bekannte Spiegelung des `src`-Ordners im `build`-Ordner:

```
ch15_5_1_migration_compatibility_mode/
`-- build
    `-- com
        `-- inden
            `-- javaprofi
                `-- UseGuavaFromClassPathExample.class
```

Die beim Kompilieren entstandene Datei `UseGuavaFromClassPathExample` lässt sich durch Angabe der Abhängigkeit auf `guava-29.0-jre.jar` im `CLASSPATH` wie folgt starten:

```
java -cp ../externallibs/guava-29.0-jre.jar:build \
     com.inden.javaprofi.UseGuavaFromClassPathExample
```

Beachten Sie dabei, dass wir unsere kompilierten Sourcen über den `CLASSPATH` und das Verzeichnis `build` einbinden. Der obige Aufruf produziert folgende Konsolenausgabe:

```
Guava, From, ClassPath
```

Packaging Schließlich erzeugen wir ein JAR namens `myjarneedsguava.jar`:

```
mkdir lib

jar --create --file lib/myjarneedsguava.jar -C build .
```

Das ergibt folgenden Verzeichnisinhalt:

```
ch15_5_1_migration_compatibility_mode
`-- lib
    `-- myjarneedsguava.jar
```

Dieses JAR lässt sich dann mit folgendem Kommando starten:

```
java -cp ../externallibs/guava-29.0-jre.jar:lib/myjarneedsguava.jar \
     com.inden.javaprofi.UseGuavaFromClassPathExample
```

Auch hier nutzen wir den `CLASSPATH` sowohl für Guava als auch für unsere Klasse aus dem JAR namens `myjarneedsguava.jar`.

Fazit

Wir haben gesehen, dass wir Programme wie bisher gewohnt kompilieren und paketieren können, obwohl das JDK bereits modularisiert ist. Dieser exklusive Einsatz des `CLASSPATH` für die Applikationsbestandteile ist eine Variante, Rückwärtskompatibilität zu erhalten. Doch was geschieht eigentlich, wenn wir unser Programm modularisieren und Module verwenden wollen?

15.5.2 Migrationsszenarien

Wenn wir unsere Programme in mehrere Module untergliedern, dann gibt es dabei mitunter den einen oder anderen Stolperstein, wenn Klassen aus externen Bibliotheken eingebunden werden müssen. Wie wir schrittweise vorgehen können und welche Migrationsszenarien existieren, betrachten wir nun. Dabei lernen wir sowohl das Unnamed Module als auch Automatic Modules und deren Eigenschaften genauer kennen.

Unnamed Module

Das sogenannte Unnamed Module ist ein künstliches Modul: Jeder im `CLASSPATH` gefundene Typ wird dem ***Unnamed Module*** zugeordnet. Dieses besitzt spezielle Eigenschaften, um Migrationen zu erleichtern: Zunächst einmal liest das Unnamed Module alle anderen Module. Dadurch kann aus Typen des `CLASSPATH` auf alle in beliebigen Modulen enthaltenen Typen zugegriffen werden. Das ist insbesondere für die vom JDK exportierten Packages und Typen wichtig. Auf diese Weise sind alle Applikationen, die auf dem `CLASSPATH`-Mechanismus basieren, wie bisher lauffähig.

Eine Migration wird außerdem dadurch erleichtert, dass das Unnamed Module alle seine Packages exportiert. Um trotz des `CLASSPATH` ein Mindestmaß einer verlässlichen Konfiguration (***Reliable Configuration***) sicherzustellen, ist aus Named Modules kein Zugriff auf das Unnamed Module, also den `CLASSPATH`, gestattet. Mehr noch: Named Modules können keine Abhängigkeiten auf das Unnamed Module definieren und somit auch nicht besitzen.[5]

Automatic Modules

Bei einer Umwandlung einer bestehenden Applikation in ein modularisiertes Programm benötigt man ab und an aber doch Zugriff aus Modulen auf den `CLASSPATH`. Für diesen Fall existieren die sogenannten ***Automatic Modules***. Diese erlauben es einem konventionellen JAR, wie ein Named Module zu erscheinen, jedoch mit der Möglichkeit, auf Typen aus dem Unnamed Module und damit dem `CLASSPATH` zugreifen zu können.

Wie entsteht ein solches Automatic Module? Ganz einfach dadurch, dass wir das JAR im Module-Path aufführen. Das Modulsystem erkennt dies und erzeugt anhand des JAR-Namens automatisch ein korrespondierendes implizites Named Module. Bitte beachten Sie die Hinweise im folgenden Praxistipp.

Weil Automatic Modules keinen Moduldeskriptor enthalten, kann man im Vorhinein nicht wissen, von welchen Modulen ein Automatic Module abhängig ist. Zur leichteren Migration wird automatisch durch den Compiler bzw. die JVM ein künstlicher Moduldeskriptor mit `requires` »all« und `exports` »all« generiert. Dadurch kann ein Automatic Module alle Named Modules lesen. Darüber hinaus ist nicht vorhersehbar, welche Packages exportiert werden sollen. Daher werden der obigen Argumentation

[5] Das ist logisch, da ja die Referenzierung über den Namen erfolgt. Demnach gilt: Kein Name, keine Referenzierbarkeit!

folgend von einem Automatic Module alle Packages exportiert.[6] Schließlich bietet ein Automatic Module allen anderen Automatic Modules eine Implied Readability (transitive Propagation von Abhängigkeiten) – auch diese erleichtert die Migration, sie sollte bei einer möglichen späteren Konvertierung in ein Named Module aber genauer geprüft und vermutlich eingeschränkt werden.

Tipp: Wissenswertes zu Automatic Modules

Die Referenzierung bestehender JARs als Automatic Modules im Moduldeskriptor funktioniert nur dann, wenn die Namen in ein erwartetes Schema passen. Beispielsweise sind Minuszeichen innerhalb des Namens eines JARs problematisch, sofern sie nicht direkt vor der Versionsnummer stehen. Das betrifft beispielsweise das JAR von MongoDB (`mongo-java-driver-3.12.7.jar`). Dafür ist es nicht erlaubt, Folgendes im Moduldeskriptor zu schreiben:

```
// Achtung: Leider ist dies so syntaktisch nicht erlaubt
requires mongo-java-driver;
```

Zwar kann man für diese Fälle die Namen der JAR-Dateien von Hand anpassen und die Minuszeichen entfernen. Das führt aber zu recht langen und unleserlichen Namen und bereitet Schwierigkeiten bei Versionswechseln. Empfehlenswert ist es, für die Minuszeichen im Dateinamen einen Punkt (.) in dem per `requires` angegebenen Namen zu nutzen, weil dies vom Modulsystem unterstützt wird:

```
requires mongo.java.driver;
```

Es wäre wohl deutlich einfacher gewesen, wenn man die Namen von Modulen im Moduldeskriptor in Anführungszeichen angeben könnte, aber es gibt eine elegantere Abhilfe, die wir jetzt betrachten.

Abhilfe durch `Automatic-Module-Name` in `MANIFEST.MF` Für die gerade beschriebenen möglichen Schwierigkeiten bezüglich der automatischen Ableitung des Namens eines Moduls aus demjenigen des JARs gibt es folgende Abhilfe:[a] Der Name eines Automatic Module lässt sich mithilfe des Attributs `Automatic-Module-Name` in der Datei `MANIFEST.MF` des JARs festlegen. Das vermeidet Probleme, falls der Namen des JARs vom erwarteten Schema abweicht. Eine Beschreibung zu den Mechanismen beim Auflösen der Namen eines Automatic Module finden Sie in der Dokumentation zum Interface `ModuleFinder`: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/module/ModuleFinder.html#of(java.nio.file.Path...)`.

[a] Diese habe ich zufällig beim Wechseln auf eine neuere Guava-Version entdeckt: Für Guava Version 24 hat sich der Name des JARs auf `guava-24.0-jre-jar` geändert. Damit schlägt der Automatismus und ein Verweis per `requires guava` fehl.

[6] Das erleichtert zwar eine Migration einer bestehenden in eine modularisierte Applikation, jedoch sollte man sich auch bewusst sein, dass so durchaus als intern gedachte Packages nach außen sichtbar und zugänglich werden.

Bottom-up-Migration und Migrationsszenarien

Um die nachfolgend vorgestellten Migrationsszenarien leichter nachvollziehen zu können, ist es wichtig, zu wissen, welche Zugriffe aus welchen Arten von Modulen erlaubt sind. Abbildung 15-13 zeigt dies, wobei die jeweils relevante Modulart leicht dunkler eingefärbt ist.

Abbildung 15-13 *Zugriffsmöglichkeiten aus unterschiedlichen Modularten*

Bei einer sogenannten Bottom-up-Migration werden JARs schrittweise in modulare JARs umgewandelt. Dabei kann man folgende zwei Szenarien unterscheiden:

1. **Bottom-up-Migration und Named Modules** – Wenn wir den Sourcecode eines JARs im Zugriff haben, können wir ein JAR in ein modulares JAR überführen, indem wir einen geeigneten Moduldeskriptor hinzufügen und das Modul neu kompilieren und paketieren.
2. **Bottom-up-Migration und Automatic Modules** – Besteht kein Zugriff auf den Sourcecode des JARs, so lässt sich das JAR nicht auf die zuvor beschriebene Weise in ein modulares JAR konvertieren. Als gute Alternative bleibt die Nutzung als Automatic Module, gegebenenfalls in Kombination mit dem Unnamed Module.

Stellen wir uns für eine Migration vor, eine Applikation bestehe aus verschiedenen JARs, die im `CLASSPATH` aufgeführt werden, etwa aus den JARs `app.jar`, `A.jar`, `B.jar` und `C.jar`.

Bottom-up-Migration und Named Modules Zunächst scheint die Bottom-up-Migration mit Named Modules recht einfach, weil man lediglich einen Moduldeskriptor erstellen muss, damit ein modulares JAR entsteht. Dabei gibt es allerdings zwei Hürden: Zunächst einmal gestaltet es sich einigermaßen aufwendig, herauszufinden, welche Packages exportiert werden sollen. Zudem können wir ein JAR nur dann in ein modulares JAR umwandeln, wenn dieses keine Abhängigkeiten auf andere, nicht modulare JARs des `CLASSPATH` besitzt, weil sich diese nicht im Moduldeskriptor referenzieren lassen. Das zeigt Abbildung 15-14.

Abbildung 15-14 *Bottom-up-Migration und Named Modules*

Die gerade geschilderte Bottom-up-Migration mit der Umwandlung einzelner JARs in Named Modules ist vielfach nicht möglich, weil es Abhängigkeiten von einem Named Module auf eine Klasse vom `CLASSPATH` (also das Unnamed Module) oder aber ein Automatic Module gibt. Außerdem kann man – wie in unserem Beispiel – nur diejenigen JARs, die man unter eigener Kontrolle hat, in Named Modules konvertieren.

Zur Erinnerung sei nochmals Folgendes erwähnt: Interessanterweise und migrationserleichternd dürfen JARs aus dem `CLASSPATH` durchaus Abhängigkeiten auf Module besitzen: Das Unnamed Module liest alle Module.

Bottom-up-Migration und Automatic Modules Für viele Anwendungsfälle ist die Bereitstellung von JARs als Automatic Modules leichter und besser geeignet als der zuvor beschriebene Ansatz. Das gilt vor allem für externe Bibliotheken ohne Zugriff auf die Sourcen.

Abbildung 15-15 *Bottom-up-Migration und Automatic Modules*

Abbildung 15-15 verdeutlicht, dass sich nicht nur die beiden JARs `A.jar` und `B.jar` in Automatic Modules überführen lassen, sondern auch `C.jar`. Tatsächlich ist dies bei Wunsch sogar auch für `D.jar` möglich.

15.5.3 Fallstrick bei der Bottom-up-Migration

Bei der Bottom-up-Migration gibt es noch einen Punkt zu bedenken: Man sollte mit den JARs starten, die keine oder kaum Abhängigkeiten besitzen. Allerdings ist man intuitiv versucht, die Umwandlung mit dem eigenen Applikations-JAR zu beginnen. Das ist aber ein Irrweg. Um das zu verstehen, nutzen wir das vorangegangene Beispiel als Basis. Für die Transformation unserer Applikation in ein Modul benötigen wir einen Moduldeskriptor sowie eine leicht abweichende Verzeichnisstruktur wie folgt – für die Beispiele soll das `guava-29.0-jre.jar` im Verzeichnis `externallibs` liegen:

```
jigsaw_ch15
|-- externallibs
|   `-- guava-29.0-jre.jar
|
`-- ch15_5_3_pitfall_bottomup_migration
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- inden
                |       `-- javaprofi
                |           `-- UseGuavaFromClassPathInNamedModuleExample.java
                `-- module-info.java
```

Schritt 1: Definition des Moduldeskriptors

Bereits beim Erstellen des Moduldeskriptors würden wir stutzen und uns fragen, wie wir die Abhängigkeit zu Guava angeben sollen. Das wollen wir kurz außer Acht lassen. Weil hier die Applikation beschrieben wird, exportieren wir keine Packages und der Moduldeskriptor enthält somit lediglich die Moduldefinition:

```
module bottomup_unnamed_example
{
}
```

Schritt 2: Implementierung

Exemplarisch betrachten wir die aus dem letzten Beispiel leicht abgewandelte und in `UseGuavaFromClassPathInNamedModuleExample` umbenannte Klasse, die wiederum die Klasse `Joiner` aus Guava nutzt:

```
public class UseGuavaFromClassPathInNamedModuleExample
{
    public static void main(final String[] args)
    {
        final Joiner joiner = Joiner.on(", ").skipNulls();

        System.out.println(joiner.join("Guava", null, "As", null, "Unnamed"));
    }
}
```

Schritt 3: Kompilieren

Zum Kompilieren geben wir über `-cp` einen `CLASSPATH` für Guava an:

```
javac -d build -cp ../externallibs $(find src -name '*.java')
```

Im Gegensatz zu der reinen `CLASSPATH`-Variante erhält man folgende Kompilierfehler, die verdeutlichen, dass aus unserem Modul kein Zugriff auf die Klasse `Joiner` besteht:

```
src/main/java/com/inden/javaprofi/UseGuavaFromClassPathInNamedModuleExample.java
      :3: error: package com.google.common.base does not exist
import com.google.common.base.Joiner;
                             ^
src/main/java/com/inden/javaprofi/UseGuavaFromClassPathInNamedModuleExample.java
      :22: error: cannot find symbol
        final Joiner joiner = Joiner.on(", ").skipNulls();
              ^
  symbol:   class Joiner
  location: class UseGuavaFromClassPathInNamedModuleExample
src/main/java/com/inden/javaprofi/UseGuavaFromClassPathInNamedModuleExample.java
      :22: error: cannot find symbol
        final Joiner joiner = Joiner.on(", ").skipNulls();
                             ^
  symbol:   variable Joiner
  location: class UseGuavaFromClassPathInNamedModuleExample
3 errors
```

Fazit

Wie man sieht, scheitert der Versuch, aus einem Named Module auf Typen aus dem `CLASSPATH` zuzugreifen. Dieser vermeintliche Migrationspfad schlägt fehl, weil, wie schon einmal kurz erwähnt, aus einem Named Module nicht auf das Unnamed Module (`CLASSPATH`) zugegriffen werden kann!

15.5.4 Beispiel: Migration mit Automatic Modules

Wir lernen nun, wie man den gerade beschriebenen Fehler bei Migrationen vermeiden kann. Dazu wird die eigene Applikation als Named Module realisiert und Fremdbibliotheken werden als Automatic Modules eingebunden, hier für das JAR `guava-29.0-jre.jar` gezeigt. Das Verzeichnislayout sieht wie folgt aus:

```
jigsaw_ch15
|-- externallibs
|   `-- guava-29.0-jre.jar
|
`-- ch15_5_4_migration_automatic_modules
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- inden
                |       `-- javaprofi
                |           `-- AutomaticModuleExample.java
                `-- module-info.java
```

Schritt 1: Definition des Moduldeskriptors

Zur Referenzierung von Guava in unserem Moduldeskriptor muss man wissen, dass bis einschließlich Guava-Version 23 die Automatik des Modulsystems greift, die aus dem Namen des JARs den Namen des Automatic Module herleitet. Ab Version 24 wird der Modulname aus der Angabe des Attributs `Automatic-Module-Name` in der Datei `MANIFEST.MF` des JARs ermittelt. Somit ergeben sich folgende Varianten:

```
module automatic_module_example
{
    // requires guava;            // Bis guava-23.0.jar
    requires com.google.common;   // Ab guava-24.0-jre.jar
}
```

Schritt 2: Implementierung der nutzenden Applikation

Unsere Beispielapplikation besteht erneut nur aus einer einzigen Klasse, die zur Konkatenation einiger Strings die Klasse `Joiner` aus Guava nutzt:

```
package com.inden.javaprofi;

import com.google.common.base.Joiner;

public class AutomaticModuleExample
{
    public static void main(final String[] args)
    {
        final Joiner joiner = Joiner.on(", ").skipNulls();

        System.out.println(joiner.join("Guava", null, "As", null, "AUTOMATIC"));
    }
}
```

Schritt 3: Kompilieren

Versuchen wir das übliche Kommando zum Kompilieren:

```
javac -d build $(find src -name '*.java')
```

Wie aufgrund der per `requires` angegebenen, aber im Module-Path nicht aufgeführten Abhängigkeit zu erwarten war, kommt es zu folgender Fehlermeldung:

```
src/automatic_module_example/module-info.java:3:
      error: module not found: com.google.common
    requires com.google.common; // Ab guava-24.0-jre.jar
```

Zur Behebung müssen wir das Verzeichnis `externallibs`, in dem Guava liegt, folgendermaßen beim Kompilieren im Module-Path angeben:

```
javac -d build --module-path ../externallibs $(find src -name '*.java')
```

Schritt 4: Starten

Schließlich können wir das Programm wie folgt starten:

```
java --module-path ../externallibs:build \
     -m automatic_module_example/com.inden.javaprofi.AutomaticModuleExample
```

Das Programm erzeugt erwartungskonform folgende Konsolenausgabe:

```
Guava, As, AUTOMATIC
```

15.5.5 Beispiel: Automatic und Unnamed Module

Bisher haben wir nur den einfachsten Fall betrachtet, dass Automatic Modules in sich abgeschlossen waren und keine Typen anderer JARs bzw. Module außer denjenigen des JDKs benötigte. Wie gehen wir aber vor, wenn Querabhängigkeiten aus dem Automatic Module auf andere Typen aus dem `CLASSPATH` bestehen?

Nachfolgend nutzen wir JARs und Module in verschiedenen Modularten und der Aufrufhierarchie »Application Module → Automatic Module → Unnamed Module«:

- Wir erstellen eine einfache Applikation als Named Application Module.
- Diese soll das JAR `myjarneedsguava.jar`, das in Abschnitt 15.5.1 zum Kompatibilitätsmodus erstellt wurde, als Automatic Module einbinden.
- Innerhalb von `myjarneedsguava.jar` wird Funktionalität aus Google Guava verwendet. Deshalb muss diese Bibliothek zur Laufzeit im Zugriff sein. Wir wollen hier den `CLASSPATH`, also das Unnamed Module, nutzen.

Dazu kopieren wir ausgehend vom fett markierten Hauptverzeichnis das zuvor in Abschnitt 15.5.1 erstellte JAR `myjarneedsguava.jar` mit folgendem Kommando:

```
cp ../ch15_5_1_migration_compatibility_mode/lib/* ../ownlibs/
```

Dadurch sieht die Verzeichnisstruktur folgendermaßen aus:

```
jigsaw_ch15
|-- externallibs
|   `-- guava-29.0-jre.jar
|--ownllibs
|   `-- myjarneedsguava.jar
|
`-- ch15_5_5_migration_automatic_and_unnamed_modules
    `-- src
        `-- main
             `-- java
                 |-- com
                 |   `-- inden
                 |       `-- javaprofi
                 |           `-- auto_uses_unnamed
                 |               `-- AutomaticAndUnnamedModuleExample.java
                 `-- module-info.java
```

Schritt 1: Erstellen des Moduldeskriptors

Nun wollen wir das JAR `myjarneedsguava.jar` als Automatic Module einbinden. Dazu notieren wir die Abhängigkeit im Moduldeskriptor:

```
module auto_unnamed_modules_example
{
    requires myjarneedsguava;
}
```

Schritt 2: Implementierung der Applikation

Die folgende Klasse `AutomaticAndUnnamedModuleExample` nutzt Funktionalität aus unserem JAR:

```
package com.inden.javaprofi.auto_uses_unnamed;

import com.inden.javaprofi.UseGuavaFromClassPathExample;

public class AutomaticAndUnnamedModuleExample
{
    public static void main(final String[] args) throws Exception
    {
        System.out.println("AutomaticAndUnnamedModuleExample"));

        UseGuavaFromClassPathExample.joinWithGuava("Guava", "From ClassPath",
                                                   "Referenced By",
                                                   "Automatic Module");
    }
}
```

Hinweis: Notwendigkeit eindeutiger Packages

Bitte beachten Sie, dass wir unsere Klasse in einem Package mit einem eindeutigen Namen – hier das Subpackage `auto_uses_unnamed` – definieren müssen, wenn wir das bereits erstellte JAR einbinden wollen. Der Grund ist folgender: Das JAR enthält bereits das Package `com.inden.javaprofi` und das Modulsystem erlaubt keine Split Packages, also gleichnamige Packages in verschiedenen Modulen.

Schritt 3: Kompilieren

Das Kompilieren geschieht mit

```
javac -d build --module-path ../ownlibs  $(find src -name '*.java')
```

wodurch ein Ordner `build` mit den kompilierten Sourcen entsteht.

Schritt 4a: Starten

Wir wissen, dass die Applikation eine transitive Abhängigkeit besitzt: Das als Modul eingebundene JAR `myjarneedsguava.jar` benötigt eine Abhängigkeit auf `guava-29.0-jre.jar`. Wir starten das Programm trotzdem mal wie folgt:

```
java -p build:../ownlibs  \
     -m auto_unnamed_modules_example/com.inden.javaprofi.auto_uses_unnamed.
          AutomaticAndUnnamedModuleExample
```

Das führt zu folgender Ausgabe:

```
AutomaticAndUnnamedModuleExample
Exception in thread "main" java.lang.NoClassDefFoundError:
      com/google/common/base/Joiner
    at myjarneedsguava/com.inden.javaprofi.UseGuavaFromClassPathExample.
          joinWithGuava(UseGuavaFromClassPathExample.java:22)
    at auto_unnamed_modules_example/com.inden.javaprofi.auto_uses_unnamed.
          AutomaticAndUnnamedModuleExample.main(AutomaticAndUnnamedModuleExample.
          java:18)
Caused by: java.lang.ClassNotFoundException: com.google.common.base.Joiner
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(
          BuiltinClassLoader.java:583)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(
          ClassLoaders.java:178)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
    ... 2 more
```

Zunächst sieht man den Text `AutomaticAndUnnamedModuleExample`. Das zeigt, dass unser Named Application Module geladen und ausgeführt wird. Jedoch kommt es beim Abarbeiten des als Automatic Module eingebundenen JARs (das wiederum eine indirekte Abhängigkeit auf Google Guava besitzt) zu einem Fehler, da die Klasse `Joiner` nicht geladen werden kann, was wiederum zum `NoClassDefFoundError` führt.

Schritt 4b: Starten mit zusätzlichem `CLASSPATH`

Demnach ist der Start nicht erfolgreich, weil die externe Bibliothek Google Guava nicht gefunden wird. Deren Verfügbarkeit war beim Kompilieren nicht wichtig, da hierbei die Abhängigkeit nur aus dem als Bytecode vorliegenden JAR besteht. Zur Laufzeit wird die Klasse `Joiner` und somit das korrespondierende JAR aber benötigt.

Fügen wir beim Start der Applikation den `CLASSPATH` hinzu:

```
java -p build:../ownlibs -cp ../externallibs/guava-29.0-jre.jar \
     -m auto_unnamed_modules_example/com.inden.javaprofi.auto_uses_unnamed.
          AutomaticAndUnnamedModuleExample
```

Dann kommt es zu folgenden Ausgaben:

```
AutomaticAndUnnamedModuleExample
Guava, From ClassPath, Referenced By, Automatic Module
```

Fazit

Selbst an diesem einfachen Beispiel erkennt man, wie fragil das Arbeiten mit dem `CLASSPATH` ist. Bei einer vollständig modularisierten Applikation würde das Problem einer fehlenden Abhängigkeit direkt schon beim Kompilieren und nicht erst zur Laufzeit ersichtlich werden.

Aus dem bisher Gesagten wird aber auch deutlich, dass Automatic Modules in Kombination mit dem Unnamed Module einen sinnvollen, gangbaren Migrationspfad vorgeben: JARs aus Fremdbibliotheken können wir zumindest so lange als Automatic Modules einbinden, bis diese in einer modularen Variante vorliegen.

15.5.6 Beispiel: Abwandlung mit zwei Automatic Modules

Im vorherigen Beispiel wurde ein externes JAR noch in Form des Unnamed Module genutzt. Weil dies keine zuverlässige Konfiguration erlaubt, wäre es wünschenswert, wenn wir stattdessen ein Automatic Module nutzen könnten. Betrachten wir das Ganze in folgendem Verzeichnislayout:

```
jigsaw_ch15
|-- externallibs
|   `--  guava-29.0-jre.jar
|-- ownllibs
|   `-- myjarneedsguava.jar
|
`-- ch15_5_6_migration_two_automatic_modules
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- inden
                |       `-- javaprofi
                |           `-- onlyauto
                |               `-- OnlyAutomaticModulesExample.java
                `-- module-info.java
```

Schritt 1: Erstellen des Moduldeskriptors

Unser eigenes Modul sowie Guava binden wir wie folgt als Automatic Module ein:

```
module two_automatic_modules_example
{
    requires myjarneedsguava;
    requires com.google.common;
}
```

Schritt 2: Implementierung

Damit wäre bereits alles Wesentliche umgestellt, im Speziellen sind keine Modifikationen an den Klassen nötig. Weil wir aber einen anderen Text ausgeben wollen, ändern wir die zuvor genutzte Klasse `AutomaticAndUnnamedModuleExample` minimal ab und nennen sie in `OnlyAutomaticModulesExample` um:

```
package com.inden.javaprofi.onlyauto;

import com.inden.javaprofi.UseGuavaFromClassPathExample;

public class OnlyAutomaticModulesExample
{
    public static void main(final String[] args) throws Exception
    {
        System.out.println("OnlyAutomaticModulesExample");

        UseGuavaFromClassPathExample.joinWithGuava("Both", "As",
                                                   "Automatic Module");
    }
}
```

Schritt 3: Kompilieren und Starten

Beim Kompilieren wird lediglich der Module-Path verwendet. Das Ganze geschieht mit folgendem Kommando:

```
javac -d build --module-path ../ownlibs:../externallibs \
       $(find src -name '*.java')
```

Schritt 4: Starten

Der Start des Programms ist nun nur unter Angabe des Module-Path möglich:

```
java -p build:../ownlibs:../externallibs \
     -m two_automatic_modules_example/com.inden.javaprofi.onlyauto.
           OnlyAutomaticModulesExample
```

Dabei kommt es zu folgender Ausgabe:

```
OnlyAutomaticModulesExample
Both, As, Automatic Module
```

Fazit

Wir haben gesehen, wie einfach sich bestehende JARs in Automatic Modules überführen lassen. Das kann Migrationen erleichtern. Demzufolge bieten Automatic Modules oftmals eine Möglichkeit zu einer flexiblen Migration.

15.5.7 Fazit

Wir haben verschiedene Migrationsszenarien sowie mögliche Probleme betrachtet. Zusammenfassend lässt sich sagen, dass eine Migration einer bestehenden Applikation, die sich aus verschiedenen JARs zusammensetzt, gut überlegt werden sollte. Zwar erreicht man eine bessere technische Struktur, jedoch schafft man dadurch oftmals kaum oder keinen Business-Nutzen. Wenn man sich dennoch für eine Migration einer bestehenden Applikation entscheiden sollte, dann erfolgt diese am besten schrittweise JAR für JAR, um sofort auf mögliche Probleme reagieren zu können.

Für eine Migration bieten sich folgende Schritte an:

1. **Keine Abhängigkeiten** – Man beginnt mit JARs ohne externe Abhängigkeiten. Sofern deren Sourcecode zur Verfügung steht, lassen sich diese per Bottom-up-Migration in Named Modules überführen – bei Bedarf können diese immer noch als Automatic Modules eingebunden werden.
2. **Mit Abhängigkeiten** – Vorhandene JARs ohne Zugriff auf deren Sourcecode lassen sich problemlos als Automatic Modules nutzen.
3. **Eigene Applikation** – Nachdem die JARs entweder in Module überführt wurden oder aber im Unnamed Module verbleiben, kann man versuchen, die eigene Applikation in ein Named Application Module zu transformieren.

Wenn man bezüglich der Migration noch ein wenig unerfahren ist – was auf Sie nach der Lektüre dieses Kapitels nicht mehr zutreffen sollte ;-) –, kann man notfalls immer noch auf den Kompatibilitätsmodus zurückgreifen.

Hinweis: Mögliche Schwierigkeiten bei Migrationen

Bei der Migration einer bestehenden Applikation existieren mitunter (wenn auch nicht wünschenswert) zyklische Abhängigkeiten zwischen den Typen zweier JARs. Wollte man diese JARs in Module überführen und würde man dazu in den Moduldeskriptoren `A requires B` und `B requires A` vermerken, käme es dann direkt beim Kompilieren zu einem Fehler:

```
src/timeclient/module-info.java:3: error:
      cyclic dependence involving timeserver
    requires timeserver;
             ^
```

Für das Aufbrechen von Zyklen haben wir Services kennengelernt. Alternativ kann man ein Modul erstellen, das lediglich die benötigten Interfaces enthält. Etwas Ähnliches wurde in Abschnitt 15.2.2 besprochen.

15.6 Zusammenfassung

Dieses Kapitel hat einen Einstieg in das Thema Modularisierung gegeben. Durch den vorgestellten Themenmix sollten Sie einen guten ersten Eindruck gewonnen haben und fit für erste eigene Experimente sein.

Allerdings gibt es noch diverse weitere Dinge zu entdecken und auszuprobieren. Da wären z. B. Services oder die Besonderheiten und neuen Möglichkeiten mit Reflection. Diese leicht fortgeschrittenen Themen behandle ich in meinem Buch »Java - die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41].

V Fallstricke und Lösungen im Praxisalltag

16 Bad Smells

In diesem Kapitel werfen wir einen Blick auf verbreitete Fallstricke und Probleme, die einem Entwickler immer mal wieder begegnen. Man spricht in diesem Zusammenhang auch von »Bad Smells«. Das bezeichnet Abschnitte des Sourcecodes, die im übertragenen Sinne einen schlechten Geruch verbreiten – an denen potenziell etwas »faul« ist. Dieser eingängige Begriff wurde von Kent Beck und Martin Fowler im Buch »Refactoring« [20] zur Beschreibung derartiger Programmabschnitte eingeführt.

In diesem Kapitel werden Sie erfahren, welche Tücken und Fehler sich leicht in den Sourcecode einschleichen. Wenn wir diese Bad Smells erkennen, lokalisieren und verstehen können, sind wir auch in der Lage, Gegenmaßnahmen zu ergreifen. In den folgenden Abschnitten präsentiere ich dazu einen Katalog von Bad Smells, der sich in die Abschnitte 16.1 »Programmdesign«, 16.2 »Klassendesign« und 16.3 »Fehlerbehandlung und Exception Handling« gliedert und diverse Sourcecode-Auszüge unterschiedlicher Qualität zeigt. Wir analysieren jeweils die darin enthaltenen Probleme und schulen damit Ihr Auge. Das Erkennen eines Problems ist gut, allerdings sollten wir auch in der Lage sein, es zu beheben. Daher werden zu jedem Bad Smell passende Tipps zur Vermeidung gegeben und kleinere Abhilfemaßnahmen sofort vorgestellt. Allgemeingültige Umbaumaßnahmen stelle ich separat in Kapitel 17 »Refactorings« vor. Den Abschluss dieses Kapitels bilden in Abschnitt 16.4 »Häufige Fallstricke« einige Programmiertücken, die nicht die Schwere eines Bad Smells besitzen, aber dennoch hier erwähnt werden sollen, um diese in eigenen Programmen zu vermeiden.

Bad Smells am Beispiel

Bekanntermaßen beschreiben Bad Smells Sourcecode-Abschnitte, an denen möglicherweise etwas nicht in Ordnung ist. Diese Programmteile vergrößern die Gefahr für Fehlfunktionen oder enthalten bereits Fehler – manchmal sind solche Sourcecode-Stellen aber auch akzeptabel. In letzterem Fall sollte man dies durch einen Kommentar beschreiben.

Mit ein wenig Übung springen beim Analysieren von Sourcecode potenziell gefährliche Stellen schnell ins Auge. Außerdem ist es in der Regel so, dass Bad Smells gehäuft auftreten. Zum besseren Verständnis betrachten wir ein Beispiel mit der folgenden Methode `setDataState(int, OperationState)`:

```
public void setDataState(final int department, final OperationState state)
{
    if (this.dataState == null)
    {
        this.dataState = new HashMap<Integer, OperationState>();
    }
    this.dataState.put(new Integer(department), state);

    if (log.isInfoEnabled())
    {
        String msg = "device data state for department " + department + ": ";
        switch (state)
        {
            case RUNNING:
                msg += "running";
                break;
            case GOING_DOWN:
                msg += "going down";
                break;
            case DOWN:
                msg += "down";
                break;
            case GOING_UP:
                msg += "going up";
                break;
            default:
                msg += "unknown";
        }
        log.info(msg);
    }
}
```

Diese Methode scheint zunächst durchaus in Ordnung. Allerdings offenbaren sich dem geübten Auge hier unter anderem folgende Schwachstellen:

1. **Fehlende Plausibilitätsprüfungen der Parameter** – Es lauern Probleme durch unerwartete Parameterwerte: Nur ein kleiner Teil des `int`-Wertebereichs des Parameters `department` entspricht gültigen Werten. Außerdem ist für den Parameter `state` der Wert `null` ungültig und sollte zurückgewiesen werden.
2. **Unpassender Einsatz von Lazy Initialization** – Die Gefahr für mögliche Multithreading-Probleme wird durch den unsynchronisierten Einsatz von Lazy Initialization[1] für das Attribut `dataState` erhöht. Tatsächlich traten diese Probleme in der Praxis auf und führten zu Inkonsistenzen in den gespeicherten Werten.
3. **Unausgewogenheit beim Logging** – Der Logging-Code dominiert die Funktionalität: Nur die ersten Programmzeilen tragen zur Anwendungsfunktionalität bei. Die restlichen Zeilen bereiten Log-Informationen auf und sollten in eine Methode herausfaktoriert werden, um die Struktur klarer erkennbar zu machen. Ergänzend bietet es sich an, das `switch-case` durch eine Lookup-Map (vgl. Abschnitt 6.1.10) zu ersetzen.

[1] Lazy Initialization beschreibt die Technik, ein Attribut nicht direkt zu initialisieren, sondern zunächst mit `null` zu belegen, um die Konstruktion komplexer Objekte zu vermeiden. Erst beim ersten Zugriff erfolgt dann eine Initialisierung. Kapitel 22 beschreibt diese Technik im Detail.

16.1 Programmdesign

Nach diesem einführenden Beispiel stelle ich nun einen Katalog von Bad Smells vor und beginne dabei mit möglichen Problemen im Programmdesign.

16.1.1 Bad Smell: Verwenden von Magic Numbers

Oftmals kommen an beliebigen Stellen im Sourcecode verschiedene Zahlenwerte vor, die explizit als Zahlenliteral – als sogenannte ***Magic Number*** – angegeben werden. Zum Teil existieren funktionale und semantische Abhängigkeiten zwischen diesen Werten, die jedoch nicht im Sourcecode, sondern bestenfalls durch Kommentare ausgedrückt werden können.

Wir betrachten hier eine Klasse `CommandExecutor`, die im Konstruktor einen `int`-Wert erhält, der steuert, wie Kommandos verwaltet und einer internen Datenstruktur hinzugefügt werden. Für diesen Zweck gibt es mehrere Konstanten, unter anderem die Konstante `ADD_AS_LAST` mit dem Wert `2`. Im folgenden Beispiel werden zwei Instanzen der Klasse `CommandExecutor` erzeugt, sowohl unter Verwendung der Konstantendefinition als auch durch Angabe des korrespondierenden Zahlenwerts:

```
final CommandExecutor executor1 = new CommandExecutor(ADD_AS_LAST);

final CommandExecutor executor2 = new CommandExecutor(2);  // Magic Number 2
```

Warum ist das ein Bad Smell? Bereits anhand dieses einfachen Beispiels sieht man, dass die Lesbarkeit unter dem Einsatz von Magic Numbers leidet. Das liegt daran, dass ein Zahlenwert wenig semantische Aussagekraft besitzt. Erschwerend kommt hinzu, dass Änderungen an den Werten – etwa wenn obige Konstante z. B. später den Wert 7 erhält – überall im nutzenden Sourcecode nachgezogen werden müssen. Das kann sich aufwendig und fehleranfällig gestalten, alle betroffenen Vorkommen einer Magic Number lediglich anhand ihres Werts aufzuspüren: Logischerweise repräsentiert nicht jedes Vorkommen der Zahl `2` die Konstante `ADD_AS_LAST`. Wird bei einer solchen Korrektur eine Stelle übersehen, so kann dies zu unerwarteten Resultaten und schwer zu findenden Fehlern führen: Wird irgendwann einmal der Wert der Konstante auf `4711` verändert und eine andere Konstante erhält den Wert `2`, so besagt diese `2` nun beispielsweise das Einfügen an erster Stelle und verändert den Programmablauf damit komplett. Wird dagegen eine andere Programmstelle versehentlich geändert, so kommt es dort zu Fehlern im Programmverhalten, weil der neue Wert (die `4711`) dort keinen Sinn ergibt und möglicherweise sogar außerhalb des zulässigen Wertebereichs liegt.

Tipps und Refactorings Mit sprechenden Konstantennamen kann man leicht ausdrücken, was bewirkt werden soll, hier das Einfügen an letzter Position. Zudem stellen nachträgliche Änderungen der Werte kein Problem im nutzenden Sourcecode dar. Ein weiterer Vorteil der Verwendung von Konstanten ist, dass diese im Nachhinein, wie

in diesem Beispiel sicher sinnvoll, leicht und ohne größere Änderungen in eine `enum`-Aufzählung umgewandelt werden können. Wenn jedoch Magic Numbers verwendet wurden, dann sollte man jedes Vorkommen überprüfen und gegebenenfalls anpassen.

16.1.2 Bad Smell: Konstanten in Interfaces definieren

Java erlaubt es, Konstanten in einem Interface zu definieren. Dabei lassen sich zwei Varianten unterscheiden. In der ersten wird ein Interface ausschließlich zur Definition von Konstanten verwendet, hier durch das Interface `FigureConstants` gezeigt:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureConstants
{
    // Randbreite und -höhe
    int BORDER_WIDTH  = 5;
    int BORDER_HEIGHT = 5;

    // Abstand der Punkte im Raster in X- und Y-Richtung
    int GRID_SIZE_X   = 20;
    int GRID_SIZE_Y   = 20;
}
```

In der zweiten Variante – hier am Beispiel des Interface `FigureIF` – werden sowohl Konstanten als auch Methoden definiert:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureIF
{
    // Rückgabewerte für hitTest
    int BORDER_HIT   = 0;
    int TITLE_HIT    = 1;
    int SIZE_BOX_HIT = 2;

    int hitTest(final int x, final int y);
    void draw(final Graphics g);
}
```

Warum ist das ein Bad Smell? Die Definition von Konstanten in Interfaces ist in vielerlei Hinsicht ungünstig. Zunächst ist durch die JLS jede dort definierte Konstante implizit `public static final` und jede Methode `public abstract`. Daher kann die Sichtbarkeit nicht eingeschränkt werden. Die Situation hat sich mit Java 8 und 9 leider nicht zum Guten geändert: Seit Java 8 sind neben Defaultmethoden auch statische Methoden in Interfaces erlaubt (vgl. Abschnitt 4.2). Mit Java 9 lassen sich sogar private Methoden in Interfaces definieren, was allerdings kein schönes Design ist.

Zudem ist es möglich, ein Konstanten-Interface durch eine Klasse zu implementieren. Dadurch werden die Konstantennamen in den Namensraum der jeweiligen Klasse aufgenommen, und somit geht der Verweis auf den tatsächlichen Definitionsort der Konstante verloren. Für Konstanten als Bestandteile eines Methoden deklarierenden Interface besteht das Problem gleichermaßen.

In der folgenden Klasse `BadFigure`, die zu Demonstrationszwecken das Interface `FigureConstants` implementiert, ist dies in der Methode `isInside(int, int)` mit der Verwendung der Konstanten `BORDER_WIDTH` und `BORDER_HEIGHT` gezeigt, wobei zum einen eine Referenzierung über `FigureConstants` und zum anderen über `BadFigure` erfolgt:

```
// ACHTUNG: Interfaces (möglichst) so nicht verwenden
public class BadFigure implements FigureConstants
{
    final Rectangle boundingRect;

    BadFigure(final Rectangle boundingRect)
    {
        this.boundingRect = boundingRect;
    }

    public boolean isInside(final int x, final int y)
    {
        final Rectangle innerRect = new Rectangle(boundingRect);

        // Definitionsort der Konstanten BORDER_WIDTH und BORDER_HEIGHT
        // ist bei Referenzierung über BadFigure unklar
        innerRect.grow(-FigureConstants.BORDER_WIDTH,
                       -BadFigure.BORDER_HEIGHT);

        return innerRect.contains(x, y);
    }
}
```

Die beschriebene und im Listing gezeigte »Namensraumverschmutzung« scheint zunächst ein etwas hergeholtes Argument zu sein. Doch in der Praxis erschwert eine solche Vorgehensweise mögliche Erweiterungen, Refactorings und die Nachvollziehbarkeit. Betrachten wir dazu die Problematik einmal genauer: Nehmen wir an, eine Klasse `BaseFigure` implementiert versehentlich ein Konstanten-Interface `Constants` mit der Konstante `OK`. Eine Erweiterung der Klassenhierarchie um eine Subklasse `ProcessingFigure` führt zu Kompilierfehlern, wenn diese Subklasse ein Interface `ProcessingResults` implementiert, in dem ebenfalls eine Konstante `OK` definiert ist: Die Konstante `OK` ist dann mehrdeutig. Das zeigt Abbildung 16-1.

Abbildung 16-1 *Doppeldeutigkeit der Konstante* `OK`

Tipps und Refactorings Benutzen Sie Interfaces nur dazu, wozu sie aus OO-Sicht dienen sollen, nämlich, um eine Schnittstelle mit angebotenen Methoden zu definieren.

Reine Konstanten-Interfaces können in der Regel in finale Konstantensammlungsklassen mit privaten Konstruktoren oder `enum`-Aufzählungen umgewandelt werden. Wie man dabei vorgeht, beschreibt das Refactoring WANDLE KONSTANTENSAMMLUNG IN `ENUM` UM in Abschnitt 17.4.12 als Schritt-für-Schritt-Anleitung.

Wurden solche Interfaces allerdings bereits (versehentlich) implementiert, dann muss mühselig jedes Vorkommen einer Konstante mit einer Referenz auf die neue Konstantenklasse versehen werden. Dabei kann es zu Problemen kommen, wenn man nicht alle Klassen im Zugriff hat oder verändern kann, die das Interface implementiert haben. Daher ist es sinnvoll, alle bekannten Nutzer vor einer solchen Maßnahme über den Umbau zu informieren. Ist die Nutzerbasis groß (z. B. für das Java-API), so ist ein solches Vorgehen nicht ohne Schwierigkeiten möglich und eine solche Designsünde lässt sich kaum mehr korrigieren, sondern oftmals nur als veraltet markieren. Dazu nutzt man das Javadoc-Tag `@deprecated` oder die Annotation `@Deprecated`. ***An diesem Beispiel erkennt man, wie wichtig es ist, gleich von Anfang an mit großer Sorgfalt zu arbeiten***. Dies gilt insbesondere bei der Implementierung aller nach außen sichtbaren Klassen und Methoden.

Info: Ähnliche Probleme durch statischen Import (`import static`)

Statische Imports ermöglichen es, Attribute oder Methoden anderer Klassen ohne Angabe des qualifizierenden Klassennamens zu nutzen. Es werden dadurch weitere Namen in den Namensraum der eigenen Klasse aufgenommen.

Beim Import statischer Attribute kommt es bei gleichnamigen, eigenen Klassenattributen zu Namenskonflikten und Kompilierproblemen, wie dies z. B. beim Implementieren verschiedener Interfaces mit gleichnamigen Konstanten der Fall ist.

Bei Mehrdeutigkeiten in Bezug auf Methoden treten keine Fehler beim Kompilieren auf, stattdessen überdeckt eine Objektmethode aus der eigenen Klasse eine statisch importierte Methode. Dadurch kommt es gegebenenfalls allerdings zu folgendem Problem: ***Wird bei einer Erweiterung eine Objektmethode eingeführt, so wird das Verhalten unerwartet verändert, da als Folge nicht mehr die statisch importierte Methode aufgerufen wird, sondern die neu eingeführte Methode der eigenen Klasse.*** Dies gilt übrigens auch dann, wenn die aufrufende Methode selbst statisch ist!

16.1.3 Bad Smell: Zusammengehörende Konstanten nicht als Typ definiert

Müssen einige Werte als Konstanten definiert werden, wie etwa die Einfügestrategien für Kommandos aus dem Beispiel der Klasse `CommandExecutor`, sieht man häufiger Definitionen zusammengehörender Konstanten als `int`-Werte:

```
/** altes Kommando durch neues ersetzen */
public static final int REPLACE_OLD_OR_ADD_AS_LAST = 0;

/** Kommando als erstes Kommando neu erzeugen */
public static final int ADD_AS_FIRST               = 1;

/** Kommando als letztes Kommando neu erzeugen */
public static final int ADD_AS_LAST                = 2;
```

Warum ist das ein Bad Smell? Werden Konstanten als `int` definiert, so sind als Werte grundsätzlich alle aus dem Wertebereich des Typs erlaubt. Dadurch können auch unsinnige Werte Anwendung finden, wie folgendes Beispiel zeigt:

```
// Was mag die 4711 bedeuten?
final CommandExecutor executor = new CommandExecutor(4711);
```

Wünschenswert ist es, sicherzustellen, dass die übergebenen Werte im Bereich der definierten Konstanten liegen. Das erfordert eine Bereichsprüfung und wird schnell unübersichtlich. Dieser Effekt verstärkt sich, wenn die Prüfungen an verschiedenen Stellen im Sourcecode notwendig sind, die Werte keinen zusammenhängenden Bereich abbilden oder sich Werte nachträglich ändern bzw. neue Konstanten hinzukommen. Machen wir es konkret für den folgenden Konstruktor:

```
public CommandExecutor(final int strategy)
{
    if (strategy < REPLACE_OLD_OR_ADD_AS_LAST || strategy > ADD_AS_LAST)
    {
        throw new IllegalArgumentException("parameter 'strategy' is invalid: " +
                        "value='" + strategy + "' not in range [" +
                        REPLACE_OLD_OR_ADD_AS_LAST + " -- " + ADD_AS_LAST + "]");
    }
    this.registrationStrategy = strategy;
}
```

Prinzipiell gut an der Parameterprüfung ist, dass sie überhaupt vorhanden ist und der Benutzer bei einem Fehler im Aufruf detailliert sowohl über die Wertebelegung als auch über gültige Parameterwerte informiert wird.

Allerdings stehen diesen Vorteilen einige Nachteile gegenüber. Die obige Wertebereichsprüfung und Fehlerbehandlung ist aus folgenden Gründen ziemlich fragil:

1. Es werden Annahmen über die konkreten Werte der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` und `ADD_AS_LAST` gemacht, insbesondere, dass der Wert von `ADD_AS_LAST` den größeren der beiden Werte repräsentiert. Wird der Wert der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` bzw. `ADD_AS_LAST` derart verändert, dass diese Annahme nicht mehr gilt, scheitert die Wertebereichsprüfung.
2. Der Vergleich setzt voraus, dass durch die Konstanten ein zusammenhängender Wertebereich abgedeckt wird. Diese Forderung erschwert unter Umständen eine sinnvolle Vergabe der Parameterwerte.

3. Wird im Nachhinein eine zusätzliche Kommandoart hinzugefügt, erfordert dies nicht nur die Definition einer neuen Konstanten, sondern es müssen überall im verwendenden Sourcecode Anpassungen in den Wertebereichsprüfungen erfolgen.

Tipps und Refactorings In der Regel möchte man Aufrufern lediglich erlauben, symbolische Werte, d. h. Konstantennamen, zu verwenden. Das lässt sich elegant und einfach durch das Sprachmittel `enum` erreichen, mit dem man Konstanten zu einem neuen Typ zusammenfassen kann:

```
enum RegistrationStrategyType
{
    REMOVE_OLD_AND_ADD_AS_LAST(0, "altes Kommando durch neues ersetzen"),
    ADD_AS_FIRST(1, "Kommando als erstes Kommando neu erzeugen"),
    ADD_AS_LAST(2, "Kommando als letztes Kommando neu erzeugen");

    final int    value;
    final String description;

    RegistrationStrategyType(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }
}
```

Die durch Einsatz eines `enum` gewonnene Typsicherheit löst sowohl die Probleme der Bereichsprüfung – diese ist nun überflüssig, da sie implizit durch den Compiler basierend auf der Typangabe sichergestellt wird – als auch die der nicht zusammenhängenden Wertebereiche oder sich ändernder Konstantenwerte. Eine Anleitung liefert das Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM in Abschnitt 17.4.12.

16.1.4 Bad Smell: Casts auf unbekannte Subtypen

Casts werden in Programmen verwendet, wenn eine Typumwandlung gewünscht wird, die vom Compiler nicht garantiert werden kann. In wenigen Fällen ist ein Einsatz unumgänglich, beispielsweise wenn man ein Zahlenliteral vom Typ `int` an eine Methode mit `short`-Parametern übergeben möchte. In der Regel weist der Einsatz eines Casts allerdings auf eine potenzielle Fehlerquelle hin, und man sollte einen genaueren Blick auf die entsprechende Sourcecode-Stelle werfen.

Betrachten wir die Problematik am Beispiel einer Methode `getPersons()`, die in einem Interface `IDataAccess` definiert ist:

```
interface IDataAccess
{
    List<Person> getPersons();
}
```

Der folgende Sourcecode-Ausschnitt nutzt eine Realisierung dieses Interface zum Aufruf von `getPersons()`. Vor der Zuweisung an die Variable `persons` findet ein Cast auf eine `LinkedList<Person>` statt:

```
final LinkedList<Person> persons = (LinkedList<Person>) dbAccessor.getPersons();
final Iterator<Person> it = persons.iterator();
while (it.hasNext())
{
    processValue(it.next());
}
```

Warum ist das ein Bad Smell? In diesem Beispiel erfolgt eine Umwandlung der zurückgelieferten Referenz auf den Typ `LinkedList<Person>`, obwohl im Interface `IDataAccess` lediglich die Rückgabe einer `List<Person>` garantiert wird. Sofern die momentane Implementierung des Interface `IDataAccess` für `getPersons()` eine `LinkedList<Person>` verwendet und zurückliefert, kommt es zu keinem Laufzeitfehler. Der aufrufende Sourcecode verlässt sich allerdings auf ein Implementierungsdetail. Würde der obige Programmcode, beispielsweise aufgrund von Performance-Messungen, auf den Einsatz einer `ArrayList<Person>` umgestellt, so wäre dies immer noch Interface-konform. Es käme jedoch im Beispiel in der aufrufenden Methode zu einer `ClassCastException`.

Casts sind an sich schon kein schöner Programmierstil, da sie Typumwandlungen vornehmen, deren Fehlerfreiheit vom Compiler nicht garantiert werden kann. Problematisch werden sie jedoch, wenn ein sogenannter ***Down Cast*** von einem Basistyp auf einen Subtyp erfolgt. Dies entspricht in der Ableitungshierarchie dem »Weg nach unten«, der allerdings nicht zugesichert ist und häufig zu `ClassCastException`s führt, beispielsweise wenn man folgende Methode `downcastToSub(Object)` mit einem Objekt vom Typ `Person` statt `Sub` aufruft:

```
public Sub downcastToSub(final Object obj)
{
    final Sub sub = (Sub) obj;        // Down Cast: Object -> Sub
    // ...
```

Erinnern wir uns an die OO-Grundlagen: Es gilt die Beziehung Subtyp »is-a« Basistyp, aber nicht umgekehrt. Nur für den Fall, dass an die obige Methode ein Objekt mindestens vom Typ `Sub` übergeben wird, ist der Cast erfolgreich. Ansonsten kommt es zu einer Inkompatibilität von Typen, wodurch eine `ClassCastException` ausgelöst wird. Einen solchen Cast sollte man daher nur nach expliziter Typprüfung durchführen:

```
public Sub downcastToSub(final Object obj)
{
    if (obj instanceof Sub)
    {
        // Cast ist nun sicher
        final Sub sub = (Sub) obj;     //  Down Cast: Object -> Sub
        // ...
    }
    else
    {
        // sollte nicht auftreten, z. B. IllegalStateException auslösen
    }
    // ...
```

Der Cast ist nun zwar sicher, aber es stellt sich die Frage, wie auf nicht erwartete Typen reagiert werden soll. Eine ähnliche Situation wird als BAD SMELL: UNVOLLSTÄNDIGE BETRACHTUNG ALLER ALTERNATIVEN in Abschnitt 16.1.8 behandelt.

Tipps und Refactorings Anhand des eingangs gezeigten Beispiels sieht man einerseits, dass der Cast in diesem Fall vollkommen überflüssig ist, und andererseits, dass es durch Programmieren gegen das Interface `List<T>` gar nicht erst zu dem Problem gekommen wäre. Als Merksatz gilt: ***Vermeide Casts auf konkrete Klassen und programmiere stattdessen gegen Interfaces***. Sofern kein gemeinsames Interface vorhanden ist, kann man – falls sinnvoll – ein solches einführen. Casts kann man vermeiden, wenn man Referenzen auf das gemeinsame Interface zum polymorphen Aufruf von Methoden nutzt. Dieses Vorgehen entspricht eher der objektorientierten Denkweise.

16.1.5 Bad Smell: Programmcode im Logging-Code

Programmcode und Logging-Code sollte man immer funktional trennen. Dieser Bad Smell betrachtet, was passieren kann, wenn man dies nicht tut.

In folgendem Beispiel wird im Log-Level `debug` die Methode `resetLineCounter()` aufgerufen, für andere Log-Level jedoch nicht:

```
if (log.isDebugEnabled())
{
    log.debug("...");
    resetLineCounter();
}
```

Noch schlimmer ist es, wenn der Aufruf noch ein wenig versteckter wie folgt geschieht:

```
log.info("Route info: " + calculateRouteInfo());
```

Warum ist das ein Bad Smell? Erfolgt das Logging, z. B. während der Entwicklung, zunächst immer im Debug-Level, scheint die Software fehlerfrei zu funktionieren. Wird im späteren Verlauf eines Projekts der Log-Level allerdings erhöht, so erzeugt man dadurch einen Softwaredefekt. Das Rücksetzen der Variablen durch Aufruf der Methode `resetLineCounter()` wird nun nicht mehr durchgeführt. Noch fataler sind die Folgen dieses Bad Smells, wenn ein Kunde die Funktion bereits abgenommen hat und selbstständig später die Log-Konfiguration ändert. Im zweiten Beispiel ist das Ganze noch fataler. Eigentlich sollte es problemlos und ohne Auswirkungen auf Business-Funktionalität möglich sein, eine Zeile mit Log-Aufruf auszukommentieren. Im obigen Beispiel würden gegebenenfalls wichtige Zustandsänderungen ausbleiben und die Software anders als erwartet funktionieren.

Tipps und Refactorings Ist die Ausführung von Programmcode abhängig vom Log-Level, so kann dies zu extrem schwer zu findenden Fehlern führen! Daher ist Programmcode innerhalb von Log-Ausgaben zu vermeiden. Der Aufruf von `get()`-Methoden ist meistens akzeptabel. Aber selbst hierbei muss man achtsam sein, denn man sieht leider immer wieder `get()`-Methoden, die Seiteneffekte verursachen.

Warnung: Ähnliche Probleme mit Assertions

Nach der Lektüre von Abschnitt 5.6.5 wissen wir, dass Assertions standardmäßig deaktiviert sind. Außerdem ist es möglich, diese Package-weise zu (de)aktivieren. Daher darf man sich niemals darauf verlassen, dass Anweisungen innerhalb eines `assert`s abgearbeitet werden. Folgende Methode ist daher fragil und löscht je nach Einstellung Daten oder eben auch nicht:

```
protected boolean deleteValues(final String key)
{
    if (exists(key))
    {
        assert delete(key);
        return true;
    }
    return false;
}
```

16.1.6 Bad Smell: Dominanter Logging-Code

Wie zuvor erkannt, sollte man Programmcode und Logging-Code funktional voneinander trennen. Teilweise ist die eigentliche Programmfunktionalität kaum mehr zu erkennen, da der Sourcecode extrem mit Logging-Code durchsetzt ist.

Folgendes Beispiel stammt aus einem Fahrgastinformationssystem zur Bereitstellung von Busabfahrten. Das Listing zeigt die Erzeugung von Informationen zu neuen Abfahrten mithilfe der Methode `DepartureFactory.createNewDepartures()` und deren Verarbeitung durch `processNewDepartures(List<Departure>)`:

```
log.info("Start creating new departures ...");
final List<Departure> newDepartures = DepartureFactory.createNewDepartures();
log.info("Finished creation of departures");

if (log.isDebugEnabled())
{
    log.debug("newly created departures:");
    final Iterator<Departure> it = newDepartures.iterator();
    while (it.hasNext())
    {
        final Departure departure = it.next();
        log.debug("new Departure: " + "time = '" + departure.getTime() + "'"
                    + "line = '" + departure.getLine() + "'"
                    + "destination = '" + departure.getDestination());
    }
}
log.info("process new departures");
processNewDepartures(newDepartures);
```

Warum ist das ein Bad Smell? Obwohl beide Aufgaben in eigene Methoden ausgelagert sind, erschließt sich in diesem Ausschnitt deren Zusammenhang und auch deren Aufruf nicht sofort. Das liegt insbesondere daran, dass der Logging-Code den Applikationscode dominiert und stark durchsetzt. Tatsächlich finden sich zwei Zeilen Nutzcode und über 10 Zeilen Logging-Code, in dem zum einen einfache Abläufe protokolliert werden und zum anderen eine längere (und bereits leicht unübersichtliche) Aufbereitung von Log-Ausgaben erfolgt.

Verschärft wird das Ganze dadurch, dass Sprünge zwischen verschiedenen Abstraktionsebenen vorliegen, wodurch wir beim Betrachten des Sourcecodes immer wieder gedanklich zwischen verschiedenen Abstraktionsgraden hin und her wechseln müssen. In Abschnitt 19.2 zeige ich, wie sich das Layout des Sourcecodes auf dessen Verständlichkeit auswirkt.

Tipps und Refactorings Wenn man einen Teil des Logging-Codes in eine Hilfsmethode `logAllDepartureInfos(List<Departure>)` auslagert, so wird der Sourcecode bereits um einiges klarer. Dadurch kann man vermutlich auch auf die protokollierenden Informationsmeldungen verzichten – oder diese in die beiden Nutzcodemethoden verlagern. Als Letztes kann man Logging und Informationsverarbeitung vertauschen. Damit wird das Verarbeiten und das Ermitteln der Informationen noch stärker verknüpft und der Sourcecode kommuniziert ohne den feingranularen Logging-Code sehr gut, was er tut:

```
final List<Departure> newDepartures = DepartureFactory.createNewDepartures();
processNewDepartures(newDepartures);

logAllDepartureInfos(newDepartures);
```

16.1.7 Bad Smell: Unvollständige Änderungen nach Copy-Paste

Bei Erweiterungen wird teilweise Funktionalität ähnlich zu bereits implementierten Programmteilen benötigt. Aus Bequemlichkeit wird dann mitunter Copy-Paste genutzt, um die erforderlichen Teile zu übernehmen und an einigen Stellen zu modifizieren.

Betrachten wir dies anhand der Methode `createWatchdog(byte[])`:

```
public Watchdog createWatchdog(final byte[] data) throws
      UnknownProtocolException
{
    final Encoding encoding = EncodingFactory.getEncodingFor(Watchdog.class);
    if (encoding == null)
        throw new UnknownProtocolException(Watchdog.class);

    return new Watchdog(data, encoding);
}
```

Diese Methode dient als Ausgangsbasis und Kopiervorlage, um eine ziemlich ähnliche Methode `createContent(byte[])` neu zu erzeugen. Die Kopie sieht nach ein paar Modifikationen wie folgt aus:

```
public Content createContent(final byte[] data) throws UnknownProtocolException
{
    final Encoding encoding = EncodingFactory.getEncodingFor(Content.class);
    if (encoding == null)
        throw new UnknownProtocolException(Watchdog.class);
    //                                          ********
    return new Content(data, encoding);
}
```

Zunächst wird versucht, ein passendes Encoding durch eine `EncodingFactory` bereitzustellen. Ist kein solches registriert, wird von der Methode `getEncodingFor(Class<T>)` der Rückgabewert `null` geliefert. Die Applikation löst daraufhin eine `UnknownProtocolException` aus. Existiert ein passendes Encoding, so wird das gewünschte Objekt aus den übergebenen Daten und dem Encoding erzeugt.

Es fällt auf, dass in den kopierten Zeilen der Fehlerbehandlung beim Auslösen der `UnknownProtocolException` die Anpassung der Typangabe von `Watchdog.class` auf `Content.class` vergessen wurde.

Warum ist das ein Bad Smell? Mitunter scheint Copy-Paste einen schnellen Weg zu bieten, um gewünschte Funktionalität übernehmen und modifizieren zu können. Als Folge steigt jedoch der Sourcecode-Umfang, und es entstehen diverse ähnliche Blöcke mit lediglich einigen (kleineren) Veränderungen. Solange die funktionale Korrektheit des Programms gegeben ist, wirkt die Duplikation eher wie ein kosmetisches Problem. Allerdings verschlechtert sich die Verständlichkeit: Dadurch ist solcher Sourcecode schwieriger erweiter- und wartbar. Zudem schleichen sich bei Änderungen sehr leicht Fehler ein. Durch die Duplikation müssen Änderungen an einer Stelle oft an allen kopierten Stellen nachgezogen werden. Im hektischen Alltag wird schnell eine davon übersehen, wodurch jedoch unvollständige Anpassungen entstehen, die Inkonsistenzen auslösen. Es beginnt ein Wartungsalbtraum.

In diesem Beispiel wird im Fehlerfall zwar lediglich eine irreführende Fehlermeldung generiert. Eine solche Fehlinformation erschwert jedoch eine spätere Fehlersuche. Mit etwas Fantasie kann man sich ausmalen, welche Folgen und Verarbeitungsfehler an kritischeren Stellen drohen, wenn nicht nur eine Warnmeldung aus einer Exception erzeugt wird.

Tipps und Refactorings Beim Verwenden von Copy-Paste sollte man immer sehr große Sorgfalt walten lassen und sich den kopierten Sourcecode und alle nachfolgenden Anpassungen genau anschauen. Häufig lässt sich Copy-Paste vermeiden, indem man Convenience-Methoden einführt, den betroffenen Sourcecode-Ausschnitt dorthin verlagert und die neue Methode an der Original- und Zielstelle aufruft, anstatt die betreffenden Stellen zu kopieren. In diesem Beispiel würde es sich anbieten, eine Methode

zur Prüfung auf ein gültiges Encoding in die Klasse `EncodingFactory` aufzunehmen und im Fehlerfall hier eine `UnknownProtocolException` auszulösen, statt `null` zurückzugeben. Derartige Problemfälle behandelt BAD SMELL: RÜCKGABE VON NULL STATT EXCEPTION IM FEHLERFALL in Abschnitt 16.3.6.

16.1.8 Bad Smell: Unvollständige Betrachtung aller Alternativen

Teilweise sieht man Sourcecode-Abschnitte, die verschiedene Alternativen über `if` oder `switch-case` unterscheiden, jedoch einige mögliche Fälle nicht betrachten.

Dieses Beispiel stammt aus einer Applikation, die für jede ihrer Teilkomponenten einen eigenen Systemzustand verwaltet. Dieser lässt sich abfragen und dient als Eingabe für die Berechnung eines Gesamtzustands. Die Einzelzustände werden in einer Methode `updateState(ComponentId)` durch den Aufruf einer Methode `calcState(ComponentId)` ermittelt und in der Variablen `stateMap` gespeichert. Wird von `calcState(ComponentId)` unerwartet der Wert `null` zurückgeliefert, so findet allerdings keine Speicherung statt:

```
private Map<ComponentId, ComponentState> stateMap = new HashMap<>();

public void updateState(final ComponentId componentId)
{
    final ComponentState state = calcState(componentId);

    if (state != null)
    {
        stateMap.put(componentId, state);
    }
}
```

Warum ist das ein Bad Smell? Betrachtet man obiges Listing, so ist zunächst kein Fehler zu erkennen. Bei einem fehlenden `else`-Zweig sollte man sich aber immer fragen, was in einem solchen Fall passiert bzw. ob ein solcher Fall eintreten kann.

Tatsächlich kam es in seltenen Fehlersituationen dazu, dass die Methode `calcState(ComponentId)` nicht einen der definierten Aufzählungswerte, sondern `null` lieferte. Dieses unerwartete Verhalten wurde aber weder protokolliert noch fand eine Aktualisierung des Zustands statt, somit wurde folglich der zuvor in der Map gespeicherte Systemzustand beibehalten. In ungünstigen Fällen wurde also irrtümlich weiterhin von einem funktionierenden System ausgegangen, obwohl ein Fehler vorlag. Das Nachvollziehen dieser seltenen Fehlersituation war zeitaufwendig, da das gesamte System keinen Hinweis auf eine Fehlfunktion lieferte. ***Der Einsatz von Aufzählungen schützt zwar vor ungültigen Werten, jedoch müssen dann von Klienten gegebenenfalls Referenzen auf `null` geprüft werden.*** Ähnliche Probleme behandeln BAD SMELL: UNBEDACHTE RÜCKGABE VON NULL in Abschnitt 16.3.5 sowie BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN in Abschnitt 16.3.8.

Tipps und Refactorings Selbst wenn man der Meinung ist, einen Fall gänzlich ausschließen zu können, sollte man einen `else`-Zweig einführen. Dort kann für diese unerwartete Situation eine Warnmeldung in eine Log-Datei erfolgen oder eine `IllegalStateException` ausgelöst werden. Durch Letzteres hat man die Gewissheit, mögliche Programmier- oder Denkfehler aufzudecken. Ein großer Vorteil dieses offensiven Umgangs mit Fehlersituationen ist, dass dadurch häufig Fehler noch relativ früh im Entwicklungsprozess erkannt werden können. Selbst wenn ein solcher Fehler erst während der Wartungsphase auftritt, kann eine Exception auf veränderte Umgebungsbedingungen hindeuten und eine Fehlerkorrektur erleichtern. Folgende Modifikation hilft, mögliche Fehler bei der Ermittlung des Systemstatus aufzudecken. Insbesondere wird nun die Fehlersituation sofort deutlich und der Normalfall erfordert keine Sonderbehandlung mehr. Folgender Sourcecode ist also verständlicher:

```
public void updateStateImproved(final ComponentId componentId)
{
    final ComponentState state = calcState(componentId);

    if (state == null)
        throw new IllegalStateException("variable 'state' must not be null!");

    stateMap.put(componentId, state);
}
```

16.1.9 Bad Smell: Prä-/Postinkrement in komplexeren Statements

Die Verwendung von Prä-/Postinkrements bzw. -dekrements erlaubt häufig eine kompaktere Schreibweise des Sourcecodes. Nutzen wir diese jedoch in Kombination mit Abfragen oder in Array-Zugriffen, so kann dies schnell unübersichtlich werden.

Betrachten wir diesen Bad Smell konkret am Beispiel einer Variablen `pos`, die innerhalb einer Schleife in einem Array-Zugriff inkrementiert wird. Das kann mit Präinkrement folgendermaßen realisiert werden:

```
while (pos < message.length)
{
    if (message[++pos] != EOL_BYTE_VALUE)
    {
```

Mit Postinkrement schreibt man Folgendes:

```
while (pos < message.length)
{
    if (message[pos++] != EOL_BYTE_VALUE)
    {
```

Warum ist das ein Bad Smell? Beides sieht auf den ersten Blick sehr ähnlich aus. Es gibt jedoch einen subtilen Unterschied: Im Fall des Postinkrements wird zunächst der Wert aus dem `message`-Array ausgelesen und *anschließend* der Wert von `pos` erhöht. Im Fall des Präinkrements wird der Wert von `pos` jedoch *vor* dem Auslesen erhöht. Dadurch greift man auf einen anderen Wert im Array zu und vergleicht andere Werte als beim Postinkrement. Der entscheidende Punkt ist, dass die unterschiedliche Bedeutung nicht sofort ins Auge springt. Das Ganze hat folgende Konsequenzen:

1. **Die Position des Array-Zugriffs ist nicht intuitiv ersichtlich:** Möglicherweise werden falsche (unbeabsichtigte) Werte miteinander verglichen, sodass der Vergleich ein unerwartetes Ergebnis liefert. Zusätzlich wird häufig, wie in diesem Fall, der Index durch das Präinkrement am Ende der Schleifendurchläufe ungültig: Für den Fall, dass `pos` den Wert `message.length-1` erreicht, kommt es beim Aufruf des Zugriffs `message[++pos]` zu einer `ArrayIndexOutOfBoundsException`. Selbst wenn eine solche Exception nicht auftritt, gilt das nächste Argument.
2. **Es wird Vergleichslogik mit Schleifenlogik vermischt:** Wird eine Schleifenvariable nicht offensichtlich (z. B. beim Array-Zugriff) innerhalb der Schleife geändert, wird es schwieriger, deren Veränderungen nachzuvollziehen. Dies gilt vor allem dann, wenn an verschiedenen Stellen in der Schleife erneut auf die Schleifenvariable zugegriffen wird. Noch komplizierter wird es, wenn mehrere solcher Anweisungen hintereinander, geschachtelt oder sogar bedingt ausgeführt werden, wie dies im folgenden Beispiel angedeutet ist. Generell vergrößert sich die Gefahr nicht abgesicherter Array-Zugriffe und daraus resultierender `ArrayIndexOutOf-BoundsException`s.

```
while (pos < message.length)
{
    // erhöht in jedem Fall
    if (message[pos++] == START_TAG_VALUE)
    {
        // erhöht nur, wenn START_TAG_VALUE gefunden wurde
        if (message[pos++] != EOL_BYTE_VALUE)
        {
```

Tipps und Refactorings Man sollte Vergleichs- und Zuweisungslogik möglichst immer von Variablenänderungen trennen. Das Durchlaufen einer Schleife und das Verändern der Schleifenvariablen sollte in der Schleifenbedingung oder aber als separate Zeile am Ende der Schleife erfolgen.

Hinweis: Extrembeispiel und tückischer Denkfehler

Extrembeispiel Betrachten wir folgendes Extrembeispiel, um die Idee dieses Bad Smells zu verdeutlichen. Es basiert auf dem Puzzle »Tricky Assignment« aus dem Buch »Java Puzzlers« von Joshua Bloch und Neil Gafter [8].

```
int tricky = 0;
for (int i = 0; i < 10; i++)
    tricky += tricky++;
```

Raten Sie mal, welchen Wert die Variable `tricky` als Ergebnis dieser Schleife hat: 0, 10, 45 oder einen anderen? Die richtige Antwort ist: 0. Merkwürdig! Wie kommt das? Schreiben wir die Zeile einmal so, dass wir einige Zwischenschritte erkennen:

```
// tricky += tricky++;
=> tricky = tricky + tricky++;
```

Auch diese Anweisung können wir noch weiter aufschlüsseln und erkennen dann die Ursache: Die Variable `tricky` wird zwar inkrementiert, jedoch wird zuvor der alte Wert von `tricky` in einer Hilfsvariablen `temp` zwischengespeichert und anschließend wieder an `tricky` zugewiesen. Die Erhöhung geht dadurch verloren:

```
// tricky = tricky + tricky++;
1) temp = tricky + tricky;
2) tricky++;
3) tricky = temp;
```

Tückischer Denkfehler beim Methodenaufruf oder bei Rekursion Bezüglich Postinkrement und -dekrement lauern weitere Fallstricke – sehr schnell schreibt man folgende Methode und ist sich vermutlich ziemlich sicher, dass diese abbricht, wenn sie mit einem Wert kleiner 10 aufgerufen wird:

```
void doSomething(int value)
{
    if (value > 10)
        System.out.println("More than ten");
    else
        doSomething(value++);
}
```

Tatsächlich terminiert diese Methode nicht, wenn nicht initial ein Wert größer 10 übergeben wird. Für einen Aufruf von `doSomething(0)` wird entgegen der Vermutung, dass danach `doSomething(1)`, `doSomething(2)` usw. aufgerufen würde, der Wert nie erhöht und es kommt zu Selbstaufrufen mit `doSomething(0)`, bis diese in einem `StackOverflowError` resultieren.

16.1.10 Bad Smell: Mehrere aufeinanderfolgende Parameter gleichen Typs

Werden Methoden überladen, so unterscheiden sich diese lediglich in ihrer Parameterliste. Unübersichtlich wird dies, wenn in den Signaturen mehrere Übergabeparameter des gleichen Typs aufeinanderfolgen.

Betrachten wir folgende Signaturen einer Methode `clearRect()`:

```
public void clearRect(int x, int y, int width, int height)
public void clearRect(int x1, int y1, int x2, int y2, long colorRGB)
public void clearRect(int red, int green, int blue,
                      int x, int y, int width, int height)
```

Warum ist das ein Bad Smell? Bei mehreren Parametern gleichen Typs hintereinander in der Parameterliste besteht die Gefahr der Verwechselung von Positionen und Bedeutungen. Bereits die Anzahl der Parameter macht das Ganze unübersichtlich und erhöht die Wahrscheinlichkeit für Fehler. Je mehr gleichartige Parameter verwendet werden, desto stärker macht sich das Problem bemerkbar.

Im obigen Beispiel variiert bei der zweiten Variante der überladenen Methode die Bedeutung der Parameter drei und vier gegenüber der ersten Methode. Dies ist problematisch, wenn man beispielsweise beim Löschen eine Farbe angeben möchte und intuitiv den ursprünglichen Aufruf lediglich um einen Parameter zur Farbinformation ergänzt. Man erhält so ein ganz anderes Applikationsverhalten als beim ursprünglichen Aufruf. Dies ist dadurch bedingt, dass die zweite Methode als Parameter drei und vier ein zweites Koordinatenpaar statt einer Angabe von Breite und Höhe erwartet.

Für Referenzparameter führt das Verwechseln von Positionen nicht nur zu geänderten Eingabewerten, sondern eventuell auch zu Exceptions: Wird an unerwarteter Stelle ein `null`-Wert übergeben, kommt es als Folge potenziell zu einer `NullPointerException`.

In diesem Beispiel ist eine weitere Inkonsequenz vorhanden: Die Methode mit fünf Parametern erwartet die Farbinformation in einem anderen Format als die Methode mit sieben Parametern, die einzelne RGB-Werte verwendet. Außerdem stimmen die Positionen nicht überein. Das Ganze verstößt zudem gegen das »***Prinzip des geringsten Erstaunens***« (»***Principle of Least Astonishment***« (POLA)).

Tipps und Refactorings Um die zuvor diskutierten Probleme zu lösen, ist es zum einen sinnvoll, ***die Reihenfolge von Parametern immer so einheitlich wie möglich zu halten und einer programmweiten Systematik folgen zu lassen***. Zum anderen erhält man durch das Zusammenfassen von Parametern zu Gruppen – sogenannten PARAMETER VALUE OBJECTS (vgl. Abschnitt 3.4.5) – verständlichere Signaturen, die eine Verwechselung von Positionen unwahrscheinlich machen. Folgendes Listing zeigt eine mögliche Realisierung mit den als PARAMETER VALUE OBJECT realisierten Klassen `Point`, `Dimension` und `Color`:

```
public void clearRect(Point point, Dimension size)
public void clearRect(Point point1, Point point2, long colorRGB)
public void clearRect(Color clearColor, Point point, Dimension size)
```

16.1.11 Bad Smell: Grundloser Einsatz von Reflection

Mithilfe von Reflection lassen sich Metainformationen über Klassen ermitteln. Dadurch kann man zur Laufzeit sogar auf solche Klassen zugreifen, die bei der Kompilierung der eigenen Klassen noch nicht zur Verfügung standen und deren Name nur textuell vorliegt, z. B. durch Angabe in einer Konfigurationsdatei. Statt einer Referenzierung über `import` erfolgt der Zugriff über eine textuelle Repräsentation des voll qualifizierten Klassennamens.

Betrachten wir ein Beispiel. Per Reflection wird mit `newInstance()` ein neues Objekt vom Typ `AnyClass` erzeugt und dieses einer Referenzvariablen vom selben Typ zugewiesen. Diese Zuweisung an einen konkreten Typ erfordert aber, dass diese Klasse bereits zur Kompilierzeit bekannt ist. Reflection wird anschließend nicht mehr verwendet, sondern es erfolgt ein »normaler« Aufruf der Methode `output(String)`:

```
try
{
    final Class<?> c = Class.forName("util.AnyClass");
    final AnyClass newObject = (AnyClass) c.newInstance();
    newObject.output("1,2,3");       // Direkter, normaler Methodenaufruf
}
catch (final ClassNotFoundException e)
{
    handleReflectionException(e);
}
catch (final InstantiationException e)
{
    handleReflectionException(e);
}
```

Warum ist das ein Bad Smell? Die Verwendung von Reflection macht ein Programm in der Regel schlechter lesbar und dadurch schwieriger wartbar. Außerdem sind diverse Exceptions abzufangen. Insgesamt handelt es sich bei Reflection um ein fragiles Gebilde, wenn Änderungen an Klassen, der Package-Struktur oder an Methodensignaturen zu erwarten sind. Der Einsatz von Reflection erschwert damit alle Basis-Refactorings (u. a. MOVE TO PACKAGE, RENAME), die detailliert im Buch »Refactoring« von Martin Fowler [20] beschrieben sind.

Der Einsatz von Reflection sollte daher immer gut überlegt sein. Für die obige Konstruktion fällt es allerdings schwer, eine Begründung zu finden, da es sich bei der angesprochenen Klasse offenbar um eine dem Compiler bekannte Klasse handelt, die problemlos wie folgt hätte erzeugt und genutzt werden können:

```
final AnyClass newObject = new AnyClass();
newObject.output("1,2,3");
```

Tipps und Refactorings Reflection birgt einige potenzielle Fehlerquellen und sollte daher nur dann eingesetzt werden, wenn kein konventioneller Zugriff möglich ist, etwa bei optionalen Komponenten. Lässt sich jedoch schon zur Kompilierzeit darauf zugreifen, sollte man auf Reflection verzichten. Praktischerweise gibt es seit JDK 7 zwei Möglichkeiten, das Exception Handling zu vereinfachen: Mit der Klasse `ReflectiveOperationException` können verschiedene, durch Reflection verursachte Exceptions in Form einer Exception behandelt werden. Auch durch das Sprachfeature Multi Catch lässt sich eine Menge von Exceptions gleichartig abfangen.

16.1.12 Bad Smell: `System.exit()` mitten im Programm

Ein Programmabbruch durch einen Aufruf von `System.exit(int)` mitten im Programmablauf birgt einige Gefahren und sollte daher immer genau hinterfragt werden.

Exemplarisch betrachten wir dies an der Methode `startConnection(int)`, die einen Verbindungsaufbau abhängig von einem durch den `int`-Parameter `transmissionMode` gewählten Übertragungsmodus startet. Wird ein unbekannter Modus angegeben, so landet man im `default`-Zweig der `case`-Anweisung, wo das Programmende mit `System.exit(int)` erzwungen wird:

```
private void startConnection(final int transmissionmode)
{
    switch (transmissionmode)
    {
        case TRANSMISSION_MODE_RADIO:
            startRadioConnection();
            break;
        case TRANSMISSION_MODE_NETWORK:
            startNetworkConnection();
            break;
        default:
            // FALSCHER ÜBERTRAGUNGSMODUS
            log.fatal("Unknown transmission mode: " + transmissionmode);
            System.exit(-1);
    }
}
```

Vor dem Programmende wird eine Information über den falsch gewählten Übertragungsmodus in die Log-Ausgabe geschrieben. Dadurch hat man zumindest im Nachhinein die Chance, das Problem einzugrenzen.

Warum ist das ein Bad Smell? Der Ausstieg mit `System.exit(int)` ist problematisch, da ein Programm dadurch sofort terminiert wird, ohne Aufräumarbeiten durchführen zu können, etwa offene Dateien, Sockets, Datenbankverbindungen oder sonstige Ressourcen zu schließen bzw. freizugeben.[2] Im Extremfall stehen dann bei einem möglichen Neustart eines anderen Programms bzw. einer anderen JVM die zuvor belegten Ressourcen nicht oder noch nicht wieder zur Verfügung und verhindern so

[2] Tatsächlich kann man mit etwas Mühe einen sogenannten Shut-down-Hook realisieren, der Aufräumarbeiten vornimmt, wie dies im nachfolgenden Praxistipp angedeutet wird.

die Programmausführung. Das gilt insbesondere für Programme, die eigentlich immer laufen sollten (24/7) und die bei Programmabstürzen oder beim Programmende automatisch neu gestartet werden. Wenn dann ein Netzwerkfehler für andauernde Neustarts sorgt, sind die zuvor beschriebenen Probleme leicht nachvollziehbar.

Tipps und Refactorings Zunächst ist es fehlerträchtig, dass Aufrufer durch die unbedachte Wahl des Parametertyps `int` überhaupt in der Lage sind, einen falschen Wert an diese Methode zu übergeben. BAD SMELL: ZUSAMMENGEHÖRENDE KONSTANTEN NICHT ALS TYP DEFINIERT diskutiert dies in Abschnitt 16.1.3. Der Einsatz eines Aufzählungstyps stellt eine Lösung dar. Eine Schritt-für-Schritt-Anleitung liefert das in Abschnitt 17.4.12 beschriebene Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM. Neben dieser offensichtlichen Korrektur sind zwei weitere Verbesserungen denkbar:

1. Das Einführen eines booleschen Rückgabewerts, der über den (Miss-)Erfolg beim Verbindungsaufbau informiert.
2. Das Auslösen einer Exception, um eine geordnete Fehlerbehandlung im aufrufenden Programmteil zu ermöglichen. Hier könnte man eine `IllegalArgumentException` nutzen oder eine selbst definierte `UnsupportedTransmissionModeException`.

Tipp: Aufräumarbeiten und Shut-down-Hooks

Über einen sogenannten ***Shut-down-Hook*** können belegte Systemressourcen sauber wieder freigegeben werden – dies gilt selbst dann, wenn ein Aufruf von `System.exit(int)` mitten im Programmablauf erfolgt. Ein Shut-down-Hook ist eine spezielle Implementierung eines Threads. Beim Aufruf von `System.exit(int)` wird dieser Thread vor dem Ende der JVM ausgeführt. Gleiches gilt beim Beenden eines Programms durch eine nicht gefangene Exception. Wird die JVM jedoch anders beendet (etwa manuell über »Prozess beenden«), so ist nicht garantiert, dass ein Shut-down-Hook ausgeführt wird.

Idealerweise definiert man sich in denjenigen eigenen Klassen, die Ressourcen freigeben sollen, eine Methode `onShutdown()`, die die Aufräumarbeiten implementiert. Des Weiteren initialisiert man möglichst beim Programmstart oder kurz nach der Erzeugung eines Objekts einen Shut-down-Hook, sodass alle folgenden Aufrufe an `System.exit(int)` eine geordnete Terminierung erlauben.

Ein entsprechendes Programmfragment könnte so aussehen:

```
Runtime.getRuntime().addShutdownHook(new Thread()
{
    public void run()
    {
        onShutdown();
    }
});
```

16.1.13 Bad Smell: Variablendeklaration nicht im kleinstmöglichen Sichtbarkeitsbereich

Manchmal sieht man die Deklaration oder Definition diverser Variablen am Methodenanfang. In C++ bzw. C ist dies eine beliebte Technik, um alle verwendeten Variablen an zentraler Stelle sichtbar zu machen.

Ein Beispiel dafür ist folgende etwas umständlich arbeitende Methode `getImageForName(String)`, die zu einem übergebenen Namen einer Bilddatei ein passendes Bild vom Typ `Image` ermittelt und bei der Suche die Groß- und Kleinbuchstaben außer Acht lässt. Dazu wird eine Liste `imageNames` nach einem passenden Namen durchsucht und gegebenenfalls auf eine zweite Liste `images` mit Bilddaten zugegriffen. Dort sind die Daten jedoch um einen Wert `DEFAULT_IMAGE_OFFSET` versetzt gespeichert. Hier werden vier Variablen am Methodenanfang definiert:

```
private static Image getImageForName(final String imageNameToSearch)
{
    boolean found = false;
    String imageName = null;
    Image image = null;
    int index = 0;

    for (int i = 0; i < imageNames.size() && found == false; i++)
    {
        imageName = imageNames.get(i);
        if (imageName.equalsIgnoreCase(imageNameToSearch))
        {
            found = true;
            index = i + DEFAULT_IMAGE_OFFSET;
            image = images.get(index);
        }
    }

    return (found ? image : NO_IMAGE);
}
```

Alle vier Variablen werden erst im Schleifenrumpf mit aussagekräftigen Werten belegt, wobei drei Variablen sogar nur dann besetzt werden, wenn der gesuchte Name in der übergebenen Liste `imageNames` vorhanden ist.

Warum ist das ein Bad Smell? Die Definition von Variablen zu Methodenbeginn ist nicht mehr zeitgemäß. Die Begründung ist relativ einfach: Eine solche Initialisierung macht nicht klar, wo eine Variable verwendet wird, und zudem vergrößert sich die Gefahr, dass eine Variable für mehrere, unabhängige Zwecke eingesetzt wird. Außerdem ist meistens noch keine sinnvolle Vorbelegung möglich. Des Weiteren kann eine solche Variable nicht als `final` deklariert werden, selbst wenn sie es bei der Deklaration am passenden Ort sein könnte. ***Ein ganz entscheidender Nachteil jedoch ist, dass diese Art der Variablendeklarationen Refactorings erschwert, die gewisse Funktionalität in eine separate Methode verlagern***.

Tipps und Refactorings Variablen sollten so lokal wie möglich deklariert werden, also in dem kleinstmöglichen Sichtbarkeitsbereich. Dies hilft, schnell zu erkennen, welche Variablen an welcher Stelle im Sourcecode eingesetzt oder verändert werden. Sofern von Coding Conventions vorgegeben, können anschließend sämtliche Variablen, die sich nicht ändern, `final` definiert werden. Dadurch verhindert man den Missbrauch oder ein versehentliches Benutzen der Variablen für andere Zwecke. Durch die verringerten Abhängigkeiten werden Refactorings und das Herauslösen von Methoden erleichtert.

Nehmen wir an dem zuvor gezeigten Beispiel einige Änderungen vor: Wir verschieben die Variablendeklarationen in den kleinstmöglichen Sichtbarkeitsbereich und vereinfachen den Sourcecode weiter. Man erkennt beispielsweise, dass die Variable `index` immer den Wert `i + DEFAULT_IMAGE_OFFSET` besitzt. Auch die Variable `found` ist künstlich und steuert den Schleifenabbruch. In dieser kurzen Methode profitiert man vom Einsatz eines Shortcut-Returns, der die Variable `found` überflüssig macht. Durch den Shortcut-Return kann auch auf die Variable `image` verzichtet werden. Es bleibt lediglich die Variable `imageName` übrig. Sie wird direkt zu ihrer Verwendung verschoben, wodurch die Methode kürzer, übersichtlicher und lesbarer wird:

```
private static Image getImageForNameImproved(final String imageNameToSearch)
{
    for (int i = 0; i < imageNames.size(); i++)
    {
        final String imageName = imageNames.get(i);
        if (imageName.equalsIgnoreCase(imageNameToSearch))
        {
            return images.get(i + DEFAULT_IMAGE_OFFSET);
        }
    }

    return NO_IMAGE;
}
```

Mögliche weitere Verbesserungen Nach diesem erfolgreichen Umbau erkennt man vielleicht auch, dass eine Map eine geeignetere Datenstruktur wäre. Mithilfe von `getOrDefault()` wäre dann auch die selbst ausprogrammierte Spezialbehandlung obsolet:

```
public static final Map<String, Image> nameToImageMapping = new HashMap<>();


private static Image getImageForName(final String imageNameToSearch)
{
    return nameToImageMapping.getOrDefault(imageNameToSearch, NO_IMAGE);
}
```

An diesem Beispiel erkennt man recht gut, dass Aufräumarbeiten nicht nur strukturelle Verbesserungen und eine gestiegene Wartbarkeit mit sich bringen, sondern dass sich mitunter ganz neue Möglichkeiten zur Vereinfachung und für mehr Klarheit bieten.

16.2 Klassendesign

16.2.1 Bad Smell: Unnötigerweise veränderliche Attribute

In der täglichen Arbeit kommt es immer wieder zu Änderungen in Klassen, um diese an neue Anforderungen anzupassen. Mitunter werden dazu Attribute ergänzt und ohne weitere Bedenken korrespondierende `get()`- und `set()`-Methoden zum Zugriff angeboten.

Betrachten wir exemplarisch eine Klasse `Telegram`, bei der dies der Fall ist. Die Erzeugung eines `Telegram`-Objekts erfolgt per Defaultkonstruktor mit anschließender Initialisierung durch den Aufruf einiger `set()`-Methoden, die das Objekt dann vollständig »gebrauchsfertig« machen:

```
// => Konstruktion mit (eventuell unsinnigen) Defaultwerten
final Telegram telegram = new Telegram();
// => Initialisierung
telegram.setTransmissionMode(TransmissionMode.TCP);
telegram.setReceiver(hostAndPort);
telegram.setPayload(data);
// => hier erst korrektes Objekt !!!
```

Warum ist das ein Bad Smell? Der Sourcecode sieht auf den ersten Blick nicht besonders fehleranfällig aus. Problematisch ist jedoch, dass zunächst durch den Konstruktoraufruf ein Objekt erzeugt wird, das diverse Attribute lediglich mit Defaultwerten versorgt. Wird nur der Konstruktor aufgerufen und erfolgen anschließend keine weiteren Initialisierungen, so verbleibt das Objekt in einem möglicherweise ungültigen Zustand (vgl. Abschnitt 3.1.5). Auf einer solchen Objektreferenz könnten außerdem bereits Methoden aufgerufen werden – ein korrektes Verhalten ist jedoch höchst unwahrscheinlich. Die anschließende mehrstufige Initialisierung über den Aufruf einzelner `set()`-Methoden ist zudem problematisch, weil dadurch weder sichergestellt werden kann, dass alle zur Objektkonstruktion benötigten Eingabewerte tatsächlich über entsprechende `set()`-Methoden gesetzt wurden, noch, dass eine gegebenenfalls erforderliche Reihenfolge der Initialisierung eingehalten wird.

Eine standardmäßige Definition von `set()`-Methoden verhindert, dass Attribute `final` deklariert werden können, obwohl diese eigentlich unveränderlich sind. Häufig sieht man zusätzlich bei der Deklaration eine Zuweisung eines unsinnigen, weil beliebigen Werts, wie dies folgendes Beispiel demonstriert:

```
public class Telegram
{
    private String           receiver         = "undefined";
    private TransmissionMode transmissionMode = TransmissionMode.UNDEFINED;
    private byte[]           payload          = null;

    public void setReceiver(final String receiver)
    {
        this.receiver = receiver;
    }
```

```
    public void setTransmissionMode(final TransmissionMode transmissionMode)
    {
        this.transmissionMode = transmissionMode;
    }

    public void setPayload(final byte[] payload)
    {
        this.payload = payload;
    }
}
```

Solche nichtssagenden (unsinnigen) Werte sind nur bis zum Aufruf einer entsprechenden `set()`-Methode »gültig«. Erfolgt allerdings keine nachträgliche Initialisierung eines »pseudoinitialisierten« Attributs mit einem sinnvollen Wert, so liegt eine unvollständige Initialisierung vor, die jedoch nur schwierig zu erkennen ist.

Tipps und Refactorings Durch Codegeneratoren oder aus Unachtsamkeit werden manchmal öffentliche `set()`-Methoden für Attribute angeboten, die eigentlich unveränderlich sein könnten. Allerdings sollten `set()`-Methode nur bei tatsächlichem Bedarf eingeführt werden. Dabei gibt es jedoch Ausnahmen, wenn Frameworks wie JPA (Java Persistence API) oder JAX-RS derartige Methoden zur korrekten Funktionsweise erfordern.

Nach der Abarbeitung des Konstruktors sollte ein Objekt in einem gebrauchsfertigen Grundzustand sein. Konstruktoren müssen daher alle dafür notwendigen Eingabeparameter entgegennehmen. Existiert noch kein solcher Konstruktor, sollte man einen solchen einführen und die Klasse gemäß dem Refactoring MINIMIERE VERÄNDERLICHE ATTRIBUTE aus Abschnitt 17.4.2 anpassen. Dadurch werden falsch oder nur teilweise initialisierte Objekte und damit ungültige Objektzustände vermieden. Die `Telegram`-Klasse kann so umgewandelt werden, dass lediglich die Nutzlast (Attribut `payload`) variabel und über eine `setPayload(byte[])`-Methode änderbar ist:

```
public class Telegram
{
    private final String           receiver;
    private final TransmissionMode transmissionMode;
    private byte[]                 payload = null;

    public Telegram(final String receiver,
                    final TransmissionMode transmissionMode)
    {
        this.receiver = receiver;
        this.transmissionMode = transmissionMode;
    }

    public void setPayload(final byte[] payload)
    {
        this.payload = payload;
    }
}
```

Zur Abhilfe kann auch ein Blick auf die Muster FABRIKMETHODE oder ERBAUER (vgl. Abschnitt 18.1.2 und 18.1.3) hilfreich sein.

16.2.2 Bad Smell: Herausgabe von `this` im Konstruktor

Eine Technik, die man häufiger sieht und die dabei harmloser wirkt, als sie in Wirklichkeit ist, ist die Übergabe der `this`-Referenz aus eigenen Konstruktoren an andere Objekte. Dies wird als »***Escaping Reference***« bezeichnet.

Im folgenden Beispiel betrachten wir den Konstruktionsprozess der Klasse `MainService`, die Referenzen auf die Klassen `CallBack` und `String` als Attribute hält:

```
public class MainService
{
    private final CallBack caller;
    private final String   info;

    public MainService()
    {
        caller = new CallBack(this);
        info = "Initialized " + MainService.class.getSimpleName();
    }

    public String getInfo()
    {
        return info;
    }
}
```

Im Konstruktor der Klasse `MainService` werden jeweils Objekte der Klassen `CallBack` und `String` erzeugt, wobei der Konstruktor von `CallBack` wie folgt definiert ist:

```
CallBack(final MainService mainService)
{
    this.mainService = mainService;

    // Zugriff auf Methode der teilinitialisierten(!) Klasse MainService
    final byte[] infoAsBytes = mainService.getInfo().getBytes();
}
```

Warum ist das ein Bad Smell? Bei der Herausgabe der `this`-Referenz verhält sich auf den ersten Blick korrekt erscheinender Sourcecode merkwürdig. Es treten Probleme durch falsch oder nicht initialisierte Attribute auf.

Konkretisieren wir dies an obigem Beispiel: Der Aufruf des Konstruktors der Klasse `CallBack` erhält als Eingabe die `this`-Referenz des gerade in der Erzeugung befindlichen `MainService`-Objekts. Zu diesem Zeitpunkt ist die Konstruktion der Klasse `MainService` allerdings noch nicht vollständig abgeschlossen, es steht noch die Initialisierung des Attributs `info` aus. ***Während der Konstruktion der Klasse `MainService` ist es der Klasse `CallBack` allerdings bereits möglich, auf alle öffentlichen Daten und Methoden der Klasse `MainService` zuzugreifen.*** Im Speziellen gilt dies auch für das Attribut `info`. Wenn die Klasse `CallBack` die Methode `getInfo()` der Klasse `MainService` aufruft, so kommt es durch die nicht initialisierte Referenz `info` zu einer `NullPointerException` beim Zugriff per `getBytes()`.

Tipps und Refactorings Durch die Herausgabe der `this`-Referenz können andere Objekte ein unvollständig initialisiertes Objekt sehen und auf dessen Attribute zugreifen. ***»Escaping References« sind daher zu vermeiden***. Es bietet sich der Einsatz des Musters ERZEUGUNGSMETHODE aus Abschnitt 18.1.1 an. Man kann dann ein Objekt zunächst ohne aggregierte Objekte erzeugen. Anschließend werden die Konstruktoren der aggregierten Objekte aufgerufen, die die Referenz benötigen. Die eigene Klasse muss allerdings um entsprechende `set()`- bzw. `init()`-Methoden erweitert werden, um die ansonsten im Konstruktor erzeugten Objekte zu initialisieren.

Achtung: »Escaping References« und »Immutable Objects«

»Immutable Objects« sollen zu ihrer Lebenszeit nur einen definierten Zustand besitzen, der zum Konstruktionszeitpunkt bestimmt wird. Reicht ein solches »Immutable Object« allerdings seine `this`-Referenz während seiner Erzeugung nach außen, kann es dadurch mehrere sichtbare Zustände einnehmen. »Escaping References« führen den Einsatz von »Immutable Objects« ad absurdum: Trotz der Realisierung als unveränderliche Klasse kann man Zwischenschritte sehen.

16.2.3 Bad Smell: Aufruf abstrakter Methoden im Konstruktor

Manchmal wünscht man sich, in Basisklassen einen gewissen Ablauf bei der Initialisierung vorgeben zu können, etwa den Aufruf einer speziellen `init()`-Methode aus dem Konstruktor. Die konkrete Realisierung der Initialisierung ist in der Basisklasse jedoch noch unbekannt und soll von Subklassen realisiert werden. Um dieses Ziel zu erreichen, erfolgt dann der Aufruf einer abstrakten Methode `init()` aus dem Konstruktor der Basisklasse.

Wieso dieses zunächst sinnvoll erscheinende Vorgehen problematisch sein kann, betrachten wir an folgendem Beispiel: Die Basisklasse `AbstractBase` definiert eine Variable `baseValue` und deklariert eine abstrakte Methode `init()`, die in deren Konstruktor aufgerufen wird. Die abgeleitete Klasse `Derived` definiert eine Variable `value` und implementiert diese `init()`-Methode und gibt dort die Werte der Variablen `baseValue` sowie `value` aus:

```
public final class AbstractMethodInConstructorExample
{
    abstract static class AbstractBase
    {
        protected final Integer baseValue = 42;

        AbstractBase()
        {
            // Aufruf der Initialisierung
            init();
        }

        abstract void init();
    }
```

```
    static class Derived extends AbstractBase
    {
        private final Integer value = 13;

        Derived()
        {
        }

        void init()
        {
            // Zugriff auf Attribut der Basisklasse
            System.out.println("baseValue = " + baseValue);

            // Zugriff auf Attribut dieser Klasse
            System.out.println("value = " + value);
        }
    }

    public static void main(String[] args)
    {
        // Konstruktion zur Demonstration der Probleme
        new Derived();
    }
}
```

Listing 16.1 *Ausführbar als* '**ABSTRACTMETHODINCONSTRUCTOREXAMPLE**'

Intuitiv erwartet man die Ausgabe der Zahlen `42` und `13`. Schauen wir mal, ob Intuition und Wirklichkeit zusammenpassen, und führen das Programm ABSTRACTMETHODINCONSTRUCTOREXAMPLE aus. Wir erhalten folgende Ausgabe:

```
baseValue = 42
value = null
```

Wie ist das zu erklären? Folgen wir dem Programmfluss, so ist der Ablauf wie folgt: In der Methode `main()` wird ein Objekt der Klasse `Derived` konstruiert: Der Aufruf des Defaultkonstruktors ruft wiederum den Konstruktor der Basisklasse auf (Details zum Ablauf sind in folgendem Hinweiskasten beschrieben). Die dortige Ausführung der `init()`-Methode führt aufgrund der Polymorphie und des dynamischen Bindens zum Aufruf der `init()`-Methode der Klasse `Derived`. Da die Variable `value` zu diesem Zeitpunkt aber noch nicht zugewiesen wurde – der Konstruktor von `Derived` ist noch nicht vollständig abgearbeitet –, wird der Wert `null` ausgegeben. Erst nach Beendigung des Aufrufs der Methode `init()` wird auch die Abarbeitung des Basisklassenkonstruktors abgeschlossen. Danach wird erst das Attribut `value` als Teilschritt der Konstruktion und Initialisierung der Klasse `Derived` auf den Wert `13` gesetzt.

Warum ist das ein Bad Smell? Der auf den ersten Blick korrekte Sourcecode verhält sich unerwartet und es treten merkwürdige Effekte durch uninitialisierte Variablen auf. Daher ist der Aufruf abstrakter Methoden im Konstruktor oftmals ein Problem. ***Noch schwerer nachvollziehbar ist dies, wenn der Konstruktor eine konkrete `init()`-Methode verwendet und darin abstrakte Methoden aufgerufen werden.***

Tipps und Refactorings Abstrakte Methoden sollten niemals im Konstruktor verwendet werden. Man umgeht Probleme, indem man eine `init()`-Methode einführt und diese nach der Objektkonstruktion aufruft. Es bietet sich der Einsatz der Muster ERZEUGUNGSMETHODE oder FABRIKMETHODE (vgl. Abschnitt 18.1.1 und 18.1.2) an, um diesen Ablauf sicherzustellen.

Durch die Refactorings ÜBERPRÜFE EINGABEPARAMETER (Abschnitt 17.5.2) und FÜHRE EINE ZUSTANDSPRÜFUNG EIN (Abschnitt 17.5.1) erreicht man zum einen die Prüfung von Eingabeparametern sowie zum anderen im Idealfall die Sicherstellung eines gültigen Objektzustands. Wird im Falle einer fehlerhaften Initialisierung eine Exception ausgelöst, so erhält man Hinweise zur Lösung zuvor möglicherweise unerklärlicher Fehler.

Hinweis: Ablauf der Objektkonstruktion

Folgende Schritte laufen beim Konstruktionsprozess eines Objekts ab:

1. Bevor ein Objekt einer Klasse per Konstruktor erzeugt wird, wird die Klassenbeschreibung von einem sogenannten `ClassLoader` geladen, wenn die Klasse bisher unbenutzt war.
2. Der Konstruktionsprozess wird begonnen. Dazu wird zunächst der Speicherplatz für die Attribute bereitgestellt und diese mit ihrem Defaultwert initialisiert. Dies geschieht ausgehend von der Klasse `Object` für alle Basisklassen bis zur eigentlich zu konstruierenden Klasse.
3. Für alle Klassen erfolgen in umgekehrter Reihenfolge (von `Object` bis zur eigenen Klasse) folgende Schritte:
   a. Alle Instanzvariablen, die in ihrer Deklaration einen initialen Wert angegeben haben, werden nun zugewiesen.
   b. Anschließend werden die Instanz-Initializer ausgeführt. Diese kann man beispielsweise für aufwendigere Berechnungen der initialen Werte einsetzen. Beim Erzeugen eines Objekts kommen die Initializer entsprechend ihrer Reihenfolge im Sourcecode zur Ausführung. Erst nach dem letzten Initializer werden die Anweisungen im Konstruktor ausgeführt.
   c. Der Konstruktor wird abgearbeitet und es erfolgen die Zuweisungen und Methodenaufrufe, so wie sie im Konstruktor stehen.

Bei einer Ableitungshierarchie wird das Objekt sozusagen Stück für Stück aus seinen Basisklassenbestandteilen zusammengesetzt, und zwar beginnend bei der obersten Basisklasse `Object`.

Variante

Dieses Beispiel zeigt ein mögliches Initialisierungsproblem im Zusammenhang mit Threads. Betrachten wir folgendes Konstrukt, das eine Utility-Klasse `ThreadAutoStart` definiert, die einen Thread erzeugt und diesen automatisch startet:

```
public class ThreadAutoStart implements Runnable
{
    public ThreadAutoStart()
    {
        new Thread(this).start();
    }
}

public class ErroneousDataAccessThread extends ThreadAutoStart
{
    private final DataService service;

    public ErroneousDataAccessThread(final DataService service)
    {
        // Aufwendige Initialisierung
        // ...
        this.service = service;
    }

    @Override
    public void run()
    {
        // Möglicherweise hier schon Zugriff, weil Initialisierung
        // im Konstruktor noch nicht abgeschlossen => NullPointerException
        final SomeData data = service.retrievData(...);
        // ...
    }
}
```

Werden Klassen analog zu `ErroneousDataAccessThread` von der Basisklasse `ThreadAutoStart` abgeleitet, würde bereits während einer Objektkonstruktion ein Thread gestartet und dessen `run()`-Methode ausgeführt. Allerdings wären zu dem Zeitpunkt die Subklassenbestandteile noch nicht vollständig konstruiert. Dadurch sind vergleichbare Probleme wie die zuvor geschilderten möglich.

Verstecktere Initialisierungsprobleme

Im folgenden Beispiel findet sich ein ähnlicher Fehler, der durch eine zusätzliche Indirektion allerdings schwieriger zu erkennen ist. Die Basisklasse `CommunicationBase` definiert eine abstrakte Methode `createComponents()`, die von ihren Subklassen überschrieben werden muss, um Kommunikationskomponenten zu erzeugen:

```
public abstract class CommunicationBase
{
    CommunicationBase()
    {
        createComponents();
    }

    abstract protected void createComponents();
}
```

Eine Klasse `RadioCommunication` realisiert eine Funkkommunikation und erzeugt in der Methode `createComponents()` jeweils einen Sender und einen Empfänger. Um eine unidirektionale Kommunikation zu erreichen, also exklusiv nur zu empfangen oder zu senden, kann man auf die Idee kommen, ein gemeinsames Synchronisationsobjekt `sharedSyncObject` folgendermaßen zu definieren und zu nutzen:

```
public final class RadioCommunication extends CommunicationBase
{
    // Definition des Synchronisationsobjekts
    private final Object  sharedSyncObject = new Object();
    private RadioSender   dataSender       = null;
    private RadioReceiver dataReader       = null;

    protected final void createComponents()
    {
        // Übergabe des Synchronisationsobjekts
        dataSender = new RadioSender(sharedSyncObject);
        // ...
```

Die `RadioSender`-Klasse erhält das gemeinsame Synchronisationsobjekt im Konstruktor und nutzt es, um ein gleichzeitiges Senden und Empfangen zu unterbinden, in der Methode `send(byte[])` wie folgt:

```
public final class RadioSender
{
    private final Object sharedSyncObject;

    public RadioSender(final Object sharedSyncObject)
    {
        this.sharedSyncObject = sharedSyncObject;
    }

    public void send(final byte[] msg)
    {
        synchronized (sharedSyncObject)
        {
            sendBytes();
        }
    }
    //...
```

Zur Laufzeit beobachten wir allerdings eine `NullPointerException` in der Zeile mit der Anweisung `synchronized (sharedSyncObject)`. Zunächst scheint der Fehler unerklärlich: Das `sharedSyncObject` wird während der Objekterzeugung initialisiert und ist zudem `final`. Wie kann es dann `null` sein?

Eine Prüfung des Eingabeparameters im Konstruktor der Klasse `RadioSender` hätte einen Hinweis darauf gegeben, dass statt der erwarteten Referenz auf das Synchronisationsobjekt tatsächlich der Wert `null` übergeben wurde. Erinnern wir uns an die Diskussion bezüglich der Initialisierung: Das `RadioCommunication`-Objekt ist zu dem Zeitpunkt nicht vollständig konstruiert, zu dem der Aufruf von `createComponents()` aus dem Konstruktor der Basisklasse `CommunicationBase` erfolgt. Insbesondere ist auch das Attribut `sharedSyncObject` noch nicht korrekt initialisiert, sondern besitzt nur den Defaultwert `null`.

16.2.4 Bad Smell: Mix abstrakter und konkreter Basisklassen

In komplexen Vererbungshierarchien werden Basisklassen zum Teil nicht abstrakt definiert, obwohl sich dies empfehlen würde, weil man davon ausgehen sollte, dass die Basisklassen noch nicht alle Funktionalitäten enthalten. Eine Instanziierung solcher Klassen ist zwar technisch möglich, aber aus Anwendungssicht wenig sinnvoll.

Betrachten wir ein Beispiel einer derartigen Ableitungshierarchie, dargestellt als UML-Diagramm in Abbildung 16-2.

Abbildung 16-2 *Mix abstrakter und konkreter Basisklassen*

Die konkrete Basisklasse `BaseProcess` definiert die Basisfunktionalität für einen Prozessschritt und erlaubt mit den Methoden `getProcessState()` und `setProcessState(ProcessState)` eine Verarbeitung von Zustandsinformationen. Die davon abgeleitete Klasse `AbstractProcess` ist abstrakt und deklariert Methoden zum Starten, Anhalten und Fortsetzen eines Prozessschritts. All diese Methoden sind abstrakt, weil die Realisierungen der auszuführenden Aktionen in der Basisklasse noch unbekannt sind und in den Subklassen definiert werden müssen. Nur diese »wissen«, welche Zustandsinformationen beim Anhalten zu speichern sind, um ein Fortsetzen der Verarbeitung sicherzustellen. Die konkreten Subklassen `ProcessA` und `ProcessB` definieren die Methoden `pause()`, `resume()` und `start()` entsprechend ihren Anforderungen.

Warum ist das ein Bad Smell? Bei einer Klasse, die nicht abstrakt definiert ist, sollte man als Klient davon ausgehen können, dass diese eigenständig zu instanziieren und zu benutzen ist. Basisklassen nicht abstrakt zu machen, birgt also die Gefahr, dass diese (versehentlich) instanziiert werden und zum Einsatz kommen. In der Regel wollen wir dies jedoch nicht erlauben, da solche Basisklassen kein funktional vollständiges Objekt darstellen. Eine Verwendung ist daher eigentlich nicht vorgesehen und

das Objektverhalten unbestimmt. Noch verwirrender wird das Ganze, wenn in einer Vererbungshierarchie die Basisklassen teilweise abstrakt sind, teilweise nicht. Welche Schlussfolgerungen soll man daraus ziehen? Eine Basisklasse ist für sich allein einsatzfähig, aber eine davon abgeleitete Subklasse nicht? Das macht wenig Sinn.

Tipps und Refactorings Es bieten sich folgende zwei Lösungsmöglichkeiten an:

1. In diesem Beispiel implementiert die vermeintliche Basisklasse `BaseProcess` nur Hilfsmethoden zur Statusverwaltung eines Prozesses. Das stellt einen Designfehler dar, weil Vererbung lediglich zur Übernahme aus der Basisklasse benötigter Funktionalität eingesetzt wird. Für die *konkrete* Klasse `BaseProcess` und ihre *abstrakte* Subklasse `AbstractProcess` liegt keine »is-a«-Beziehung vor. Als Korrektur wird die Vererbungsbeziehung aufgelöst und die Klasse `BaseProcess` in eine Utility-Klasse umgewandelt. Durch Umbenennen in `ProcessStateUtils` wird deren Utility-Charakter verdeutlicht. Die Funktionalität wird dann über eine »has-a«-Beziehung, die durch eine Assoziation ausgedrückt werden kann, referenziert. Eine mögliche Realisierung ist in Abbildung 16-3 dargestellt.

Abbildung 16-3 *Abhilfe durch Einführen einer Utility-Klasse*

2. Nehmen wir an, es läge tatsächlich eine »is-a«-Beziehung vor und die Funktionalität wäre eher »künstlich« oder willkürlich auf die Basisklassen verteilt. In diesem Fall sollten die beiden Basisklassen zu einer gemeinsamen verschmolzen werden. Für dieses Beispiel kann die gesamte Funktionalität aus der dann überflüssigen Basisklasse `BaseProcess` in die abstrakte Klasse `AbstractProcess` integriert werden. Mitunter besteht eine sinnvolle Erweiterung darin, ein Interface `IProcess` zu erzeugen. Dieses definiert die Methoden aus der Klasse `BaseProcess` sowie die abstrakten Methoden `pause()`, `resume()` und `start()`. Durch den Einsatz wird eine bessere Kapselung und damit auch eine losere Kopplung erreicht. Eine mögliche Umsetzung zeigt Abbildung 16-4.

Abbildung 16-4 *Abhilfe durch Interface und Zusammenfassen von Funktionalität*

16.2.5 Bad Smell: Referenzierung von Subklassen in Basisklassen

Entsprechend dem OO-Gedanken der Generalisierung definieren Basisklassen gemeinsame Funktionalität potenzieller Subklassen. In diesen Spezialisierungen sollten lediglich Unterschiede zur Basisklasse beschrieben werden. Teilweise sieht man aber Klassen, die ihre Spezialisierungen kennen.

Schauen wir uns dies in Abbildung 16-5 anhand der Basisklasse `DataManager` an, die drei konkrete Subklassen `DataManagerA` bis `DataManagerC` besitzt. In der `main()`-Methode der Klasse `DataManager` wird abhängig von einem Konfigurationsparameter `managerType` die jeweils zu erzeugende Subklasse gewählt.

Warum ist das ein Bad Smell? Basisklassen sollten ihre Spezialisierungen nicht kennen. Andernfalls sind Anpassungen in der Basisklasse nötig, wenn neue Subklassen definiert werden. Dies widerspricht dem OO-Gedanken der Basis- und Subklassen und dem Herausfaktorieren gemeinsamer Funktionalität in Basisklassen.

Eine Referenzierung von Subklassen in Basisklassen ist unelegant, wird aber leider auch in einigen Klassen des JDKs verwendet: Die Klasse `NumberFormat` kennt ihre Spezialisierung `DecimalFormat` und erzeugt sogar Instanzen davon.

Abbildung 16-5 *Referenzierung von Subklassen in der Basisklasse*

Tipps und Refactorings Durch den Einsatz des FABRIKMETHODE-Musters (vgl. Abschnitt 18.1.2) kann man das Problem der Referenzierung von Subklassen lösen. Nachfolgend wird die Klasse `DataManager` dazu in drei Bestandteile zerlegt. Das ist sinnvoll, da sie ursprünglich für zu viele Dinge verantwortlich war, nämlich für das Starten des Programms, für die Auswahl der konkreten Subklasse und für die Bereitstellung von Funktionalität als Basisklasse. Nur Letzteres verbleibt im neuen Design bei ihr. Zum Applikationsstart wird die `main()`-Methode in eine eigenständige Klasse `DataManagerStarter` verlagert. Die Auswahl und das Erzeugen von Spezialisierungen erledigt die Klasse `DataManagerFactory` (vgl. Abbildung 16-6).

Abbildung 16-6 *Einführen einer Starter-Klasse und einer Fabrikmethode*

Abschließend möchte ich noch erwähnen, dass eine weitere Verbesserung darin bestünde, statt eines Strings einen `enum` als Konfigurationsparameter zu verwenden.

16.2.6 Bad Smell: Öffentlicher Defaultkonstruktor lediglich zum Zugriff auf Hilfsmethoden

Man sieht häufig Klassen, in denen ein öffentlicher Defaultkonstruktor existiert. Abgesehen von Anforderungen durch Frameworks wird für reine Applikationsklassen ein solcher Defaultkonstruktor aber eher selten benötigt. Zum Teil ist dessen Existenz lediglich dadurch begründet, dass ein Zugriff auf Konstanten oder Hilfsmethoden erfolgen soll, die fälschlicherweise nicht statisch definiert sind.

Betrachten wir dies am Beispiel der folgenden Utility-Klasse `FigureDrawUtils`. Hier werden eine Hilfsmethode `drawBorder(Graphics, Color, BaseFigure)` sowie eine Konstante `BORDER_COLOR` definiert – beide sind jedoch nicht statisch:

```
public final class FigureDrawUtils
{
    public final Color BORDER_COLOR = Color.RED;

    public FigureDrawUtils()
    {}

    public void drawBorder(final Graphics graphics, final Color borderColor,
                           final BaseFigure figure)
    {
        graphics.setColor(borderColor);
        graphics.drawRect(figure.getX(), figure.getY(),
                          figure.getWidth(), figure.getHeight());
    }
}
```

Möchte man die Konstante `BORDER_COLOR` oder die gezeigte Hilfsmethode verwenden, um beispielsweise einen Selektionsrahmen zu zeichnen, so benötigt man ein (beliebiges) `FigureDrawUtils`-Objekt. Im Extremfall würde für jeden Zugriff eine Objektkonstruktion erfolgen:

```
new FigureDrawUtils().drawBorder(graphics,
                                 new FigureDrawUtils().BORDER_COLOR,
                                 circleFigure);
```

Warum ist das ein Bad Smell? Zum Zugriff auf nicht statische Konstanten oder Utility-Methoden muss man entweder bereits ein Objekt der Klasse referenzieren oder dafür sogar ein neues »Dummy«-Objekt erzeugen. Letzteres ist problematisch, da eventuell in einem Nutzungskontext kein Objekt erzeugt werden kann, beispielsweise, weil zu übergebende Werte für Konstruktorparameter nicht sinnvoll gewählt werden können. Als vermeintliche »Lösung« wird aus diesem Grund dann zum Teil ein Defaultkonstruktor bereitgestellt, ohne die daraus resultierenden Konsequenzen zu bedenken: Ein später hinzugefügter Defaultkonstruktor birgt die Gefahr, Objekte ohne sinnvollen Initialisierungszustand erzeugen zu können. Mögliche Auswirkungen davon wurden bereits in Abschnitt 16.2.1 als BAD SMELL: UNNÖTIGERWEISE VERÄNDERLICHE ATTRIBUTE angesprochen.

Ein solches Vorgehen provoziert einerseits Inkonsistenzen, andererseits leiden die Verständlichkeit sowie die Lesbarkeit. Inkonsistenzen entstehen dadurch, dass der Zugriff auf Methoden und Attribute möglich ist, obwohl derart erzeugte Objekte häufig keinen gültigen Zustand besitzen. Die Lesbarkeit wird eingeschränkt, weil Objektkonstruktionen notwendig werden, um auf diese nicht statischen Utility-Methoden zuzugreifen. Ein solcher Konstruktor führt demnach zu hässlichem und unverständlichem Sourcecode.

Tipps und Refactorings Existiert ein öffentlicher Defaultkonstruktor ohne Initialisierungen nur zum »bequemeren« Zugriff auf Hilfsmethoden oder Konstanten, so empfiehlt es sich, die Utility-Methoden und Konstanten statisch zu deklarieren, um einen Zugriff ohne vorherige Objektkonstruktion zu ermöglichen. Anschließend kann der Defaultkonstruktor oft entfernt oder als `private` deklariert werden.

Diese Transformation folgt der Guideline, dass ***Methoden, die keinen Objektzustand abfragen oder modifizieren, statisch sein sollten und gegebenenfalls in eine separate Utility-Klasse ausgelagert werden können.*** Das dazu notwendige Vorgehen wird als Refactoring WANDLE IN UTILITY-KLASSE MIT STATISCHEN HILFSMETHODEN UM in Abschnitt 17.4.14 beschrieben.

16.3 Fehlerbehandlung und Exception Handling

16.3.1 Bad Smell: Unbehandelte Exception

Unbehandelte Exceptions können verschiedenste Probleme verursachen. Durch automatische Sourcecode-Vervollständigungen entstehen etwa leere `catch`-Blöcke. Oft findet sich dort ein `TODO`-Kommentar und ein `printStacktrace()`. Noch schlimmer ist das »Verschlucken« der Exception wie im folgenden Beispiel:

```
catch (final RemoteException e)
{
    System.out.println("to make Volker happy :-)");
}
```

Ob man mit einer solchen Ausgabe, wie mit dem `System.out.println()` angedeutet, wirklich jemanden glücklich macht, kann man bezweifeln. Ich konnte dem Entdecken dieser Ausgabe im Produktionscode jedenfalls nicht wirklich Freude abgewinnen, denn ich durfte eine sinnvolle Fehlerbehandlung implementieren.

Warum ist das ein Bad Smell? Bleiben Exceptions unbehandelt, so erfährt der Programmanwender eventuell niemals von möglichen Problemen. Eine `RemoteException`, wie im obigen Beispiel, deutet aber auf ein Netzwerkproblem hin. Derartige Probleme sollten dem Benutzer kommuniziert oder zumindest für eine spätere Analyse in eine Log-Datei geschrieben werden.

Tipps und Refactorings Man sollte Exceptions innerhalb der Methode, in der die Exception auftreten kann, behandeln, sofern eine sinnvolle Fehlerbehandlung möglich ist. Erfolgt keine Behandlung, so sollte die Exception weiter propagiert werden. Dadurch können höhere Applikationsschichten auf eine Fehlersituation reagieren. In der Regel ist dies dort zudem angemessener möglich als beispielsweise inmitten einer Berechnung.

Anhand von `IOException`s und `RemoteException`s kann man sich dies gut verdeutlichen: Bei einem Zugriffsproblem kann eine ausführende Methode mitunter nicht adäquat reagieren. Ein Aufrufer kann aber eine Wiederholung veranlassen oder eine neue Verbindung herstellen.

Ist zunächst offen, ob auf eine Fehlersituation angemessen reagiert werden kann oder eben nicht, so sollte man zumindest eine Log-Ausgabe durchführen. Sinnvoll ist es, einen `@TODO`-Kommentar als eine Erinnerung an die noch ausstehende Fehlerbehandlung im Sourcecode zu hinterlassen.

Eine Sache sollte man noch bedenken: Im Umfeld von Applikationsservern können während der Verarbeitung ausgelöste `Exception`s zu Rollbacks von Transaktionen führen und somit ungewollt den Programmablauf beeinflussen.

Achtung: Gefahr des Verschleierns

Eine Ausgabe von Exceptions in eine Log-Datei kann hilfreich sein. Als einzige Reaktion ist dies meistens jedoch für eine sinnvolle Fehlerbehandlung nicht ausreichend, da die Exception lautlos »versickert«. Allerdings ist ein solches Logging für eine nachträgliche Fehlersuche und -analyse hilfreich. Folgende Methode `storeTask(ITask)` liefert ein Beispiel für das »Versickern« einer `RemoteException`:

```
private void storeTask(final ITask task)
{
    try
    {
        taskService.storeTask(task);
    }
    catch (final RemoteException e)
    {
        log.error("Unable to store task in db! task = " + task, e);
    }
}
```

Im gezeigten Exception Handling fehlt eine weitere Behandlung der auftretenden `RemoteException`, die auf ein Verbindungsproblem hinweist. Nehmen wir an, in einer höheren Applikationsschicht existiere genau für solche Verbindungsprobleme eine eingebaute Logik zum Neuverbinden. Diese wird als Folge der fehlenden Propagation jedoch nicht mehr aufgerufen. Daher sollte in diesem Fall die Exception an die aufrufende Methode weitergeleitet werden. Im Beispiel könnte man auf das `try-catch` verzichten, jedoch müsste die Exception in die Signatur aufgenommen werden.

16.3.2 Bad Smell: Unpassender Exception-Typ

Beim Auftreten von Fehlern sollten diese möglichst sauber behandelt oder aussagekräftig propagiert werden. Im Falle von Exceptions ist besondere Aufmerksamkeit bei der Wahl des Exception-Typs erforderlich. Ein unpassend gewählter Exception-Typ erschwert eine Analyse des Programms im Fehlerfall.

Betrachten wir dazu als Beispiel folgende Methode `fill(String)`, in der auf unterschiedliche Fehlersituationen jeweils auf dieselbe Art und Weise reagiert wird: Eine fehlende Initialisierung des Attributs `startFolder` sowie die Belegung des Attributs `ascending` mit `false` lösen jeweils eine `NullPointerException` aus:

```
public void fill(final String path)
{
    if (this.startFolder != null)
    {
        if (this.ascending)
            fillFolderContent(path);
        else
            throw new NullPointerException("Sort order not implemented!");
    }
    else
    {
        throw new NullPointerException("No startfolder!");
    }
}
```

Nochmals zur Erinnerung, weil das Beispiel vielleicht konstruiert scheint: Das Ganze entstammt tatsächlich Produktivcode – jedoch leicht anonymisiert durch abgeänderte Variablennamen.

Warum ist das ein Bad Smell? Die hier für die Fehlerbehandlung gewählten `NullPointerException`s werden mit Informationstexten erzeugt und geben Hinweise auf eine mögliche Fehlerursache. Damit sind auftretende Fehler zwar anhand des Stacktrace relativ gut nachvollziehbar, allerdings ist der verwendete Exception-Typ irreführend. Es ist schlecht, durch eine `NullPointerException` eine unvollständige Implementierung der unterstützten Sortierreihenfolgen auszudrücken. In der Praxis werden zum Teil leider Exceptions ohne Hinweistext ausgelöst. Das erschwert eine mögliche Fehlersuche zusätzlich. Dieser Effekt verstärkt sich, wenn – wie hier – mehrfach der gleiche Exception-Typ Anwendung findet. Nachfolgende Änderungen anderer Entwickler können dazu führen, dass die Zeilennummern aus einem Stacktrace nicht mehr mit denen des aktuellen Sourcecodes übereinstimmen. Welche Situation zum Fehler geführt hat, ist dann nur noch schwierig zu ermitteln.

Tipps und Refactorings Zunächst führen wir eine Parameterprüfung ein, wie dies im Refactoring ÜBERPRÜFE EINGABEPARAMETER in Abschnitt 17.5.2 beschrieben wird. Ein nicht initialisiertes Attribut `startFolder` löst nun eine `IllegalStateException` anstelle einer `NullPointerException` aus. Dadurch wird nicht der offensichtliche Fehler (Wert ist `null`), sondern der zugrunde liegende semantische Feh-

ler (fehlerhafte Initialisierung des Objektzustands) deutlicher ausgedrückt. Noch kritischer ist das Auslösen einer `NullPointerException` für den Fall einer fehlenden Implementierung, hier der nicht realisierten, absteigenden Sortierung. Dafür ist eine `UnsupportedOperationException` viel besser geeignet. Befolgen wir diese Hinweise, so erhalten wir nach einigen Änderungen folgenden Sourcecode, der zunächst mögliche Fehlersituationen und benötigte Vorbedingungen prüft und gegebenenfalls bei Verstößen Exceptions auslöst. Aufgrund der einzeiligen `throw`-Anweisungen wird dabei auf eine Klammerung verzichtet. Sind alle Vorbedingungen erfüllt, kommt es zur Ausführung der Methodenaktion `fillFolderContent()`:

```
public void fill(final String path)
{
   // Parameter-Check
   Objects.requireNonNull("parameter 'path' must not be null!");

   // Initialization-Check
   if (startFolder == null)
       throw new IllegalStateException("attribute 'startFolder' must not be " +
                       "null! Call initialize() before any other method call!");

   // Implementation-Check
   if (!ascending)
       throw new UnsupportedOperationException("descending sort order " +
                                               "not implemented!");

   fillFolderContent(path);
}
```

Beachten Sie dabei die Unterschiede in den Hinweistexten der Exceptions: Diese enthalten nun zusätzliche Informationen zu Methodenparametern und Attributen, um mögliche Fehlerursachen klarer nachvollziehbar zu machen.

16.3.3 Bad Smell: Fangen der allgemeinsten Exception

Leider sieht man zu häufig das Fangen der unspezifischen Exception als eine – wenn auch oft unangebrachte, weil unspezifische – Form der Fehlerbehandlung:

```
catch (final Exception ex)
{
    // beliebiger Sourcecode ...
}
```

Warum ist das ein Bad Smell? Ein solcher `catch`-Block verfehlt nahezu immer das Ziel, angemessen auf die jeweilige Fehlersituation reagieren zu können. Es findet eine Vermischung der Behandlung erwarteter und beliebiger anderer, unerwarteter Exceptions statt. Durch ein solches unspezifisches Exception Handling können demnach Fehlersituationen verdeckt werden.

Tipps und Refactorings Eine mögliche Verbesserung stellt das Fangen eines spezifischeren Exception-Typs dar, um angemessen auf die Fehlersituation reagieren zu können.

Ausnahmen

Es gibt ganz spezielle Fälle, in denen ein `catch (Exception ex)` tatsächlich etwas mehr Klarheit in den Sourcecode bringen kann. Dies ist immer dann der Fall, wenn auf verschiedenste Exceptions exakt gleich reagiert wird. Wir betrachten dazu folgendes Beispiel, das für drei unterschiedliche Arten von Exceptions (`Remote`-, `Finder`- und `RemoveException`) jeweils eine identische Log-Ausgabe produziert:

```
public void removeFromHistory(final int taskId)
{
    try
    {
        taskService.remove(Integer.valueOf(taskId));
    }
    catch (final RemoteException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
    catch (final FinderException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
    catch (final RemoveException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
}
```

Diese Sourcecode-Duplikation ist zu vermeiden. Denken wir kurz nach und schreiben die Methode dann um. Dabei müssen wir beachten, unerwartete Exceptions nicht wie die drei explizit abzufangenden und erwarteten Exceptions zu behandeln. Diese Aufgabe können wir mit einigen `instanceof`-Prüfungen wie folgt lösen:

```
public void removeFromHistory(final int taskId)
{
    try
    {
        taskService.remove(Integer.valueOf(taskId));
    }
    catch (final Exception ex)
    {
        if (ex instanceof RemoteException ||
            ex instanceof FinderException ||
            ex instanceof RemoveException)
        {
            log.error("Unable to remove task with id = " + taskId, ex);
            return;
        }
        throw new RuntimeException(ex); // Ummantelung mit RuntimeException
    }
}
```

Hiermit vermeidet man Sourcecode-Duplikation und erhöht die Lesbarkeit sowie Übersichtlichkeit – das Exception Handling erfordert aber einen Trick: Die ursprüngliche Methode definiert keine Checked Exception in der Signatur. Um dazu kompatibel zu bleiben, muss eine auftretende Exception mit einer `RuntimeException` ummantelt werden, weil diese nicht in der Signatur angegeben werden muss (vgl. Abschnitt 5.6.2).

Vereinfachungen mit JDK 7

Mit dem Sprachfeature Multi Catch wird das Exception Handling erleichtert (vgl. Abschnitt 5.6.3), weil mehrere Exceptions mit einem `catch`-Block gefangen werden können. Das sehen wir in der Methode `removeFromHistory(int)`:

```
public void removeFromHistory(final int taskId)
{
    try
    {
        taskService.remove(Integer.valueOf(taskId));
    }
    // Multi Catch ab JDK 7
    catch (final RemoteException | FinderException | RemoveException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
}
```

16.3.4 Bad Smell: Exceptions zur Steuerung des Kontrollflusses

Dieser Abschnitt erklärt, warum Exceptions normalerweise nicht zur Steuerung des Kontrollflusses eingesetzt werden sollten.

Betrachten wir dazu folgende Methode `sum(int[])`, die die Summe eines übergebenen `int[]` berechnen soll. In einer Endlosschleife wird der aktuelle Wert an Position `i` auf die Variable `sum` addiert, solange keine Exception auftritt. Zum Abbruch der Schleife wird eine `ArrayIndexOutOfBoundsException` erwartet:

```
private int sum(final int[] values)
{
    int i = 0;
    int sum = 0;
    try
    {
        while (true)
        {
            sum += values[i];
            i++;
        }
    }
    catch (final Exception ex)
    {
        // Ende des Arrays erreicht, Summenberechnung stoppen
    }
    return sum;
}
```

Warum ist das ein Bad Smell? Der indizierte Zugriff löst irgendwann – je nach Größe des übergebenen Arrays – eine `ArrayIndexOutOfBoundsException` aus. Dadurch endet die Schleife. Diese ungewöhnliche Art des Schleifendurchlaufs ist einige Zeilen länger und durch das Exception Handling schlechter lesbar als eine »normale« `for`- oder `while`-Schleife. Auch die eigentliche Programmlogik lässt sich schlechter nachvollziehen. Darüber hinaus birgt diese »Exception«-Schleife das Problem, dass alle möglichen Arten von Exceptions im `catch`-Block gefangen werden. Dadurch werden möglicherweise Fehlersituationen »verschluckt«: Die (versehentliche) Übergabe einer `null`-Referenz für die Variable `values` löst dort eine `NullPointerException` aus. Dieser Fehler bleibt vor dem Aufrufer verborgen, weil er vermeintlich in `catch (Exception ex)` behandelt wird, allerdings erfolgt keine Summierung und es kommt zur Rückgabe von 0 als Ergebnis. Der auslösende Applikationsfehler, hier das nicht initialisierte `values`-Array, fällt so allerdings nicht direkt auf: Es ist lediglich ein Fehler im Algorithmus, jedoch kein Programmabsturz beobachtbar. Weitere Probleme solcher `catch`-Blöcke werden als BAD SMELL: FANGEN DER ALLGEMEINSTEN EXCEPTION in Abschnitt 16.3.3 sowie als BAD SMELL: UNBEHANDELTE EXCEPTION in Abschnitt 16.3.1 diskutiert.

Tipps und Refactorings Eine mögliche Verbesserung stellt das Fangen eines besser geeigneten Exception-Typs dar, für die `sum(int[])`-Methode etwa eine `ArrayIndexOutOfBoundsException`. Aber selbst das funktioniert nur für den Fall, dass die Methode `sum(int[])` keine weitere Methode aufruft, die eine solche Exception auslösen kann. Ansonsten würde die Summierung einfach an einer beliebigen Stelle abgebrochen und falsche Ergebnisse liefern.

Exceptions sollten nur zur Behandlung von außergewöhnlichen Fehlersituationen eingesetzt werden. Eine Steuerung des Kontrollflusses erfolgt vorzugsweise durch konventionelle Java-Sprachmittel: Die obige Variante des Schleifendurchlaufs ohne Prüfung auf Array-Grenzen war zwar zur Anfangszeit von Java in JVMs schneller als eine `for`- oder `while`-Schleife mit einer Prüfung der Array-Grenzen – allerdings natürlich bei Weitem schlechter lesbar. Die mit JDK 5 eingeführte for-each-Schleife erlaubt es, weiter zu abstrahieren, und hilft, Fehler bei der Prüfung auf Array-Grenzen und beim indizierten Zugriff zu vermeiden.

16.3.5 Bad Smell: Unbedachte Rückgabe von `null`

Dieser Bad Smell beschreibt, warum die unbedachte Rückgabe von `null` Probleme bereiten kann. Eher selten sollte man überhaupt mit dem Wert `null` als Rückgabe arbeiten. Dies gilt etwa für Suchfunktionen, um ausdrücken zu können, dass kein Treffer gefunden wurde. Seit Java 8 verwendet man dafür besser die Klasse `Optional<T>` (vgl. Abschnitt 6.5). Für »normale« Verarbeitungsroutinen ist die Rückgabe von `null` meistens nicht sinnvoll, weil Aufrufer mit diesem Wert entweder nicht rechnen oder aber übervorsichtig nahezu überall auf `null` prüfen, selbst wenn das gar nicht erforderlich ist.

Betrachten wir eine Umwandlung zweier Eingabewerte in der folgenden `main()`-Methode. Dort wird durch den Aufruf der Methode `toHashSeparatedString()` nach jedem Zeichen ein '`#`' im übergebenen String eingefügt:

```
public static void main(final String[] args)
{
    final String value1 = toHashSeparatedString("012");
    final String value2 = toHashSeparatedString("");

    System.out.println("Encoded='" + value1 + "' / length=" + value1.length());
    System.out.println("Encoded='" + value2 + "' / length=" + value2.length());
}

public static String toHashSeparatedString(final String message)
{
    if (message.length() == 0)
        return null;

    final StringBuffer sb = new StringBuffer("#");
    for (int i = 0; i < message.length(); i++)
    {
        sb.append(message.charAt(i)).append('#');
    }
    return sb.toString();
}
```

Listing 16.2 *Ausführbar als* '**DONTRETURNNULLEXAMPLE**'

Das Programm DONTRETURNNULLEXAMPLE produziert folgende Ausgaben:

```
Encoded='#0#1#2#' / length=7
Exception in thread "main" java.lang.NullPointerException
        at ch16_badsmells.DontReturnNullExample.main(DontReturnNullExample.java
              :18)
```

Warum ist das ein Bad Smell? Rückgaben von `null` müssen im aufrufenden Sourcecode speziell behandelt werden. Geschieht dies – wie in der obigen `main()`-Methode – nicht, so kommt es unerwartet zu einer `NullPointerException`: In diesem Beispiel also bei der Ausgabe der Länge des leeren, zweiten Eingabewerts.

Aus der Methodensignatur und aufgrund der fehlenden Dokumentation ist aber nicht ersichtlich, dass der Wert `null` überhaupt als Rückgabe möglich ist. Zudem wird hier mit dem Wert `null` ein erlaubter Randfall anders als gewöhnliche

Eingabewerte ausgewertet. Ein derartiges Problem beschreibt BAD SMELL: SONDERBEHANDLUNG VON RANDFÄLLEN in Abschnitt 16.3.7. Eine Nachricht der Länge 0 sollte auf keinen Fall im nachfolgenden Sourcecode eine `NullPointerException` auslösen. Diese Realisierung provoziert unnötigerweise Exceptions in aufrufenden Methoden.

Tipps und Refactorings Die Rückgabe eines leeren Strings für leere Eingabedaten vermeidet oftmals Probleme. Seit Java 8 ist es viel eleganter, die Klasse `Optional<T>` (vgl. Abschnitt 6.5) einzusetzen. Das sorgt für mehr Verständlichkeit und Lesbarkeit.

Ausbreitung dieses Bad Smells

Problematisch ist vor allem die Rückgabe von `null`-Werten für Referenzen auf Arrays, Listen oder sonstige Containerklassen, da dies häufig zu einer Ausbreitung dieses Bad Smells führt. Betrachten wir dazu folgende vereinfachte Methode `getPersonsByPrefix(String)`, die als Rückgabe eine `List<Person>` besitzt. In dieser Methode wird durch Aufruf der Hilfsmethode `findObjectsByPrefix(Class<?>, String)` der Utility-Klasse `DBAccess` ein `Object[]` ermittelt, das auf ein `Person[]` gecastet wird. Ist dieses `null`, wird hier auch der Wert `null` zurückgeliefert. Ansonsten erfolgt eine Umwandlung des Ergebnisses in ein `Person`-Array:

```
public static List<Person> getPersonsByPrefix(final String prefix)
{
   final Person[] result = (Person[]) DBAccess.findObjectsByPrefix(Person.class,
                                                                   prefix);

   if (result != null)
   {
      return Arrays.asList(result);
   }

   // Problematische Rückgabe von null
   return null;
}
```

Eine leere Treffermenge drückt man für Arrays und Containerklassen besser durch NULL-OBJEKTE aus (vgl. Abschnitt 18.3.2). Tut man dies nicht, so pflanzen sich `null`-Prüfungen immer weiter fort. In diesem Beispiel sieht man das an der Methode `findObjectsByPrefix(Class<?>, String)`. Würde diese ein leeres Array statt `null` zurückliefern, könnte man sich die Spezialbehandlung in der aufrufenden Methode `getPersonsByPrefix(String)` sparen. Dann könnte man als Aufrufer kurz und elegant Folgendes schreiben:

```
public List<Person> getPersonsByPrefix(final String prefix)
{
   final Person[] result = (Person[]) DBAccess.findObjectsByPrefix(Person.class,
                                                                   prefix);

   return Arrays.asList(result);
}
```

Mögliche Schwierigkeiten Wenn man externe Bibliotheken einsetzt, muss man selbst etwas tun, um zu verhindern, dass sich dieser Bad Smell ausbreitet. Das gilt insbesondere, wenn Methoden `null`-Werte für Collections zurückliefern. Da sich die Implementierungen von Methoden aus Bibliotheken nicht ändern lassen, kann man sich eine eigene Hilfsmethode schreiben, die die Bibliotheksmethode aufruft und passend erweitert. Dadurch wird die ansonsten notwendige Spezialbehandlung für `null`-Werte vor der eigenen Applikation versteckt und der nutzende Sourcecode vereinfacht. Selbst wenn wir in diesem Beispiel die Klasse `DBAccess` nicht im Zugriff hätten, so könnten wir doch etwas tun. Durch Befolgen es eben genannten Vorgehens lässt sich die Methode `getPersonsByPrefix()` folgendermaßen korrigieren, um eine Fortpflanzung von `null`-Abfragen zu verhindern:

```
public List<Person> getPersonsByPrefix(final String prefix)
{
   final Person[] result = (Person[]) DBAccess.findObjectsByPrefix(Person.class,
                                                                   prefix);

   if (result != null)
   {
      return Arrays.asList(result);
   }

   // Vermeide problematische Rückgabe von null
   return Collections.emptyList();
}
```

16.3.6 Bad Smell: Rückgabe von `null` statt Exception im Fehlerfall

Dieser Bad Smell schildert, warum schwerwiegende Fehlersituationen besser durch das Auslösen einer Exception als durch die Rückgabe von `null` (oder eines Fehlerwerts) ausgedrückt werden sollten. Bekanntermaßen können Rückgabewerte von Methoden durch Aufrufer einfach ignoriert werden. Exceptions besitzen den Vorteil, nicht allzu leicht unbehandelt zu bleiben: Eine Checked Exception muss entweder durch einen `catch`-Block behandelt oder mit `throw` weiter propagiert werden. Für Unchecked Exceptions ist beides optional auch möglich.

Betrachten wir folgende Methode `getServerAndPort()`, die aus einer Property-Datei über die Methode `getPropertyValue(String)` zwei Werte liest. Sind beide vorhanden, wird aus diesen ein Servername mit Portnummer zusammengebaut. Fehlt eine der Informationen, so wird der Wert `null` zurückgeliefert. Dies ist jedoch weder im Methodenkommentar beschrieben, noch stellt der Wert `null` einen sinnvollen oder geschweige denn gültigen Rückgabewert dar:

```
/**
  * @return der Name des Servers in der Form 'host:port'.
  */
public String getServerAndPort()
{
    final String host = getPropertyValue(KEY_HOST);
    final String port = getPropertyValue(KEY_PORT);

    if (host != null && port != null)
    {
        return host + ":" + port;
    }

    return null;
}
```

Warum ist das ein Bad Smell? Bei Referenzparametern ist die unbedachte Rückgabe des Werts `null` problematisch, wenn Aufrufer mit dem Rückgabewert ohne Prüfung weiterarbeiten. Dies beschreibt detailliert BAD SMELL: UNBEDACHTE RÜCKGABE VON NULL in Abschnitt 16.3.5.

Verlässt sich aufrufender Sourcecode auf eine gültige Rückgabe und führt keine Prüfung auf `null` durch, so kommt es zu einer unbehandelten `NullPointerException`. Im obigen Beispiel bleibt bei korrekter Konfiguration der beschriebene Softwaredefekt unentdeckt – vielleicht sogar während der gesamten Entwicklungs- und Testphase. Wird die Konfiguration jedoch im Nachhinein (versehentlich) dahingehend verändert, dass einer der beiden Werte fehlt, löst dies unerwartet eine `NullPointerException` aus. Der Zusammenhang mit einem geänderten oder fehlenden Property-Wert ist jedoch nicht ersichtlich.

Tipps und Refactorings Das Auslösen einer Exception ermöglicht die Ausgabe eines Stacktrace, wodurch sich Fehler in der Regel gut nachvollziehen lassen. Mithilfe einer `IllegalStateException` wird ein möglicher Konfigurationsfehler frühzeitig erkennbar und semantisch besser beschrieben. Eventuell kann ein Problem dann umgehend durch Einfügen eines fehlenden Werts in eine Property-Datei behoben werden. Dazu ist es hilfreich, im Text der Exception zusätzlich den Pfad und den Namen der verwendeten Property-Datei anzugeben. Eine Realisierung könnte wie folgt aussehen:

```
public String getServerAndPort()
{
    final String host = getPropertyValue(KEY_HOST);
    final String port = getPropertyValue(KEY_PORT);

    if (host != null && port != null)
    {
        return host + ":" + port;
    }

    throw new IllegalStateException("missing property entries for server!" +
                    " no parameters '" + KEY_HOST + "' and/or '" + KEY_PORT +
                    "' in file '" + getPropertyFilePath() + "'.");
}
```

An diesem Beispiel wird deutlich, dass eine offensive Fehlerbehandlung eine spätere Wartung ungemein erleichtern kann. Exceptions stören zwar den Programmablauf, zur Fehlersuche sind sie meistens aber hilfreich. Wird ein Fehler durch `null`-Prüfungen oder leere `catch`-Blöcke verschluckt, so macht er sich möglicherweise lediglich durch ein merkwürdiges Programmverhalten bemerkbar. Er lässt sich jedoch nur mit Mühe zurückverfolgen und durch intensive Analysen finden und beheben.

16.3.7 Bad Smell: Sonderbehandlung von Randfällen

Zum Teil findet man Sonderbehandlungen für erlaubte, aber seltene oder ungewöhnliche Eingabewerte. Gleiches gilt für Rückgabewerte. Gerade deren geeignete Wahl kann das Erkennen von Fehlersituationen erleichtern.

Betrachten wir dies beispielhaft an der folgenden Methode `toHexString()`, die eine Stringrepräsentation in hexadezimaler Form aus einem übergebenen `byte`-Array `message` erzeugt. Sowohl bei der Übergabe von `null` als auch für eine leere Eingabe wird `null` zurückgegeben:

```
public static String toHexString(final byte[] message)
{
    if (message == null)
        return null;
    if (message.length == 0)
        return null;

    final StringBuffer sb = new StringBuffer("|");
    for (int i = 0; i < message.length; i++)
    {
        // Wandle byte in einen Hex-String
        final String hexStr = "0" + Integer.toHexString((int) message[i]);
        sb.append(hexStr.substring(hexStr.length() - 2)).append('|');
    }
    return sb.toString();
}
```

Warum ist das ein Bad Smell? In diesem Beispiel stellt eine Eingabe von `null` eigentlich einen Fehler dar, was anhand des Rückgabewerts nicht deutlich wird. Die Eingabe eines leeren Arrays ist dagegen ein seltener, aber erlaubter Randfall.

Die Gleichbehandlung von Fehlern und Randfällen durch Rückgabe gleicher Werte führt zu Inkonsistenzen. Man stellt eine normale Situation – hier die Eingabe eines Arrays der Länge 0 – wie eine Fehlersituation dar. Betrachten wir die verschiedenen Fälle für den Eingabewert `message` genauer, um zu verstehen, warum Sonderbehandlungen von Randfällen und daraus resultierende, inkonsistente Rückgabewerte einen Bad Smell darstellen:

1. Wird ein Array der Länge 0 übergeben, so kann dafür eine sinnvolle Stringrepräsentation erzeugt werden, etwa `"[]"`. Dieser Randfall sollte demnach genauso bearbeitet werden wie ein Array mit Eingabedaten. Von diesem Vorgehen sollte nur abgewichen werden, wenn dies fachlich notwendig ist und zudem explizit in der Methodendokumentation erwähnt wird. In diesem Beispiel führt ein leeres Array aber zur Rückgabe von `null` und ist damit anhand des Rückgabewerts nicht mehr von einem Fehler zu unterscheiden.
2. Im Fall einer ungültigen Eingabe (`null`) kann die Methode keine geeignete Stringrepräsentation erzeugen. Die Rückgabe von `null` drückt dies nur unzureichend aus. Wenn dies explizit dokumentiert ist, stellt eine derartige Rückgabe nicht unbedingt einen Fehler dar. Es besteht aber immer die Gefahr, dass es zu einer Verschleierung eines Fehlers kommt.

Tipps und Refactorings Die Spezialbehandlung für das leere Eingabe-Array ist überflüssig und kann komplett entfallen. Auf die Eingabe von `null` sollte mit einer `IllegalArgumentException` reagiert werden, wie dies bereits in Abschnitt 16.3.6 als BAD SMELL: RÜCKGABE VON NULL STATT EXCEPTION IM FEHLERFALL besprochen wurde. Der Aspekt der Parameterprüfung zur Sicherstellung eines gültigen Objektzustands ist in Abschnitt 16.3.8 Thema von BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN. Weshalb die Rückgabe von `null` problematisch ist und man die dadurch notwendigen `null`-Prüfungen nicht auf Klienten übertragen sollte, wurde bereits als BAD SMELL: UNBEDACHTE RÜCKGABE VON NULL in Abschnitt 16.3.5 vorgestellt.

16.3.8 Bad Smell: Keine Gültigkeitsprüfung von Eingabeparametern

Eine fehlende Prüfung von Eingabeparametern (insbesondere in öffentlichen Schnittstellen oder an Komponentengrenzen) kann schnell zu inkonsistenten Objektzuständen führen. Meistens lassen sich Inkonsistenzen durch ungültige Parameterwerte bereits beim Aufruf von Business- und öffentlichen `set()`-Methoden oder idealerweise direkt schon im Konstruktor feststellen.

Betrachten wir exemplarisch den folgenden Konstruktor der Klasse `FramedDisplayMsg`. Die Parameter `receiverNo` und `message` werden ungeprüft an korrespondierende Attribute zugewiesen:

```
private final int    receiverNo;
private final byte[] message;

public FramedDisplayMsg(final int receiverNo, final byte[] message)
{
    this.receiverNo = receiverNo;
    this.message = message;
}
```

Weiterhin ist unter anderem folgende Zugriffsmethode definiert:

```
public byte[] getMessageWithoutFraming() throws IOException
{
    if (MsgSupport.checkMessage(this.message) != MsgSupport.CHECK_MSG_OK)
        throw new IllegalArgumentException("parameter 'message' has an " +
                                " invalid format " + Arrays.toString(message));

    return ByteArrayUtils.extractByteArray(message, FRAME_OFFSET,
                                           message.length - FRAME_OFFSET);
}
```

Warum ist das ein Bad Smell? Bei fehlerhaften Eingabewerten verbleibt ein neu erzeugtes `FramedDisplayMsg`-Objekt in einem nicht korrekt initialisierten Zustand, ohne dies zu kommunizieren. Auf ungültige Eingaben sollte allerdings so früh wie möglich reagiert werden, um inkonsistente Objektzustände zu vermeiden. Dies geschieht hier nicht und kann diverse unerklärliche Folgefehler auslösen.

Erst bei einem Aufruf der `getMessageWithoutFraming()`-Methode wird auf Inkonsistenzen geprüft. Werden diese erkannt, wird eine Exception ausgelöst, um einen Fehler zu signalisieren.

Besonders problematisch ist jedoch, dass die Konsistenzprüfung zu spät erfolgt. Zwischen der Konstruktion und einem Aufruf der obigen Methode mit der Konsistenzprüfung könnten Klienten bereits mit dem Objekt arbeiten. In diesem Beispiel sind `null`-Werte für die Attribute nicht erlaubt, aber durch die fehlende Parameterprüfung nicht ausgeschlossen. Das Auslesen eines Attributs kann unerwartet den Wert `null` liefern und damit `NullPointerException`s bei Aufrufern auslösen.

Das Schlimme in beiden Fällen ist, dass es sich hierbei um eine Fehlerverschleierung handelt, für die wir in meiner damaligen Arbeitsgruppe den Spitznamen »Delayed Exception« verwendet haben. Er besagt, dass Probleme erst beim Zugriff und nicht bereits beim Initialisieren auftreten. ***Das Ungerechte daran ist, dass nicht der Verursacher der Inkonsistenzen mit einer Exception bestraft wird, sondern irgendein späterer Nutzer der Klasse. Ist dies z. B. das GUI, so wird der Fehler schnell einem falschen Verursacher zugeordnet.***

Tipps und Refactorings Die Prüfung aller Parameter öffentlicher Methoden und von Konstruktoren hilft, Initialisierungsprobleme so früh wie möglich aufzuspüren. Dadurch kann man falsch initialisierte Objekte (invalide Objektzustände) vermeiden. Ein entsprechendes Vorgehen ist in Abschnitt 17.5.2 als Refactoring ÜBERPRÜFE EINGABEPARAMETER detailliert beschrieben. Dort wird auch diskutiert, ob man alle Parameter oder nur eine Auswahl überprüfen sollte.

In diesem Fall kopieren wir die Konsistenzprüfung aus der `getMessageWithout-Framing()`-Methode und entfernen diese dort:

```
public byte[] getMessageWithoutFraming() throws IOException
{
    return ByteArrayUtils.extractByteArray(message, FRAME_OFFSET,
                                           message.length - FRAME_OFFSET);
}
```

Anschließend fügen wir diese Konsistenzprüfung in den Konstruktor ein und ergänzen noch eine Prüfung des Parameters `message` auf `null`.

```
public CorrectedFramedDisplayMsg(final int receiverNo, final byte[] message)
{
    if (MsgSupport.checkMessage(message) != MsgSupport.CHECK_MSG_OK)
        throw new IllegalArgumentException("parameter 'message' has an " +
                              "invalid format" + Arrays.toString(message));

    // Parameterprüfung mit Utility-Klasse Objects
    this.message = Objects.requireNonNull(message, "parameter 'message' must " +
                                                  "not be null");
    this.receiverNo = receiverNo;
}
```

16.3.9 Bad Smell: Fehlerhafte Fehlerbehandlung

Zum Teil findet man Fehlerbehandlungsroutinen, die selbst wiederum Fehler enthalten oder die Fehlerbehandlung nur unvollständig oder nicht korrekt durchführen.

Das folgende Beispiel der Methode `synchronizeTime(long)` zeigt eine Abfrage des Attributs `currentSender` auf den Wert `null`. Im Falle eines `null`-Werts kommt es zu einer Fehlermeldung in Form einer Log-Ausgabe:

```
public void synchronizeTime(final long newTime)
{
    if (currentSender == null)
    {
        log.error("Called synchronizeTime for a null current sender.");
    }
    currentSender.synchronizeTime(new Date(newTime));
}
```

Warum ist das ein Bad Smell? Eine Fehlerbehandlung, die selbst Fehler auslöst, die vom eigentlichen Fehler ablenken, ist nicht wirklich sinnvoll. Im Beispiel hilft die Ausgabe des fehlerhaften Zustands nur bedingt weiter, da mit der nächsten Anweisung `currentSender.synchronizeTime()` eine `NullPointerException` beim Dereferenzieren der `null`-Referenz ausgelöst wird. Wie man leicht sieht, fehlt in diesem Fall ein Methodenabbruch. Der fehlerhafte Objektzustand (die unvollständige Initialisierung) sollte per `IllegalStateException` an den Aufrufer kommuniziert werden. Die ansonsten auftretende `NullPointerException` besitzt nicht die gleiche Aussagekraft, weil diese den ungültigen Objektzustand semantisch nicht so klar ausdrückt.

Außerdem sollte das Attribut `currentSender` besser `private` sein. Dadurch wäre das Attribut als Implementierungsdetail nach außen nicht sichtbar. Ein Aufrufer würde sich dann im Fehlerfall allerdings fragen: Was ist "`currentSender`" im Stacktrace der Exception?

Tipps und Refactorings Eine Zustandsprüfung lässt sich oftmals sinnvollerweise in eine eigene Methode auslagern. Das beschreibt in Abschnitt 17.5.1 das Refactoring FÜHRE EINE ZUSTANDSPRÜFUNG EIN. Wenden wir dieses an, so entsteht die folgende Methode `checkInitialization()`:

```
private void checkInitialization()
{
    if (currentSender == null)
    {
        log.error("Illegal state: attribute 'currentSender' is uninitialized");
        throw new IllegalStateException("Illegal state: attribute " +
                                        "'currentSender' is uninitialized");
    }
}
```

Durch den Einsatz einer solchen Prüfmethode wird der Sourcecode besser lesbar. Bei Bedarf kann die Abfrage auch in anderen Methoden eingesetzt werden, ohne die Prüfungen durch Copy-Paste im Sourcecode zu duplizieren.

```
public void synchronizeTime(final long newTime)
{
    checkInitialization();

    currentSender.synchronizeTime(new Date(newTime));
}
```

Spezialfälle

In diesem einfachen Beispiel war der Fehler noch recht schnell zu finden, im folgenden Beispiel der Methode `send(Object)` und dem Einsatz von `instanceof` und `getClass()` ist er jedoch besser verborgen:

```
public void send(final Object telegram)
{
    if (!(telegram instanceof RadioTelegram))
    {
        log.error("Invalid telegram class " + telegram.}getClass().getName());
        return;
    }
    // ...
}
```

Der Aufruf von `instanceof` ist zwar laut JLS sicher bezüglich `null`. Das bedeutet: Ein Test `null instanceof XYZ` liefert immer `false`. Vorsicht ist jedoch geboten, wenn man die Aussage negiert. Denn eine Negation besagt nicht, dass die Variable ungleich `null` ist! Wird an die obige Methode eine `null`-Referenz als Parameter übergeben, so löst der Aufruf von `getClass()` eine `NullPointerException` aus.

Tipp: Test der Fehlerbehandlung

Achten Sie darauf, auch die Fehlerbehandlung zu testen. Dazu sollten bewusst Testfälle definiert werden, in denen beispielsweise Referenzvariablen mit `null` initialisiert sind. Auch andere invalide Einträge sollten getestet werden, etwa ungültige Wertebereiche: Fehler finden sich häufig nicht im normalen Wertebereich, sondern bei Extremwerten. Es sollte dazu ein ***Extremwerttest*** durchgeführt werden. Der normale Programmablauf wird durch ***Äquivalenzklassentests*** abgesichert. Auf Details zum Testen gehe ich in Kapitel 20 ein.

16.3.10 Bad Smell: I/O ohne `finally` oder ARM

Bei der Arbeit mit Ressourcenobjekten werden zusätzlich zu den Java-Objekten auch Betriebssystemressourcen angefordert. Wird ein entsprechendes Java-Objekt zerstört, so werden nicht in jedem Fall als Folge davon auch die belegten Systemressourcen freigegeben. Dazu ist eine spezielle `close()`-Methode vom Entwickler aufzurufen.

Betrachten wir hier eine typische Methode `readProperties(String)`, die Properties aus einer Datei lesen soll. Im Regelfall erfolgt nach einem Zugriff auf die Datei ein Aufruf von `close()`. Im `catch`-Block wurde dieser Aufruf vergessen:

```
private boolean readProperties(final String fileName)
{
    try
    {
        final FileInputStream inputStream = new FileInputStream(fileName);
        loadProperties(inputStream);
        inputStream.close();
    }
    catch (final IOException ex)
    {
        log.warn("can't read file '" + fileName + "'", ex);
        return false;
    }
    return true;
}
```

Warum ist das ein Bad Smell? Problematisch an dieser Methode ist das Verhalten im Falle eines Fehlers beim Zugriff auf die Property-Datei. Tritt dabei eine `IOException` auf, so werden Ressourcen nicht garantiert wieder freigegeben, da kein direkter Aufruf von `close()` erfolgt.[3]

Tipps und Refactorings Wie bereits in Abschnitt 5.6 diskutiert, sollte man Aufräumarbeiten bevorzugt in einem `finally`-Block durchführen. Tritt dann während der Ausführung im `try`- oder `catch`-Block eine `IOException` auf, so kann das zuvor geöffnete `InputStream`-Objekt in jedem Fall umgehend geschlossen und somit auch die allozierten Betriebssystemressourcen freigegeben werden. Wir korrigieren diese Schwachstelle in unserer Implementierung. Dadurch entsteht folgende verbesserte Methode `readProperties(String)`, die das seit JDK 7 vorhandene Sprachfeature ARM (Automatic Resource Management) (vgl.Abschnitt 5.6.3) zur Freigabe von Ressourcen nutzt:

```
private boolean readProperties(final String fileName)
{
    try (final InputStream inputStream = new FileInputStream(fileName))
    {
        loadProperties(inputStream);
    }
    catch (final IOException ex)
    {
        log.warn("can't read file '" + fileName + "'", ex);
        return false;
    }
    return true;
}
```

16.3.11 Bad Smell: Resource Leaks durch Exceptions im Konstruktor

Zum Teil erfolgen Zugriffe auf Streams oder andere Ressourcen bereits im Konstruktor. Das ist oftmals eher ungünstig, weil dabei schon Betriebssystemressourcen alloziert werden. Zudem können bei I/O-Aktionen immer auch Exceptions ausgelöst werden. Werden diese im Konstruktor behandelt, so ist selten ein sinnvoller Objektzustand zu erreichen.

Der folgende Sourcecode löst `IOExceptions` (genauer die Spezialisierung `FileNotFoundException`) im Konstruktor aus, wenn eine übergebene Datei zum Einlesen nicht gefunden oder aber keine Datei zum Schreiben erzeugt werden kann. In beiden Fällen werden möglicherweise bereits Dateiressourcen belegt, aber eventuell nicht (sofort) freigegeben.

[3] Als letztes Sicherheitsnetz besitzen einige I/O-Klassen, im Speziellen die Klassen `FileInputStream` und `FileOutputStream`, eine überschriebene `finalize()`-Methode, die einen Aufruf an `close()` ausführt.

```
public class ZipConverter
{
    private final InputStream  inStream;
    private final OutputStream outStream;

    public ZipConverter(final String inputFileName, final String outputFileName)
           throws FileNotFoundException
    {
        // beide Konstruktoren können eine FileNotFoundException auslösen
        inStream = new FileInputStream(inputFileName);
        outStream = new FileOutputStream(outputFileName);
    }

    protected final InputStream getInputStream()
    {
        return inStream;
    }

    protected final OutputStream getOutputStream()
    {
        return outStream;
    }

    // ...
```

Warum ist das ein Bad Smell? Durch I/O ausgelöste Exceptions im Konstruktor führen dazu, dass Objekte sich nicht vollständig konstruieren lassen und damit der Programmablauf gestört wird. Eine sinnvolle Fehlerbehandlung fällt schwer. Bereits belegte Ressourcen können dann zum Teil nicht korrekt freigegeben werden, weil nach dem Abbruch der Objektkonstruktion durch Auslösen einer Exception kein Zugriff auf die Ressourcenattribute möglich ist.

Tipps und Refactorings Im Folgenden betrachten wir Möglichkeiten, die genannten Probleme zu lösen, und beginnen mit der Idee, im Konstruktor die erwarteten Exceptions abzufangen. Die Konstruktion der Streams wird dazu jeweils mit einem eigenen `try-catch`-Block wie folgt umschlossen:

```
private static class ZipConverter
{
    // Indikator, ob bereits initialisiert
    private InputStream  inStream = null;
    private OutputStream outStream = null;

    public ZipConverter(final String inputFileName, final String outputFileName)
    {
        try
        {
            inStream = new FileInputStream(inputFileName);
        }
        catch (final FileNotFoundException fnfe)
        {
            // Achtung: „silent fail", d. h., Exception wird nicht propagiert
            IOUtils.closeQuietly(inStream);
        }
```

```
        try
        {
            outStream = new FileOutputStream(outputFileName);
        }
        catch (final FileNotFoundException fnfe)
        {
            // Achtung: „silent fail", d. h., Exception wird nicht propagiert
            IOUtils.closeQuietly(outStream);
        }
    }

    // ...
```

Dabei nutzen wir Funktionalität aus der Java-Bibliothek Apache Commons IO, um die Fehlerbehandlung lesbar zu gestalten.

Der Einsatz einer solchen Fehlerbehandlung im Konstruktor führt beim Auftreten von I/O-Fehlern allerdings nur dazu, dass die eigentliche Objektkonstruktion vermeintlich erfolgreich verläuft. Das neu erzeugte Objekt ist jedoch anschließend in einem unbrauchbaren Zustand. Im Extremfall sind beide Streams geschlossen und jede Aktion darauf löst `IOException`s aus. Man könnte solche Initialisierungsprobleme dadurch dokumentieren, dass die korrespondierenden Stream-Attribute auf `null` gesetzt werden. Allerdings erfordert dies wiederum Fehler- und Spezialbehandlungen im aufrufenden Sourcecode: Dort muss nun vor jedem Zugriff zunächst eine `null`-Prüfung der Stream-Attribute erfolgen.

Demnach verlagert man das Problem lediglich: Nun sind die Aufrufer und nicht das Objekt selbst verantwortlich, Aufräumarbeiten durchzuführen. Dazu benötigen andere Klassen allerdings Zugriff auf die privaten, internen Stream-Attribute, d. h., die beiden `get()`-Methoden müssen als `public` definiert werden. Außerdem müssen Klienten nun wissen, dass ein `null`-Wert auf ein nicht erzeugtes Stream-Objekt hindeutet. Eine derartige Realisierung ist wenig objektorientiert. Es ist problematisch, diverse Annahmen nach außen zu publizieren und andere Objekte die eigenen Aufräumarbeiten ausführen zu lassen. Wie bekommen wir das in den Griff?

Zur Lösung verwenden wir das Entwurfsmuster ERZEUGUNGSMETHODE (vgl. Abschnitt 18.1.1). Um eine korrekte Konstruktion sowie eine saubere Fehlerbehandlung zu ermöglichen, müssen wir die Aktionen aus dem Konstruktor in eine `init()`-Methode verlagern. Nur für den Fall, dass sich ein Objekt erzeugen lässt, ohne dass es dabei zu einer `IOException` kommt, geben wir eine gültige Objektreferenz nach außen. Ansonsten schließen wir die Streams und propagieren die Exception. Folgendes Listing zeigt die entsprechende Erzeugungsmethode `createConverter()` sowie die anderen notwendigen Anpassungen in der Klasse:

```
private static class ZipConverter
{
    // Indikator, ob bereits initialisiert
    private InputStream  inStream = null;
    private OutputStream outStream = null;

    public static final ZipConverter createConverter(final String inputFileName,
           final String outputFileName) throws FileNotFoundException
    {
        // Definition, damit wir Streams schließen können
        ZipConverter zipConverter = null;
        try
        {
            zipConverter = new ZipConverter();
            zipConverter.init(inputFileName, outputFileName);
            return zipConverter;
        }
        catch (final FileNotFoundException ex)
        {
            // Die Variable zipConverter ist hier ungleich null
            IOUtils.closeQuietly(zipConverter.getInputStream());
            IOUtils.closeQuietly(zipConverter.getOutputStream());
            throw ex;
        }
    }

    private ZipConverter()
    {
    }

    private void init(final String inputFileName, final String outputFileName)
          throws FileNotFoundException
    {
        inStream = new FileInputStream(inputFileName);
        outStream = new FileOutputStream(outputFileName);
    }
    // ...
```

Es stellt sich zuletzt noch die Frage, was eigentlich passiert, wenn das `ZipConverter`-Objekt nicht mehr referenziert wird? Wer gibt die Systemressourcen dann wieder frei? Als erste Idee kommt uns die `finalize()`-Methode in den Sinn. Die Methode wird vor der Zerstörung eines Objekts durch den Garbage Collector aufgerufen. ***Die Ausführung der `finalize()`-Methode wird jedoch nicht garantiert. Ein sicherer Aufräumvorgang ist auf diese Weise demnach nicht möglich***.

Es bietet sich daher folgendes Vorgehen an: Man führt eine Methode `release()` ein, die benötigte Aufräumarbeiten erledigt, hier die geöffneten Streams schließt. Diese Methode sollte dann im Fehlerfall aus der Erzeugungsmethode sowie zusätzlich aus der `finalize()`-Methode aufgerufen werden. Zudem können Klienten diese Methode nach Abschluss ihrer Arbeiten aufrufen. Dadurch werden Aufräumarbeiten im Normalfall vom Programm selbst vorgenommen. Nur in unerwarteten Situationen steht als letztes Sicherheitsnetz eine Freigabe über `finalize()` bereit. Diese Variante wird auch von Joshua Bloch in »Effective Java« vorgeschlagen.

Jetzt könnte man sich natürlich fragen, worin der Unterschied dieses Vorgehens zu dem zuvor kritisierten Zugriff über `get()`-Methoden auf die Streams liegt. Die Antwort

ist bereits in der Frage enthalten: Bei der ursprünglichen Lösung wird ein Zugriff auf die Streams und die Kenntnis der Implementierungsdetails des Objekts benötigt. Eine `release()`-Methode ist dagegen objektorientiert und versteckt diese Details. Folgendes Listing zeigt eine mögliche Umsetzung:

```
private static class ZipConverter
{
    // Indikator, ob bereits initialisiert
    private InputStream  inStream = null;
    private OutputStream outStream = null;

    public static final ZipConverter createConverter(final String inputFileName,
           final String outputFileName) throws FileNotFoundException
    {
        // Definition, damit wir release() aufrufen können
        ZipConverter zipConverter = null;
        try
        {
            zipConverter = new ZipConverter();
            zipConverter.init(inputFileName, outputFileName);
            return zipConverter;
        }
        catch (final FileNotFoundException ex)
        {
            // Die Variable zipConverter ist hier ungleich null
            zipConverter.release();
            throw ex;
        }
    }

    protected void finalize() throws Throwable
    {
        release();
        super.finalize();
    }

    public void release()
    {
        IOUtils.closeQuietly(getInputStream());
        IOUtils.closeQuietly(getOutputStream());
    }
    // ...
```

Fazit

Für Klassen, die Systemressourcen als Attribute halten und bei deren Konstruktionsprozess Exceptions ausgelöst werden können, bietet sich eine Erzeugungsmethode zur sicheren Objektkonstruktion an. Weiterhin ist es empfehlenswert, eine `release()`-Methode zur kontrollierten Freigabe von Systemressourcen bereitzustellen.

Um im Fehlerfall verbleibende Aufräumarbeiten durchzuführen und Resource Leaks zu verhindern, kann als allerletzte Instanz eine `finalize()`-Methode dienen. Wichtig ist hierbei, auch die Implementierung der Basisklasse aufzurufen, damit die Aufräumarbeiten für alle »Objekt-Teile«, d. h. alle Klassenbestandteile aller Basisklassen, erfolgen können. Das Ganze ist jedoch recht unsicher, weil nicht in jedem Fall der Zeitpunkt und die Ausführung von `finalize()` bekannt bzw. garantiert ist.

16.4 Häufige Fallstricke

Dieser Abschnitt beschreibt einige weitere Programmierprobleme, die nicht die Schwere eines Bad Smells aus dem zuvor vorgestellten Katalog haben, die man aber trotzdem kennen sollte, um sie vermeiden zu können.

Fallstrick: Nutzung statischer Attribute statt Membervariablen

Manchmal werden statische Attribute zur Speicherung von Daten oder Eigenschaften eines bestimmten Objekts verwendet. Allerdings sind statische Attribute keinem Objekt konkret zugeordnet und sollten daher nur in Ausnahmefällen von Objektmethoden verändert werden, z. B. zur Referenzzählung im Konstruktor.

Fallstrick: Änderung statischer Attribute im Konstruktor / in Methoden

Werden statische Attribute im Konstruktor oder durch Objektmethoden verändert, so liegt meistens ein Problem vor. Wie bereits angedeutet, gibt es Situationen und Anwendungsfälle, in denen dies Sinn macht. Häufig geschieht die Veränderung aber lediglich aus Unachtsamkeit. Diese Wertänderung ist für alle Objekte dieser Klasse sichtbar.

Fallstrick: Missachtung der Initialisierungsreihenfolge statischer Attribute

Es ist wichtig, die Initialisierungsreihenfolge statischer Attribute zu kennen und zu beachten. Statische Attribute werden in der Reihenfolge ihrer Deklaration – d. h. entsprechend ihrem Auftreten im Sourcecode – initialisiert. Daran sollte man denken, um Probleme durch eine falsche Ausführungsreihenfolge zu verhindern. Das gilt insbesondere beim Aufruf statischer Methoden, die auf diese statischen Attribute zugreifen. Wir betrachten dies anhand eines Beispiels. Hier wird die `Logger`-Referenz `log` fälschlicherweise nicht als erstes statisches Attribut definiert:

```
public final class LogInitExceptionExample
{
  private static final long test_fail = init();
  private static final Logger log = Logger.getLogger("LogInitExceptionExample");

  private static long init()
  {
    log.info("init()");
    return 4712;
  }

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

Listing 16.3 *Ausführbar als* '**LOGINITEXCEPTIONEXAMPLE**'

Führen wir das Programm LOGINITEXCEPTIONEXAMPLE aus, so kommt es zu einer zunächst unerklärlichen `NullPointerException`:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.NullPointerException
        at ch16_badsmells.LogInitExceptionExample.init(LogInitExceptionExample.
              java:20)
        at ch16_badsmells.LogInitExceptionExample.<clinit>(
              LogInitExceptionExample.java:15)
```

Der Grund ist ein Reihenfolgeproblem bei der Initialisierung: Das statische Attribut `log` wurde noch nicht zugewiesen und besitzt somit den Defaultwert `null`, wenn in der Methode `init()` darauf zugegriffen wird. Für statische Attribute mit primitiven Typen kommt es nicht zu Exceptions, sondern stattdessen sind einige Werte je nach Typ einfach unerwartet mit dem Wert 0 bzw. `false` initialisiert.

Die Missachtung der Initialisierungsreihenfolge statischer Attribute kann somit zu merkwürdigen und schwierig auffindbaren Fehlern führen.

Fallstrick: Statische Methoden über Objektreferenzen aufrufen

An den beiden statischen Methoden `yield()` und `sleep(long)` der Klasse `Thread` kann man gut verdeutlichen, warum es irreführend ist, wenn man statische Methoden so ausführt, als ob sie Objektmethoden wären. Nehmen wir an, der `main`-Thread wäre aktiv und ein Thread `threadX` lauffähig, und es werden folgende Aufrufe ausgeführt:

```
threadX.sleep(2000);
threadX.yield();
```

Auf den ersten Blick meint man, der Thread `threadX` würde zunächst zwei Sekunden pausieren und danach kurz die Kontrolle abgeben. Tatsächlich wirken sich beide Methoden jedoch auf den zum Zeitpunkt des Methodenaufrufs aktiven Thread aus. Dies ist zunächst der laufende `main`-Thread. Auf welchen der beiden Threads (oder einen ganz anderen) sich der Aufruf von `yield()` auswirkt, ist zufällig und nicht vorhersagbar.

Fallstrick: Einsatz von Vererbung und statischen Methoden

Wird Vererbung bei Utility-Klassen eingesetzt, die nur statische Methoden bereitstellen, so ist kein Polymorphismus möglich, sondern es erfolgt nur ein Überdecken von Methoden. Dies deutet Abbildung 16-7 für eine `print()`-Methode an, die sowohl in der Basisklasse `Base` als auch in der Subklasse `Sub` definiert ist. Bei einem Aufruf dieser Methode erfolgt *kein* dynamisches Binden, sondern es wird die Methode basierend auf dem Kompiliertyp ausgewählt. Insbesondere ist das problematisch, wenn die Utility-Klassen Konstruktoren anbieten. Folgende Zeilen verdeutlichen das und zeigen Probleme durch den vorherigen Fallstrick nochmals auf.

```
final Base base = new Base();     // => Base::print()
final Base baseSub = new Sub();   // => Base::print()
final Sub sub = new Sub();        // => Sub::print()
```

Abbildung 16-7 *Vererbung, aber nur statische Methoden*

Fallstrick: Utility-Klasse mit öffentlichem Konstruktor

Eine Utility-Klasse mit öffentlichem Konstruktor suggeriert, dass es sich dabei um eine normale Klasse handeln würde, von der Instanzen erzeugt werden können. Dies sollte bei einer Utility-Klasse jedoch nicht der Fall sein. Stattdessen sollten hier nur einige (statische) Hilfsmethoden bereitgestellt werden. Der Konstruktor einer Utility-Klasse sollte daher immer als `private` definiert werden. Weiterhin sollte Ableitung vermieden werden, indem Utility-Klassen `final` definiert werden.

```
public final class MyUtilityClass
{
    private MyUtilityClass()
    {
        // Verhindert Konstruktoraufrufe durch andere Klassen
    }

    public static int someUtilityMethod(final int value1, final int value2)
    {
        // ...
    }
}
```

Fallstrick: Chaotische Konstruktor-/Methoden-Aufruffolgen

Werden Methoden oder Konstruktoren mit unterschiedlicher Anzahl an Parametern definiert und rufen sich diese gegenseitig auf, so führt dies schnell in ein Chaos. Dies gilt insbesondere dann, wenn die Methoden bzw. Konstruktoren untereinander nicht hierarchisch geordnet sind und die Aufrufe unstrukturiert erfolgen. Die linke Grafik in Abbildung 16-8 verdeutlicht dies. Jede Methode ist hier als Klammerpaar dargestellt. Die Parameter sind durch Punkte symbolisiert. In der Praxis ist eine derartige chaotische Aufrufhierarchie (vor allem bei Konstruktoren) leider nicht ungewöhnlich. Dadurch wird jedoch eine korrekte Initialisierung ungewiss.

Die rechte Seite der Grafik zeigt eine streng hierarchische Aufrufreihenfolge. Der Verlauf ist leicht nachzuvollziehen, und die Wahrscheinlichkeit für Fehler beim Aufruf von Methoden mit mehreren Parametern sinkt.

Abbildung 16-8 *Methoden- bzw. Konstruktordelegation*

Fallstrick: Missverständliches API durch Überladen

Wenn man nicht genau hinschaut, sieht es im folgenden Beispiel zunächst so aus, als ob ein Überschreiben der Methode `wait()` der Klasse `Object` stattfindet. Tatsächlich ist dies nicht möglich, da die Methode dort `final` definiert ist. Es erfolgt hier vielmehr ein Überladen:

```
public void wait(final int millis)
{
    try
    {
        Thread.sleep(millis);
    }
    catch (final InterruptedException ex)
    {
        Thread.currentThread().interrupt();
    }
}
```

Eine solche Definition suggeriert, dass eine ähnliche oder erweiterte Funktionalität wie bei der originalen Methode geboten werden soll. Dies ist hier jedoch nur begrenzt der Fall! Im Gegensatz zur Methode `wait()` der Klasse `Object`, die zur Steuerung von Threads und im Speziellen zur Verarbeitung von Benachrichtigungen über `notify()` bzw. `notifyAll()` dient, wartet diese Methode `wait(int)` einfach nur eine angegebene Zeitdauer. Dabei behält der gerade aktive Thread alle akquirierten Locks. Im Gegensatz dazu wartet die Methode `Object.wait(long)` zwar auch eine gewisse Zeitdauer auf das Eintreffen einer Nachricht, allerdings wird hierbei der Lock auf das Objekt wieder freigegeben (vgl. Abschnitt 10.2.2).

Fallstrick: Mehrfachverkettung der ».«-Notation

Die Hintereinanderschaltung verschiedener Methodenaufrufe mit der ».«-Notation birgt die Gefahr nicht initialisierter Referenzen. Methodenaufrufe lösen eine `NullPointerException` aus, wenn eine der Referenzen `null` ist. Außerdem widerspricht die mehrfache ».«-Notation dem in Abschnitt 3.5.2 vorgestellten Law of Demeter und wird schnell unübersichtlich. Das folgende Beispiel zeigt eindrucksvoll, dass schon wenige Verkettungen problematisch sind, wobei wir den Fokus auf die Lesbarkeit und Verständlichkeit legen, da im Beispiel die Referenzvariablen korrekt initialisiert sind:

```
public static long calcMinutesToGo(final Departure departure)
{
    return departure.getDepartureTimeInMillis() -
           Calendar.getInstance().getTimeInMillis() / 1000 / 60 /*[min]*/;
}
```

Abhilfe schafft das Refactoring EXTRACT LOCAL VARIABLE. Bei Eclipse findet man dieses im Menü REFACTOR und aktiviert es mit dem Tastaturkürzel ALT+SHIFT+L. Damit kann man Ausdrücke in lokale Variablen ausgliedern und den ursprünglichen Ausdruck vereinfachen, wie dies im folgenden Listing gezeigt ist:

```
public static long calcMinutesToGo(final Departure departure)
{
    final long departureTimeInMillis = departure.getDepartureTimeInMillis();
    final long nowInMillis = Calendar.getInstance().getTimeInMillis();

    return (departureTimeInMillis - nowInMillis) / 1000 / 60 /*[min]*/;
}
```

Durch die deutlich übersichtlichere und verständliche Umsetzung entdeckt man vermutlich auch den im ersten Ausdruck versteckten Berechnungsfehler: Dort wurde nur der zweite Wert in Minuten umgewandelt, der erste Wert war noch in Millisekunden. Zur Korrektur wurde oben bereits die Klammerung um die Subtraktion hinzugefügt.

Soll der Sourceocde die Umwandlung von Millisekunden in Minuten noch klarer kommunizieren, kann man dazu die Aufzählung `TimeUnit` wie folgt nutzen:

```
public static long calcMinutesToGo(final Departure departure)
{
    final long departureTimeInMillis = departure.getDepartureTimeInMillis();
    final long nowInMillis = Calendar.getInstance().getTimeInMillis();

    return TimeUnit.MILLISECONDS.toMinutes(departureTimeInMillis - nowInMillis);
}
```

Fallstrick: Unnötige Methoden und Komplexität

Leider sieht man immer wieder Sourcecode, der keinen Mehrwert hat, sondern sogar eher der Übersicht sowie der Nachvollziehbarkeit schadet. Unglaublich, aber wahr, ist das folgende Beispiel der Methode `getString(Object)`:

```
public static final String getString(final Object argument)
{
    return argument.toString();
}
```

Diese Methode ist vollkommen überflüssig, da sie lediglich zusätzliche Komplexität einführt. Ein direkter Aufruf von `toString()` würde den Sourcecode klarer machen und das intuitive Verständnis erleichtern. Als Faustregel gilt: ***Eine Methode sollte nur dann erstellt werden, wenn sie neue Funktionalität hinzufügt oder für eine bessere Struktur oder Lesbarkeit des bestehenden Sourcecodes sorgt***.

Abgesehen davon existiert ein noch viel schwerwiegenderes Problem: Jeder Aufrufer besitzt nun eine Abhängigkeit von der Utility-Klasse `StringUtils`, in der die Methode `getString(Object)` definiert ist (vgl. Abbildung 16-9). Würde diese Utility-Klasse auf weitere Klassen verweisen, so bekäme man schnell eine Menge ungewünschter und unerwarteter Abhängigkeiten.

Abbildung 16-9 *Unerwartete Abhängigkeiten durch* `getString(Object)`

Fallstrick: Objektvergleich durch Einsatz von `toString()` und `equals()`

Die Methode `equals(Object)` dient bekanntermaßen dazu, einen semantischen Vergleich von Objekten zu ermöglichen (vgl. Abschnitt 5.1.2). Ist die `equals(Object)`-Methode nicht überschrieben, findet man mitunter folgendes Konstrukt, um dennoch inhaltliche Vergleiche ausführen zu können: Zunächst erfolgt eine Umwandlung in einen String durch einen Aufruf von `toString()`. Die so erzeugten Stringrepräsentationen werden dann mit deren `equals(Object)`-Methode verglichen. Im Folgenden wird für ein Objekt `databaseJob` über den Aufruf von `toString()` eine textuelle Repräsentation erzeugt und diese anschließend mit einer Variablen `jobId` verglichen:

```
if (databaseJob.toString().equals(jobId))
```

Dieser Vergleich verlässt sich auf das Implementierungsdetail der `toString()`-Methode dieses Objekts, dass in diesem Beispiel nur der Wert der `jobId` des Ob-

jekts zurückgeliefert wird. Eine derartige Annahme ist gewagt und der resultierende Vergleich ist fragil. Wird die Ausgabe der `toString()`-Methode im Nachhinein verändert, z. B. um weitere Ausgaben von Attributwerten ergänzt oder selbst nur minimal verändert, so schlägt der gezeigte Vergleich immer fehl. Ein solcher Fehler ist schwierig zu finden, da er sich lediglich indirekt durch Merkwürdigkeiten im Programmablauf bemerkbar macht. Liefert die selbst definierte Methode `toString()` gar den Wert `null` zurück, so ergibt sich ein weiterer, später behandelter Fallstrick.

Fallstrick: Zugriff ohne Bereichsprüfung

Zum Teil sieht man Zugriffe auf Arrays, Strings, Listen, Iteratoren und Enumerations ohne vorherige Prüfung, ob dieser Zugriff überhaupt möglich und erlaubt ist (Index im Bereich, Daten vorhanden usw.). Ich möchte hier nicht zum übervorsichtigen Prüfen vor jedem Zugriff aufrufen, sondern lediglich motivieren, die Validität von Zugriffen abzusichern. Häufig kann dies an zentraler Stelle einmal geschehen. Nach einer Prüfung müssen aufgerufene Methoden diese nicht wiederholen, sofern sich die Zusammensetzung der Daten nicht ändert. Ansonsten kann auch eine erneute Prüfung notwendig sein.

Um Zugriffsprobleme zu vermeiden, sollten zumindest einmalig folgende Tests durchgeführt werden:

- Bei Array-Zugriffen sollte die Indexposition mit der Array-Größe verglichen werden. Gleiches gilt für Listen und einen indizierten Zugriff mit `get(int)`.
- Indizierte Zugriffe per `charAt(int)` in Strings sollten die Stringlänge prüfen.
- Beim Einsatz eines Iterator- oder Enumeration-Zugriffs ist es erforderlich, durch Aufruf der Methode `hasNext()` bzw. `hasMoreElements()` festzustellen, ob weitere Elemente existieren.

Fallstrick: Intensive Nutzung von Sprungmarken, `break` und `continue`

Die intensive Nutzung von Sprungmarken, `break` und `continue` erinnert an die GOTO-Programmierung lange vergangener Tage. In den allermeisten Fällen lassen sich Abbruchbedingungen klarer durch boolesche Variablen formulieren.

Fallstrick: `default` mitten in den `case`-Anweisungen versteckt und/oder unerwartetes Fallthrough

Ein weiterer Fallstrick ist die Angabe der `default`-Aktion zwischen beliebigen Alternativen einer `switch`-Anweisung. Ein intuitives Verständnis des Ablaufs wird dadurch erschwert. Es ist guter Programmierstil, die Standardaktion, die nicht durch `case` abgedeckt wird, am Ende der `case`-Alternativen aufzuführen.

Praktischerweise wurde mit Java 14 die Syntax von `switch` deutlich überarbeitet. Durch die Neuerungen gibt es kein Fallt Through mehr und somit verliert ein `default` zwischen den `case`s seinen Schrecken, aber nicht sein schlechtes Geschmäckle.

```
switch (value)
{
   case 0:
   case 2:
   case 4:
       System.out.println(value + " accepted even value [0 - 4]");
       // Fehlendes break => Fallthrough

   default:
       System.out.println(value + " performing default action");
       break;

   case 1:
   case 3:
       System.out.println(value + " is odd");
}
```

Das Listing zeigt aber eine weitere Unschönheit: Standardmäßig erfolgt bei Angaben in `switch` ein Fallthrough, d. h., für die Werte 0, 2 und 4 wird nicht nur ihr `case`-Zweig ausgeführt, sondern auch noch der nachfolgende Defaultzweig, weil kein `break` aufgeführt ist.

Fallstrick: Berechnungen in `case`-Anweisungen

Fallunterscheidungen mit `case` sollte man möglichst lesbar und verständlich halten. Obwohl es möglich ist, Berechnungen innerhalb von `case` auszuführen, sollte man dies vermeiden, da darunter die Nachvollziehbarkeit und Wartbarkeit leiden. Folgendes Listing zeigt eindrucksvoll, wie kompliziert der Wertebereich 7 – 10 ausgedrückt werden kann:

```
public static void main(final String[] args)
{
    final int byteValue = 7;

    int value = (int) (10 * Math.random());
    switch (value)
    {
        case 7:
        case 19 - 11:           // 8
        case 17 - 8:            // 9
        case 2 * byteValue - 4: // 10
            System.out.println(value + " accepted [7 - 10]");
            break;

        default:
            System.out.println(value + " is not in range 7 - 10");
    }
}
```

Listing 16.4 *Ausführbar als* **'BADCASEEXAMPLE'**

Startet man das Programm BADCASEEXAMPLE, so kommt es für den Wert 9 zu folgender Ausgabe:

```
9 accepted [7 - 10]
```

Fallstrick: Rückgabe von `null` in `toString()`-Methoden

Eine `toString()`-Methode soll eine textuelle Repräsentation eines Objekts erzeugen. Sie sollte allerdings niemals den Wert `null` zurückgeben, denn dadurch provoziert sie `NullPointerException`s im aufrufenden Sourcecode.

Fallstrick: Einsatz der Literale `false` und `true` in booleschen Vergleichen sowie künstlich komplizierte boolesche Bedingungen

Der Einsatz boolescher Bedingungen dient der Auswertung verschiedener Zustände. Nicht immer muss eine solche Bedingung aber so kompliziert geschrieben werden, wie dies manchmal ohne viel Nachdenken geschieht. Insbesondere sollte man den expliziten Vergleich mit den Literalen `false` und `true` vermeiden. Der Wert `false` lässt sich durch eine Negation mit `!` ersetzen.

Im folgenden Listing wird ein Wechsel im Betriebszustand festgestellt, indem das boolesche Attribut für den alten Zustand `lastWorkingState` und die boolesche Variable `newWorkingState` für den aktuellen Zustand verglichen werden:

```
if ((lastWorkingState == false && newWorkingState == true) ||
    (lastWorkingState == true  && newWorkingState == false))
```

Tatsächlich lässt sich dies mühelos wie folgt klarer umschreiben, wenn man zunächst einmal die Literale vermeidet

```
if ((!lastWorkingState && newWorkingState) ||
    (lastWorkingState && !newWorkingState))
```

und danach einen weiteren Optimierungsschritt vornimmt:

```
if (lastWorkingState != newWorkingState)
```

Fallstrick: Einsatz doppelter Verneinung

Der Einsatz von booleschen Variablen kann zur Klarheit von Programmen enorm beitragen. Werden jedoch Bedingungen umständlich oder als Verneinung formuliert, so stört dies die Lesbarkeit. Ein Beispiel, das von mir exakt so im Sourcecode eines realen Projekts gefunden wurde, ist die Variable `dontCleanUpDatabase`:

```
boolean dontCleanUpDatabase = false;
```

Für den Wert von `false` wird die Datenbank aufgeräumt. Für den Wert `true` geschieht dies nicht und es bleiben Datensätze erhalten. Diese doppelte Verneinung erschwert das Verständnis und ist nicht intuitiv. Je nach Einsatzzweck wären folgende Variablennamen besser verständlich gewesen:

```
boolean cleanUpDatabase = true;
boolean keepDatabaseEntries = false;
```

Fallstrick: Keine Klammern um Blöcke

Im Rahmen dieses Fallstricks wird beschrieben, warum es sinnvoll ist, Blöcke normalerweise durch Klammern zu kennzeichnen, und welche Probleme auftreten können, wenn man dies nicht tut.

Das folgende Beispiel stammt aus einer Implementierung einer Tabelle, die sortierbare Spalten anbietet. Der Sourcecode sieht vereinfacht so aus:

```
if (COLUMN_PATH.equals(strColumnName))
    // do not sort path

if (COLUMN_CREATION_DATE.equals(strColumnName))
    Collections.sort(documents, new CreationDateComparator(ascending));

if (COLUMN_MODIFICATION_DATE.equals(strColumnName))
    Collections.sort(documents, new ModificationDateComparator(ascending));

...
```

Obwohl für alle Spalten (außer dem Pfad) passende Komparatoren hinterlegt sind, zeigen erste Tests, dass die Inhalte gewisser Spalten nicht sortiert werden können. Mit ein wenig Testaufwand findet man heraus, dass dieser Fehler tatsächlich nur den Inhalt der Spalte `COLUMN_CREATION_DATE` betrifft.

Wie ist das zu erklären? Für alle Leser, die bereits sensibilisiert sind, ist der Fall sofort klar. Für alle anderen vereinfache ich den Sourcecode. Wenn die Variable `strColumnName` den Wert `COLUMN_CREATION_DATE` besitzt, steht dort:

```
if (false)
    // do not sort path

if (true)
    System.out.println("sort");
```

Führt man den Sourcecode aus, so gibt er nichts aus! Das liegt daran, dass ein einzeiliger Kommentar laut JLS keine Anweisung ist. Aufgrund dessen wird der Sourcecode folgendermaßen ausgewertet – dies derart formatiert und mit eingefügten Hilfsklammern ist klar, dass nichts ausgegeben wird:

```
if (false)
{
    // do not sort path

    if (true)
        System.out.println("sort");
}
```

Demnach erschweren es fehlende Klammern um Blöcke, den Programmablauf basierend auf dem Sourcecode-Layout nachzuvollziehen. Die Auswertung von Bedingungen geschieht dann zum Teil nicht so, wie man es aufgrund der Formatierung erwarten würde. Im Besonderen gilt dies, wenn mehrere `if`-Anweisungen aufeinanderfolgen.

Um Probleme im Voraus zu verhindern, sollte man bevorzugt Klammern um Blöcke verwenden. Ausgenommen davon sind einzeilige Shortcut-Returns oder `throw`-Anweisungen in Zustandsprüfungen am Methodenanfang. Werden jedoch Anweisungen oder ein `else` ergänzt oder möchte man sich bei Modifikationen keine Gedanken machen müssen, ob man etwas am Ablaufverhalten ändert, empfiehlt es sich fast immer, Klammern zu verwenden.

Fallstrick: Kommentierte Klammern

Manchmal werden schließende Klammern kommentiert, um einen Hinweis auf die Anweisung der zugehörigen öffnenden Klammer zu geben. Das zeigt oft eigentlich nur, dass die Struktur des Programms unübersichtlich ist und die Blöcke zu groß gewählt sind. Betrachten wir ein Beispiel, das von der absoluten Länge her in Ordnung ist, aber durch seine kommentierten Klammern bereits unübersichtlich wird:

```
public void writeXML(final String filename)
{
    try
    {
        final Element root = toDOMElement();
        final Document doc = new Document(root);
        try
        {
            final XMLOutputter xmlout = createXMLOutputter();
            xmlout.output(doc, new FileOutputStream(filename));
        } // try
        catch (final IOException e)
        {
            log.error("Can't write " + filename, e);
        } // catch
    } // try
    catch (final NullPointerException npe)
    {
        log.error("Can't create " + filename, npe);
    } // catch
} // writeXML
```

Durch die massive Kommentierung der Klammern kann man dem Programmfluss in diesem Beispiel eher schlecht folgen. Tatsächlich sieht man erst auf den zweiten Blick, dass hier fälschlicherweise keine Ressourcenfreigabe für den `FileOutputStream` erfolgt. Zudem könnte man sich über das Abfangen der `NullPointerException` wundern. Das ist für sich schon ein Bad Smell.

Fallstrick: `finally`-Block mit Exception verlassen

Bei der Behandlung von Fehlersituationen dienen `finally`-Blöcke dazu, gewisse Aktionen abschließend garantiert ausführen zu können. Die dort angegebenen Aktionen sollten zuverlässig und überraschungsfrei sein, also wieder dem POLA bzw. POLS folgen. Dagegen wird jedoch verstoßen, wenn innerhalb eines `finally`-Blocks eine Exception ausgelöst wird und dadurch nur ein Teil der Aufräumarbeiten erfolgt. Betrachten wir zum besseren Verständnis folgendes Beispiel:

```
finally
{
    cleanUpPart1();

    if (someCondition)
        throw new IllegalStateException("Condition ...");

    cleanUpPart2();
}
```

Ist die Bedingung `someCondition` erfüllt, würde die Methode `cleanUpPart2()` nicht aufgerufen und somit würden auch die dortigen Aufräumarbeiten und Ressourcenfreigaben nicht ausgeführt. Allerdings wird kaum jemand derart offensichtlich in einem `finally`-Block bewusst eine Exception auslösen. Deutlich anders sieht es jedoch aus, wenn in dem `finally`-Block Methoden aufgerufen werden, die potenziell Exceptions auslösen können. Würden die obigen zwei Zeilen Bestandteil der Methode `cleanUpPart1()`, so sähe der `finally`-Block harmlos und korrekt aus:

```
finally
{
    cleanUpPart1();  // Achtung: Versteckte IllegalStateException
    cleanUpPart2();
}
```

Um sich vor Merkwürdigkeiten bei der Abarbeitung von `finally`-Blöcken zu schützen, sollten die dort aufgerufenen Methoden oder Anweisungen möglichst keine Exceptions auslösen. Lässt sich eine solche Exception nicht verhindern (z. B. weil man den Sourcecode der aufgerufenen Methode nicht im Zugriff hat), so kann man ineinander verschachtelte `try`-`finally`-Blöcke wie folgt nutzen:

```
finally
{
   try
   {
       cleanUpPart1();  // Achtung: Versteckte IllegalStateException
   }
   finally
   {
      cleanUpPart2();
   }
}
```

Fallstrick: Zeitmessungen mit `System.currentTimeMillis()`

Manchmal sieht man Sourcecode, dessen Ausführungszeit statt mit einem Profiler über Aufrufe mit `System.currentTimeMillis()` gemessen werden soll, etwa wie bei folgender Schleife für die Ausführungsdauer einer komplexen Berechnung:

```
long runtimeInMs = 0;

for (int i = 0; i < count; i++)
{
    final long start = System.currentTimeMillis();
    performComplexCalculation();
    final long end = System.currentTimeMillis() - start;

    runtimeInMs += end;
}
```

Die von `System.currentTimeMillis()` gelieferten Werte sind leider recht ungenau. In der Regel deutlich präziser – weil auf dem genauesten Timer des Betriebssystems basierend – sind Aufrufe von `System.nanoTime()`.

Aber selbst damit wäre eine solche Zeitmessung noch ungeschickt, da hier einzelne Laufzeiten addiert werden, statt eine Gesamtlaufzeit zu ermitteln. Auch der Variablenname `end` ist irreführend, weil hier die Laufzeit einer Einzelberechnung gemessen wird und nicht der Endezeitpunkt.

```
long runtimeInMs = 0;

final long start  = System.nanoTime();
for (int i = 0; i < count; i++)
{
    performComplexCalculation();
}
final long runtimeInNanos = System.nanoTime() - start;
final long runtimeInMs = TimeUnit.NANOSECONDS.toMillis(runtimeInNanos);
```

Bei Bedarf kann dann mithilfe der Klasse `TimeUnit` eine Umwandlung in eine beliebige andere Zeiteinheit außer den hier gezeigten Millisekunden erfolgen.

Fallstrick: Wahllos `synchronized`

Manchmal sieht man Sourcecode, in dem Methoden nahezu wahllos `synchronized` sind – oft im guten Willen, um für Thread-Sicherheit zu sorgen, etwa wie folgt:

```
public static synchronized String convertWindowsToUnixPath(final String input)
{
    final String result = input.replaceAll("\\\\", "/");
    return result;
}
```

Schauen wir uns die Problematik an: In diesem Fall verändert die Methode offensichtlich nur den lokalen Zustand. Dadurch kann es niemals zu Multithreading-Problemen kommen. Demnach ist auch kein `synchronized` nötig, sondern sogar überflüssig und sinnlos. Schlimmer noch: Anderer Sourcecode, der auch einen kritischen Bereich benö-

tigt, kann nun eventuell nicht mehr ausgeführt werden, solange die obige Methode den Lock der Objektinstanz besitzt. Hier wird der Aufruf von `replaceAll()` wenig Einfluss haben, würden aber komplexere Aktionen ausgeführt, würde man Nebenläufigkeit potenziell stärker behindern.

Fallstrick: Berechnungen und Logik in Ausgaben versteckt

Mitunter erfolgen wichtige Berechnungen unscheinbar innerhalb von Programmausgaben etwa wie im folgenden Beispiel:

```
System.out.println("Terminated all?" +
                   (executorService.awaitTermination(10, TimeUnit.SECONDS);
```

Eine solche Aktion kann leicht übersehen werden, wenn man nicht sehr achtsam ist. Dadurch wird später möglicherweise eine Änderung vorgenommen und der entscheidende Abschluss des `ExecutorService` nicht mehr ausgeführt, beispielsweise wenn die Ausgabe nicht mehr erwünscht ist – die Aktion aber schon. Allerdings ist ganz schnell die Programmzeile und damit auch die Logik auskommentiert. In seiner Auswirkung erinnert dieser Fallstrick somit ein wenig an den BAD SMELL: PROGRAMMCODE IM LOGGING-CODE aus Abschnitt 16.1.5.

Es empfiehlt sich, wichtige Aktionen bzw. Logik explizit außerhalb einer Ausgabe vorzunehmen, wie es die folgenden Zeilen zeigen:

```
final boolean terminatedAll =
              executorService.awaitTermination(10, TimeUnit.SECONDS);

System.out.println("Terminated all?" + terminatedAll);
```

Würde die Programmausgabe entfernt, weil sie doch nicht mehr gewünscht ist, so verändern wir die Logik erwartungskonform nicht.

Fallstrick: Falsche Annahmen bei Typprüfungen

Es gibt immer mal wieder Situationen, in denen eine explizite Typprüfung per `instanceof` erfolgt, um typspezifische Aktionen auszuführen – im Beispiel für zwei verschiedene Typen von Views:

```
public void reportViewClosing(final Object view)
{
    String msg;
    if (view instanceof DetailView)
    {
        msg = ((DetailView)view).getName() + " closed";
    }
    else
    {
        msg = ((SummaryView)view).getName() + " closed";
    }
    ViewTracker.updateLog(msg);
}
```

An diesem kurzen Sourcecode-Schnipsel (abgesehen von der Namensgebung tatsächlich aus Produktivcode stammend) ist zunächst einmal merkwürdig, dass zweimal die gleiche Aktion ausgeführt wird. Zudem würde man bei verschiedenen Views annehmen, dass diese einen gemeinsamen Basistyp besitzen. Schlimmer noch: Der Sourcecode enthält einige Annahmen, sodass die Methode nur dann korrekt arbeitet, wenn als Übergabe lediglich die Typen `DetailView` oder `SummaryView` genutzt werden – für alle anderen Typen, z. B. `ErrorLogView`, erhalten wir eine `ClassCastException`. Das ist alles andere als intuitiv und stabil.

Eine erste Variante wäre folgender Sourcecode, der zwei explizite Typprüfungen enthält, die jeweils typabhängige Aktionen (hier die gleiche) ausführen. Im Falle eines nicht unterstützten Typs habe ich eine Fehlerbehandlung angedeutet, die alternativ eine Rückfallmeldung oder eine `IllegalStateException` bewirkt:

```
public void reportViewClosing(final Object view)
{
    String msg = null;
    if (view instanceof DetailView)
    {
        msg = ((DetailView)view).getName() + " closed";
    }
    else if (view instanceof SummaryView)
    {
        msg = ((SummaryView)view).getName() + " closed";
    }
    else
    {
        // throw new IllegalStateException("Unsupported view type " + view.
              getClass());
        msg = "Unsupported view type: '" + view.getClass().getName() + "'";
    }
    ViewTracker.updateLog(msg);
}
```

Viel besser wäre es jedoch (falls noch nicht existent), einen Basistyp `View` mit der Methode `getName()` einzuführen und dann Polymorphie zu nutzen, wodurch der ganze Sourcecode nahezu in sich zusammenfällt und absolut einfach und verständlich wird:

```
public void reportViewClosing(final View view)
{
    final String msg = view.getName() + " closed";
    ViewTracker.updateLog(msg);
}
```

Fallstrick: Implizite Aktionen im Konstruktor

Ab und an findet man Klassen, die in ihren Konstruktoren neben der Initialisierung noch diverse Aktionen ausführen. Das ist in der Regel schon mal nicht gut. Schlimmer sind allerdings implizite Annahmen oder Aktionen, weil Abhängigkeiten oder Reihenfolgen nicht klar ersichtlich sind. Als Negativbeispiel betrachten wir die zur Serialisierung gedachten Klassen `ObjectInputStream` und `ObjectOutputStream`. Möchte man Daten in serialisierter Form über ein Netzwerk versenden, so muss man wissen, dass der Konstruktor des `ObjectOutputStream`s spezielle Header-Daten in den Ausgabe-Stream schreibt. Der korrespondierende `ObjectInputStream`-Konstruktor blockiert, bis diese Daten vom zugeordneten `ObjectOutputStream` geschrieben wurden. Daher empfiehlt es sich, dass man direkt nach der Konstruktion eines `ObjectOutputStream`s die Methode `flush()` aufruft:

```
try (ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
     ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream()))
{
    oos.flush();

    oos.writeUTF("Info");
    oos.writeInt(1234567);
    oos.writeObject(new Person("Mike", "Zürich"));
}
```

Diese Verhaltensweise, also das implizite Wissen über Eigenarten der Verarbeitung in den Konstruktoren, muss in eigenen Anwendungen bekannt sein und beachtet werden, damit sich die Kommunikationspartner nicht gegenseitig blockieren. Idealerweise sollten Objekte keine solchen Abhängigkeiten bei ihrer Konstruktion besitzen. Eine Abhilfe können Fabrikmethoden sein, die in Abschnitt 18.1.2 vorgestellt werden.

16.5 Weiterführende Literatur

Zwei empfehlenswerte Bücher, die sich mit Bad Smells, aber vor allem auch mit dem Schreiben von verständlichem Sourcecode beschäftigen, sind folgende:

- »**Code Craft: The Practice of Writing Excellent Code** « von Pete Goodliffe [25]
  Pete Goodliffe hat ein fantastisches Buch zu gutem Softwareentwurf geschrieben. Es werden viele Tipps gegeben, um sauberen, robusten Sourcecode zu schreiben. Ich empfehle es ausdrücklich. Ein Must-have!
- »**Clean Code: A Handbook of Agile Software Craftsmanship**« von Robert C. Martin [50]
  In diesem Buch von Robert C. Martin findet man hilfreiche Informationen zu sauberem Sourcecode und gutem Programmierstil. Allerdings ist es mir manchmal ein wenig (zu) dogmatisch.

17 Refactorings

Bei der Entwicklung und Pflege von Software ist das Gesetz der Entropie (der zunehmenden Unordnung) aus der Physik zu beobachten. Man kennt Ähnliches aber auch aus dem täglichen Leben. In einer Wohnung nimmt die Unordnung ständig zu, wenn man nicht ab und zu aufräumt. Auf Software übertragen gilt, dass sich im Laufe der Zeit durch Änderungen die Struktur und Lesbarkeit derart verschlechtern können, dass sich Fehlerbehebungen oder Erweiterungen immer schwieriger in den bestehenden Sourcecode integrieren lassen.

Das Thema Wartbarkeit ist beim professionellen Programmieren aber sehr wichtig, denn oftmals hat man eine große Sourcecode-Basis zu pflegen und zu erweitern. Nur selten kommt man in den Genuss, ein System vollständig neu entwerfen zu dürfen. Und auch in diesem Fall wandelt sich die Situation schnell, wenn man nicht fortlaufend Qualitätssicherung betreibt und kontinuierlich Überarbeitungen durchführt. Wurde dies versäumt, so hilft – wie im realen Leben – nur eine gründliche Aufräumaktion oder gar ein Umzug. Letzteres entspräche auf die Software übertragen einer kompletten Neuimplementierung. Eine solche ist aufgrund des hohen Risikos, zu scheitern, jedoch meistens keine Alternative. Es verbleibt die Aufräumaktion, die Umbaumaßnahmen im Sourcecode, sogenannten ***Refactorings***, entspricht. Diese sollen problembehafteten Sourcecode derart verändern, dass dieser besser verständlich, lesbar und auch leichter wartbar wird. Martin Fowler definiert den Begriff Refactoring folgendermaßen: »Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify without changing the observable behavior of that software component« [20].

Dieses Kapitel stellt verschiedene in der Praxis erprobte Überarbeitungen von Sourcecode vor. In Abschnitt 17.1 betrachten wir ein einführendes Beispiel. Beim Überarbeiten des Sourcecodes ist ein Standardvorgehen hilfreich, das ich in Abschnitt 17.2 beschreibe. Dieses sorgt zunächst mit einigen Vorbereitungsmaßnahmen und der Beachtung von Coding Conventions (vgl. Kapitel 19) dafür, dass die weitere Bearbeitung und gewünschte Transformation leichter fallen. Je nach erkannter Schwachstelle kann man gemäß den Schritt-für-Schritt-Anleitungen aus Abschnitt 17.4 vorgehen. ***Nicht bei allen davon handelt es sich im strengen Sinne um Refactorings, da diese mitunter das nach außen sichtbare Verhalten leicht ändern und somit die obige Forderung nicht strikt einhalten***. Allerdings gibt es auch Meinungen, die die Aussage auf relevantes Verhalten lockern. Unabhängig von der genauen Auslegung spreche ich der Einfachheit halber immer von Refactorings. Natürlich müssen derartige Änderungen mit

Vorsicht und Sorgfalt ausgeführt werden, um dadurch keine Fehler einzuführen oder das Programmverhalten grundlegend zu ändern. Dazu ist es sinnvoll, Veränderungen in kleinen, überschaubaren Schritten durchzuführen und für eine Absicherung durch Unit Tests zu sorgen (vgl. Kapitel 20).

17.1 Refactorings am Beispiel

Anhand eines von mir so in der Praxis gefundenen Beispiels möchte ich darstellen, welche Möglichkeiten zur Vereinfachung sich mit Refactorings ergeben können. Folgende statische Methode `isNumber(String)` der Klasse `NumberUtilsV1` prüft auf naive Weise, ob ein übergebener String eine ganze Zahl darstellt. Dazu wird vor allem die Methode `Character.isDigit(char)` verwendet:

```
public static boolean isNumber(final String value)
{
    if (Character.isDigit(value.charAt(0)))
    {
        for (int i = 1, n = value.length(); i < n; i++)
        {
            if (!(Character.isDigit(value.charAt(i))))
            {
                return false;
            }
        }
    }
    else
    {
        return false;
    }
    return true;
}
```

In der Methode finden zwei `if`-Abfragen statt. Zunächst wird das erste Zeichen geprüft. Nur wenn dieses eine Ziffer ist, wird anschließend in einer `for`-Schleife beginnend ab Index 1 der `isDigit(char)`-Vergleich wiederholt, bis entweder das Ende des Strings erreicht oder das betrachtete Zeichen keine Ziffer ist. Durch etwas Sourcecode-Analyse erkennen wir, dass die Methode `isNumber(String)` folgende Probleme enthält:

1. **Unerwartete Exception bei leerer Eingabe** – Wenn die Eingabe nicht mindestens ein Zeichen enthält, wird eine `java.lang.StringIndexOutOfBoundsException` ausgelöst. Die Ursache ist der indizierte Zugriff per `charAt(0)`, ohne zuvor die Länge des Parameters `value` zu überprüfen. Ein solches Verhalten ist zu vermeiden. Dies gilt im Speziellen, wenn die Werte aus einer Benutzereingabe stammen.
2. **Fehleranfällig für `null`** – Bei Übergabe eines `null`-Werts löst die Methode erst bei der Verarbeitung und dem Aufruf von `charAt(0)` eine `NullPointerException` aus. Öffentliche Methoden sollten gemäß Design by Contract (vgl. Abschnitt 3.1.5) ihre Vorbedingungen sicherstellen und dazu vor der eigentlichen

Verarbeitung die Eingabeparameter auf Gültigkeit prüfen. Damit wird eine Störung der Abarbeitung durch fehlerhafte Übergabewerte vermieden.

3. **Zu kompliziert** – Die initiale `if`-Bedingung und das `if` innerhalb der `for`-Schleife sorgen für eine weitere Schachtelungsebene und sind konträr zueinander formuliert. Bei flüchtigem Hinsehen könnte man die Abweichungen in der initialen Prüfung übersehen und das Ganze falsch zusammenführen.
4. **Missverständlicher Name bzw. unklares Verhalten** – Der Methodenname `isNumber(String)` suggeriert die Möglichkeit der Verarbeitung beliebiger Zahlen. Momentan werden aber weder Zahlen mit Vorzeichen noch solche mit Nachkommastellen unterstützt.
5. **Viele `return`-Anweisungen** – Für eine derart kurze Methode existieren mit drei `return`-Anweisungen schon recht viele Ausgänge.

Um die Funktionalität zu prüfen und mögliche Fehler aufzudecken, entwickeln wir einige elementare Unit Tests mit JUnit 5. Dabei können uns die eben ermittelten obigen Kritikpunkte helfen, mögliche Testfälle zu identifizieren und diese Schwachstellen für die Zukunft auszuschließen. Das ist nützlich, weil wir ergänzend kleinere Änderungen und Erweiterungen realisieren wollen. Damit beginnen wir jedoch erst, nachdem wir ein Sicherheitsnetz aus diversen Testfällen erstellt haben.

Problem 1: Unerwartete Exception bei leerer Eingabe

Bevor wir uns dem eigentlichen Problem bei leeren Eingaben zuwenden, erstellen wir einige Funktionstests: Wir prüfen die Verarbeitung einer gültigen Eingabe, etwa `"12345"`, und eines fehlerhaften Werts, etwa `"ABC"`. Im ersten Fall erwarten wir `true` und im zweiten `false` als Ergebnis. Dies lässt sich mithilfe von JUnit und dessen Methoden `assertTrue(boolean)` und `assertFalse(boolean)` ausdrücken. Diese beiden Tests sollten bestanden werden.

```
@Test
public void testValidNumberInput()
{
    assertTrue(NumberUtilsV1.isNumber("12345"));
}

@Test
public void testInvalidInput()
{
    assertFalse(NumberUtilsV1.isNumber("ABC"));
}
```

Wir müssen uns aber noch mindestens zwei weiteren Fällen widmen, nämlich einer leeren Eingabe und einer Eingabe mit nur einer Ziffer. Unsere Voranalyse hat schon aufgedeckt, dass es dabei Probleme gibt. Wir schreiben nun passende Unit Tests, die das bestätigen. In der Zukunft soll die Methode für eine leere Eingabe keine `StringIndexOutOfBoundsException` auslösen, sondern den Wert `false` für die Aussage »keine Zahl« liefern. Eine einzelne Ziffer gilt als Zahl.

Basierend auf den gerade gemachten Vorüberlegungen komplettieren wir den Unit Test wie folgt:

```
@Test
public void testNumberInputLength0()
{
    assertFalse(NumberUtilsV1.isNumber(""));
}

@Test
public void testNumberInputLength1()
{
    assertTrue(NumberUtilsV1.isNumber("1"));
}
```

Listing 17.1 *Ausführbar als* '**NUMBERUTILSV1TEST**'

Wie erwartet, schlägt der Test `testNumberInputLength0()` mit leerer Eingabe fehl.

Korrektur Bevor wir weitere Tests ergänzen, korrigieren wir die Funktionalität. Wir fügen folgende Sicherheitsprüfung am Anfang der Methode `isNumber()` hinzu:

```
if (value.isEmpty())
{
    return false;
}
```

Nach dieser Korrektur werden die Unit Tests wieder bestanden. Allerdings ist unser Nutzcode durch die Abfrage und das weitere `return` etwas komplizierter geworden – aber wir leben erst mal damit. Darum kümmern wir uns, sobald wir das Sicherheitsnetz aus Unit Tests ausgebaut haben. Das soll dabei helfen, Änderungen an der inneren Struktur mit mehr Vertrauen in die Korrektheit ausführen zu können.

Problem 2: Fehleranfällig für `null`

In der initialen Analyse haben wir erkannt, dass die Eingabe von `null` unbehandelt ist und eine `NullPointerException` auslöst – allerdings nicht durch eine Parameterprüfung, sondern erst beim Zugriff auf den Parameter `value`. Wir wollen prüfen, dass eine explizite Behandlung erfolgt und ein Hinweis in der ausgelösten Exception mehr Informationen liefert. Mit JUnit 5 lässt sich das elegant mithilfe von `assertThrows()` formulieren:

```
@Test
public void testNullInput()
{
    NullPointerException npe = assertThrows(NullPointerException.class,
                        () -> NumberUtilsV2EmptyInputCorrected.isNumber(null));

    assertFalse(StringUtils.isEmpty(npe.getMessage()));
}
```

Listing 17.2 *Ausführbar als* '**NUMBERUTILSV2NULLINPUTTEST**'

Abschließend wird durch Aufruf der Methode `isEmpty(String)` der Klasse `StringUtils` aus Apache Commons Lang[1] getestet, ob ein Hinweistext hinterlegt ist.

Und wie erwartet, schlägt auch dieser Test zunächst fehl, weil der Hinweistext leer ist, da die Exception durch die JVM automatisch beim Dereferenzieren der `null`-Referenz ausgelöst wurde. Das Ganze können Sie prüfen, wenn Sie den Unit Test als NUMBERUTILSV2NULLINPUTTEST starten.

Korrektur Wir haben in Kapitel 9 die Bibliothek Google Guava und deren Utility-Klasse `Preconditions` zum Sicherstellen von Vorbedingungen kennengelernt. Dementsprechend fügen wir eine Sicherheitsprüfung am Anfang der Methode hinzu:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    if (Character.isDigit(value.charAt(0)))
    {
        for (int i = 1, n = value.length(); i < n; i++)
        {
            if (!(Character.isDigit(value.charAt(i))))
            {
                return false;
            }
        }
    }
    else
    {
        return false;
    }
    return true;
}
```

Der zugehörige Test lässt sich als Programm NUMBERUTILSV3NULLINPUTCORRECTEDTEST ausführen. Es zeigt sich, dass durch die obige Korrektur die Unit Tests wieder bestanden werden. Allerdings ist unser Nutzcode durch die Abfrage nochmals etwas komplizierter geworden. Da wir aber mittlerweile ein recht gutes Sicherheitsnetz erstellt haben, ist es nun an der Zeit, den Nutzcode – sofern möglich – ein wenig aufzuräumen.

Problem 3: Zu kompliziert

Wenn wir uns die Methode und insbesondere die Schleife und die dortigen `if`-Abfragen anschauen, ist das schon einigermaßen kompliziert. Bei genauem Hinsehen fällt auf, dass man Vereinfachungen erreichen kann: Die abgefragten Bedingungen sind semantisch gleich, aber negiert. Somit lassen sich die beiden `if`-Abfragen zusammenfassen, indem die `if`-Startbedingung in die `if`-Abfrage der `for`-Schleife integriert wird. Als Folge kann die Schleife bei 0 gestartet werden.

[1] `http://commons.apache.org/`

Auch die Formulierung der `for`-Schleife ist uns zu kompliziert und leicht unleserlich. Zur Vereinfachung entfernen wir die Zuweisung `n = value.length()` aus dem Initialisierungsteil der `for`-Schleife, die zur Optimierung des Vergleichs `i < n` diente.[2] Dies schreiben wir kürzer als `i < value.length()`. Das erhöht die Lesbarkeit und Verständlichkeit. Insgesamt ergibt sich folgende Korrektur:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    for (int i = 0; i < value.length(); i++)
    {
        if (!(Character.isDigit(value.charAt(i))))
        {
            return false;
        }
    }
    return true;
}
```

Wir führen die Unit Tests als NUMBERUTILSV4COMBINEDCONDITIONSTEST aus und diese bestätigen uns, dass alle bisher akzeptierten Zahlen weiterhin gültig sind.

Problem 4: Missverständlicher Name bzw. unklares Verhalten

Nach Rücksprache mit dem Kunden oder einem Requirements Engineer wird deutlich, dass selbstverständlich erwartet wird, dass die Methode `isNumber(String)` neben positiven auch negative Ganzzahlen verarbeiten kann. Eventuell wird später sogar eine Erweiterung auf Gleitkommazahlen gewünscht.

Hierbei gehen wir testgetrieben vor, d. h., wir erstellen zuerst mindestens einen Testfall. Dieser sollte fehlschlagen, da die dadurch geprüfte Funktionalität noch nicht existiert. Daraufhin korrigieren wir die Funktionalität. Gerade bei Sourcecode, den man nicht so gut kennt und in dem kleinere Erweiterungen oder Verbesserungen zu realisieren sind, ist dieses Vorgehen hilfreich. Nicht so empfehlenswert finde ich dieses Vorgehen allerdings, wenn man ganz genau weiß, was man realisieren möchte, und das Design und die Implementierung schon detailliert im Kopf hat. Dann können die Sprünge zwischen Test und Codierung stören. Das thematisiere ich nochmals in Kapitel 20.

Nach diesem kleinen gedanklichen Ausflug kommen wir wieder zu der Erweiterung zurück. Wir wollen die Verarbeitung von Vorzeichen prüfen und nutzen die Werte `"+4711"` und `"-4711"` als Eingaben, die in beiden Fällen das Ergebnis `true` liefern sollten, was wir folgendermaßen prüfen:

[2] Normalerweise ist das nicht notwendig, weil Optimierungen auf dieser feingranularen Ebene durch die JVM automatisch erfolgen (vgl. Kapitel 22).

```
@Test
public void testNumberPositive()
{
    assertTrue(NumberUtils.isNumber("+4711"), "plus sign should be accepted");
}

@Test
public void testNumberNegative()
{
    assertTrue(NumberUtils.isNumber("-4711"), "minus sign should be accepted");
}
```

Listing 17.3 *Ausführbar als* '**NUMBERUTILSV4SIGNTEST**'

Weil wir die Implementierung mittlerweile ziemlich gut kennen, wissen wir, dass keine Vorzeichen unterstützt werden. Wie erwartet, werden demnach beide Tests auch nicht bestanden, wie es auch ein Start des Programms NUMBERUTILSV4SIGNTEST zeigt. Aber durch das mittlerweile erworbene Know-how fällt die Korrektur der Methode nicht schwer – auch wenn der Umfang wieder zunimmt:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    // Verarbeite Vorzeichen
    String number = value;
    if (value.startsWith("-") || value.startsWith("+"))
    {
        number = value.substring(1, value.length());
    }
    // Weitere Prüfung auf Zahl wie zuvor
    for (int i = 0; i < number.length(); i++)
    {
        if (!(Character.isDigit(number.charAt(i))))
        {
            return false;
        }
    }
    return true;
}
```

Starten Sie den Test als Programm NUMBERUTILSV5SIGNTEST. Das zeigt, dass nun alle Tests bestanden werden. Allerdings hat die Komplexität des Applikationscodes schon wieder zugenommen. Spätestens jetzt sollten wir uns fragen, ob es nicht eine adäquate Funktionalität im JDK oder externen Bibliotheken gibt. Dadurch kann man sich in der Regel einige Arbeit sparen. Immerhin haben wir nun einen guten Satz an Unit Tests. Damit können wir dann auch gleich das letzte Problem der eingangs aufgestellten Mängelliste angehen. Schauen wir einmal, was möglich ist.

Problem 5: Viele `return`-Anweisungen

Die Methode `isNumber(String)` sollte auf Ganzzahlen prüfen. Statt diese Funktionalität, wie initial geschehen, selbst zu programmieren, bietet sich der Einsatz der Java-Bibliotheken zum Parsen von Zahlen an. Da unsere Tests bereits recht ausgereift sind, müssen wir hier nichts ergänzen. In der Methode `isNumber(String)` verwenden wir nun die Methode `Integer.parseInt(String)` (vgl. Abschnitt 5.2.2). Durch deren Einsatz können im Gegensatz zu der eigenen Realisierung sowohl positive als auch negative Zahlen verarbeitet werden.

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    try
    {
        Integer.parseInt(value); // Rückgabe ignorieren, nur prüfen
        return true;
    }
    catch (final NumberFormatException ex)
    {
        return false;
    }
}
```

Wir nutzen wieder unsere mittlerweile auf sieben Testfälle angewachsene Testklasse, um die korrekte Funktionalität zu prüfen. Alle Unit Tests werden bestanden, wie es ein Start des Programms NUMBERUTILSV6WITHJDKTEST bestätigt. Ziehen wir ein Fazit und schauen uns mögliche Erweiterungen an.

Was haben wir erreicht? Die Intention und Realisierung der Methode ist nun deutlich klarer, da nicht mehr auf Basis einzelner Zeichen geprüft wird. Vielmehr erfolgt auf logischer Ebene eine Umwandlung in eine Zahl. Die Lesbarkeit hat dadurch enorm zugenommen. Wir kommentieren zudem die bewusst fehlende Auswertung des Rückgabewerts von `Integer.parseInt(String)`, um möglicherweise aufkommende Fragen anderer Entwickler sofort zu klären.

Einschränkungen und mögliche Erweiterungen Die obige Realisierung ist deutlich klarer und besser verständlich als die Originalmethode sowie alle zuvor gezeigten Zwischenschritte.

Allerdings gibt es doch noch eine Kleinigkeit zu bedenken bzw. zu bemängeln: Durch die Prüfung mit `Integer.parseInt(String)` wird der Wertebereich auf `Integer.MIN_VALUE` bis `Integer.MAX_VALUE` begrenzt, also in etwa auf ± 2 Milliarden. Für größere Wertebereiche können wir auf die Wrapper-Klasse `Long` zurückgreifen. Um Gleitkommazahlen zu unterstützen, können wir die Klassen `Float` bzw. `Double` verwenden. All dies führt jedoch zu Veränderungen im nach außen sichtbaren Verhalten der Methode, weil dadurch weitere Eingabewerte erlaubt sind. Dabei müs-

sen wir aber die Anmerkungen im folgenden Hinweis »Vorsicht selbst bei minimalen Modifikationen am Verhalten« bezüglich der Änderung von Verhalten bedenken.

Hinweis: Vorsicht selbst bei minimalen Modifikationen am Verhalten

In einigen Schritten wurde durch die initiale Prüfung sowie den Einsatz von Bibliotheksfunktionen zum Parsing minimal etwas am nach außen veröffentlichten Verhalten verändert. So etwas übersieht man leicht. Das kann problematisch sein, weil Inkompatibilitäten verursacht werden können. Betrachten wir dies im Detail.

Auswirkungen der Korrekturen für Problem 1 Die ursprüngliche Methode hat bei Übergabe leerer Eingaben eine `StringIndexOutOfBoundsException` ausgelöst. Die neue Realisierung greift gar nicht erst indiziert zu, wenn der übergebene Text leer ist, sondern es wird direkt `false` zurückgegeben.

Auswirkungen der Korrekturen für Problem 4 und 5 Durch die Akzeptanz von Vorzeichen sowie den Einsatz von Bibliotheksfunktionen beim Parsing werden nun je nach gewählter Realisierung auch Vorzeichen und Nachkommastellen unterstützt. Die ursprüngliche Methode hat lediglich Ganzzahlen ohne Vorzeichen, dafür aber beliebiger Länge, als Zahl erkannt. Dies war höchstwahrscheinlich nicht als Feature gedacht, sondern war vermutlich eher ein »Unfall«.

Kompatibilität und Einsatz von Bibliotheksfunktionen

Die Kompatibilität zum ursprünglichen Verhalten kann entscheidend sein, wenn man nicht alle Aufrufer im Zugriff hat und somit nicht weiß, ob vielleicht einer von diesen die Information »keine Zahl« durch Abfangen einer `StringIndexOutOfBoundsException` ermittelt. Andere Aufrufer könnten beliebig lange Ziffernfolgen als Zahlen auswerten wollen.

Weil Ersteres meiner Ansicht nach ein Designfehler ist, werde ich hier keine Lösung angeben. Die Auswertung beliebig langer Ziffernfolgen ist ein denkbarer Anwendungsfall. Hätte die Aufgabe der ursprünglichen Methode `isNumber(String)` tatsächlich darin bestanden, nur eine derartige Prüfung durchzuführen, kann man dies elegant und kompatibel zu allen Unit Tests mit der Klasse `java.math.BigInteger` realisieren.

17.2 Das Standardvorgehen

Das einleitende Beispiel hat Ihnen einen ersten Eindruck von Refactoring-Schritten vermittelt. Um reproduzierbare Ergebnisse zu erreichen und mehr Sicherheit zu erhalten, sollten wir eine Art Checkliste zur Überarbeitung von Sourcecode befolgen. Diese beschreibt das bereits angesprochene Standardvorgehen, das folgende Schritte umfasst:

1. **Testen** – Führe vorhandene Unit Tests aus. Im Speziellen sollte dies kontinuierlich nach jedem der folgenden Schritte geschehen.
2. **Coding Conventions anwenden** – Beachte die später in Abschnitt 19.3 vorgestellten Coding Conventions:
   - Formatiere den Sourcecode.
   - Definiere, falls möglich, Übergabeparameter und lokale Variablen `final`.
   - Sorge durch das Basis-Refactoring RENAME (in Eclipse im Menü REFACTOR –> RENAME erreichbar) für verständliche Konstanten-, Variablen- und Methodennamen.
3. **In Einzelbestandteile zerlegen** – Zerlege, wenn notwendig und sinnvoll, eine komplexere Programmstelle zunächst mithilfe (des mehrmaligem Einsatzes) des Basis-Refactorings EXTRACT METHOD, das in Eclipse im Menü REFACTOR –> EXTRACT METHOD zu finden ist, in handhabbare und sinnvolle kleinere Bestandteile.
4. **Unit Tests erstellen** – Nutze, sofern die Anforderungen von der Fachseite oder dem Kunden oder aus einem Requirements-Dokument bekannt sind, diese, um daraus Testfälle zu gestalten. Ansonsten schreibe für zuvor extrahierte Methoden entsprechende Unit Tests:
   - Für normale Eingaben
   - Für ungültige Eingaben sowie für Rand- oder Spezialfälle
5. **Aufräumen** – Räume den Sourcecode auf:
   - Entferne unbenutzte Variablen und unbenutzte Methoden.
   - Entferne alte, unnötige oder (mittlerweile) falsche Kommentare.
6. **Vereinfachen** – Reduziere die Komplexität:
   - Entferne duplizierten Sourcecode, erzeuge gegebenenfalls Hilfsmethoden.
   - Vereinfache Bedingungen.
   - Füge bei Bedarf erklärende Kommentare ein – oftmals sollte man zunächst sprechende Bezeichner für Attribute und Methoden in Betracht ziehen.
7. **Konkrete Refactorings durchführen** – Verbessere den Sourcecode durch den Einsatz eines für den erkannten Schwachpunkt passenden Refactorings aus dem Refactoring-Katalog aus Abschnitt 17.4.

Die einzelnen Schritte müssen nicht sklavisch exakt in dieser Reihenfolge abgearbeitet werden. Es ist durchaus möglich, die Reihenfolge zu variieren und einige Schritte auszulassen oder möglicherweise sogar mehrfach auszuführen. Um ein Gespür für die Vorgehensweise beim Einsatz in der Praxis zu bekommen, wollen wir die obigen Schritte an einem weiteren kurzen Beispiel nachvollziehen.

Beachten Sie bitte, dass ich sowohl im folgenden Beispiel als auch bei der Vorstellung der Refactorings auf eine detaillierte Darstellung der notwendigen Unit Tests verzichten werde, um so den Fokus auf die eigentliche Transformation zu legen. In der Praxis entspricht die Vorgehensweise jedoch dem zuvor gezeigten einleitenden Beispiel, in dem Unit Tests zur Absicherung erstellt und eingesetzt wurden.

Beispiel

Als Ausgangsbasis dient folgender überarbeitungswürdiger `catch`-Block, den ich tatsächlich – abgesehen von Details – so in Produktionscode vorgefunden habe:

```
catch (final Fault aF)
{
    final String stSrc = aF.getSource();
    final String stErr = aF.getFaultCode();
    if (stErr != null && stErr.equalsIgnoreCase("FileNotFound"))
    {
        stErrorMsg = aF.getFaultString() + ": " + stSrc;
    }
    else
    {
        stErrorMsg = aF.getFaultString() + ": " + stSrc;
    }
}
```

Wir führen folgende Schritte (Nummerierung laut Standardvorgehen) durch:

Schritt 2: Coding Conventions anwenden Befolgt man Namenskonventionen, führt das zu mehr Lesbarkeit. Es fällt auf, dass in diesem Sourcecode-Abschnitt einige Präfixe in den Variablennamen verwendet werden. Wie später in Abschnitt 19.3.1 genauer beschrieben, stellen Präfixe in Variablennamen eher ein Relikt aus alten Tagen dar, weil damals die IDEs noch nicht in der Lage waren, Informationen aus dem Sourcecode on the fly zu extrahieren und etwa Attribute farblich anders darzustellen. Heutzutage sollte man Präfixe selten verwenden oder besser ganz vermeiden – ***beim Überarbeiten eines bestehenden Programmteils ist es aber ratsam, nicht alles umzukrempeln und dem vorhandenen Stil einigermaßen zu folgen***, sofern dies nicht (allzu sehr) gegen die vereinbarten Konventionen verstößt. Im Beispiel gilt, dass ein Präfix nutzlos ist, wenn kein sinnvoller Variablenname folgt wie hier für die Variable `aF`. Diese wird daher in `fault` umbenannt. Solche Namensänderungen sind für lokale Variablen immer möglich. Wir benennen die Variable `stErr` zunächst in `strErr` um. Aus `stSrc` wird `strSource`. Beide Male wird das merkwürdige Präfix `st` dabei zu `str` für Stringvariablen. Das Beibehalten der Kürzel ist ein Zugeständnis, nicht allzu viel am Stil zu ändern. Zudem wird `strErr` durch Umbenennung in `strFaultCode` besser lesbar

und verständlich. Die Namensähnlichkeit mit dem privaten Attribut `stErrorMsg` und die daraus resultierende Verwechselungsgefahr existiert nach dieser Namensänderung nicht mehr. Das Attribut wird außerdem durch den Einsatz der `this`-Notation explizit als solches gekennzeichnet und mit dem Präfix `str` zu `strErrorMsg`. Es lässt sich somit visuell gut von lokalen Variablen abgrenzen.[3]

```
catch (final Fault fault)
{
    final String strSource = fault.getSource();
    final String strFaultCode = fault.getFaultCode();
    if (strFaultCode != null && strFaultCode.equalsIgnoreCase("FileNotFound"))
    {
        this.strErrorMsg = fault.getFaultString() + ": " + strSource;
    }
    else
    {
        this.strErrorMsg = fault.getFaultString() + ": " + strSource;
    }
}
```

Nicht in jedem Fall ist eine Korrektur von Namen derart möglich. Private Attribute können problemlos umbenannt werden – sofern darauf nicht per Reflection zugegriffen wird. ***Im nachfolgenden Text dieses Kapitels gehe ich auf den Spezialfall Reflection und Refactorings kaum mehr explizit ein.*** Die Änderung der Namen öffentlicher Attribute kann Änderungen in einsetzenden Klassen erfordern und ist daher möglicherweise problematisch. Eine Verbesserung der Kapselung erreicht man mit dem Refactoring REDUZIERE DIE SICHTBARKEIT VON ATTRIBUTEN (vgl. Abschnitt 17.4.1), das auf dem Basis-Refactoring ENCAPSULATE FIELD basiert. Als Folge lässt sich bei Bedarf das Design ändern. Statt eines Attributs kann man Delegation nutzen oder den Wert dynamisch berechnen – das bietet sich etwa für die Eigenschaft Alter einer Person an, das sich aus dem aktuellen Datum und dem Geburtstag ergibt.

Schritt 6: Vereinfachen In diesem einfachen Beispiel findet man eine exakte Duplikation der Anweisungen im `if`- und `else`-Anweisungsblock. Zwar ist das realer Anwendungscode, aber meistens ist eine derart extreme Duplikation selten und eher als Wiederholung von Teilabschnitten zu beobachten. In diesem Beispiel kann eine enorme Vereinfachung erzielt werden, weil die Auswertung der Bedingung überflüssig ist. Der Anweisungsblock muss dadurch lediglich einmal notiert werden:

```
catch (final Fault fault)
{
    final String strSource = fault.getSource();
    final String strFaultCode = fault.getFaultCode();

    this.strErrorMsg = fault.getFaultString() + ": " + strSource;
}
```

[3] Allerdings besitzen moderne IDEs mittlerweile eine ausgefeilte farbliche Darstellung von Programmelementen, wodurch eine gute visuelle Trennung möglich wird, ohne dass man dazu `this` nutzen muss.

Schritt 5: Aufräumen Als Folge dieser Vereinfachungen wird ersichtlich, dass die Variable `strFaultCode` nun unbenutzt und damit überflüssig ist. Damit kann sie entfallen. Auch die Variable `strSource` ermittelt nur Daten aus dem übergebenen `Fault`-Objekt. Wir schreiben einfacher Folgendes:

```
catch (final Fault fault)
{
    this.strErrorMsg = fault.getFaultString() + ": " + fault.getSource();
}
```

Fazit

Das Resultat ist beeindruckend: Aus zehn Zeilen Sourcecode ist eine Zeile geworden. Ähnlich wie sich Bad Smells ausbreiten, wenn man unachtsam ist, gibt es glücklicherweise auch einen gegenteiligen Effekt: ***Je mehr man für Klarheit und Struktur sorgt, desto leichter fallen weitere Verbesserungsmaßnahmen.***

Jedoch haben wir ein Problem nicht entfernen können: Nach wie vor gibt es eine Zuweisung an das Attribut `strErrorMsg` im `catch`-Block. Ein solcher Seiteneffekt ist zu vermeiden, da es an unerwarteter Stelle im Programm zu Zustandsänderungen kommt. ***Durch Refactorings wollen wir normalerweise das nach außen sichtbare Programmverhalten nicht verändern, sondern lediglich die innere Struktur verbessern***. An dieser Stelle können wir im Sourcecode einen Kommentar mit einem Hinweis auf diesen Seiteneffekt einfügen und das Ganze später nochmals prüfen bzw. überarbeiten.

17.3 Kombination von Basis-Refactorings

Bevor wir uns einen Katalog einiger komplexerer Refactorings – und zum Teil genau genommen sogar leicht verhaltensverändernder Transformationen des Sourcecodes – anschauen, wollen wir zunächst an einem Beispiel verschiedene durch die IDE unterstützte Basis-Refactorings, wie sie Martin Fowler in seinem Buch »Refactoring: Improving the Design of Existing Code« [20] beschreibt, betrachten. Die Basis-Refactorings zeichnen sich dadurch aus, dass sie oftmals keine Modifikation am sichtbaren Verhalten vornehmen. Die hohe Kunst ist es, die Schritte so klein und sicher zu gestalten, dass es dabei möglichst selten zu Kompilierfehlern oder anderweitigen Problemen kommt. Damit es klappt, muss man sich der Refactoring-Automatiken aus der IDE bedienen. Die Wahrscheinlichkeit für Komplikationen steigt, wenn man eher freihändig refaktorisiert.

17.3.1 Refactoring-Beispiel: Ausgangslage und Ziel

Als Ausgangsbasis dient eine Utility-Klasse `TimeStampUtils` mit einer Methode `createTimeStampString()`. Schauen wir zunächst auf einen Aufruf eines Nutzers:

```
final String timeStamp = TimeStampUtils.createTimeStampString(currentPeriod,
                                                              frequency);
```

Weil der Aufruf unproblematisch scheint, betrachten wir nun die statische öffentliche Methode an sich, um mögliche Schwachpunkte zu erkennen:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                           final ComplexFrequency frequency)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    final int divisor = frequency == ComplexFrequency.P1M ? 1 : 3;
    final String addition = frequency == ComplexFrequency.P1M ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Ein erster Blick zeigt eine vermeintlich einfache Realisierung, die Jahresangaben gefolgt von Monat oder Quartal ausgeben soll.[4] Das Ganze ist recht kurz, aber vielleicht durch die Abfragen mit dem `?`-Operator ein wenig unübersichtlich. Problematischer ist jedoch, dass die Methode unerwünschte Abhängigkeiten auf die zwei Klassen `ExtTimePeriod` und `ComplexFrequency` besitzt, die aus einem externen Package (`external`) stammen. Ein genauerer Blick offenbart zusätzlich folgende Probleme:

- Die Methode scheint für beliebige Frequenzen des Typs `ComplexFrequency` ausgelegt zu sein. Tatsächlich ist sie es aber nicht, denn durch einen versteckten Logikfehler wird alles außer der Frequenz monatlich auf Quartale abgebildet. Dadurch können nur Monats- oder Quartalswerte korrekt verarbeitet werden.
- Es ist unklar, welches der gewünschte Rückgabewert ist. Gerade im Bereich von Datumsarithmetik findet man 0- oder 1-basierte Werte: Startet `getMonthValue()` also mit 0 oder 1? Und wieso erfolgt eine Subtraktion von 1?

Für die nachfolgenden Refactorings steht zunächst die Auflösung der Abhängigkeiten im Fokus. Auf die beiden anderen Details der Verarbeitung gehe ich später ein.

Definition des Ziels

Die Methode `createTimeStampString(ExtTimePeriod, ComplexFrequency)` soll nun mithilfe von Basis-Refactorings so umgestaltet werden, dass nur Abhängigkeiten auf Standards wie Klassen aus öffentlichen Bibliotheken oder besser noch dem JDK bestehen und der Sourcecode verständlicher wird. Bevor wir mit den Umbauarbeiten beginnen, erstellen wir ein UML-Klassendiagramm von der Ausgangslage und insbesondere auch von einem möglichen Zieldesign. Beides ist in Abbildung 17-1 dargestellt. Das gezeigte Ziel ist nicht ganz starr, sondern eher ein Anhaltspunkt, da man beim Entwickeln gewöhnlich immer noch kleinere Änderungen vornimmt. Das ist auch der Grund, warum wir hier keine Typparameter in den Signaturen angeben.

[4] Dabei wird auf `null`-Prüfungen verzichtet, weil es sich um eine interne Hilfsklasse handelt und wir uns hier auf die Refactoring-Schritte konzentrieren wollen.

Abbildung 17-1 *Refactoring der Methode* `createTimeStampString()`

Wir wollen nun folgende Schritte ausführen, um die angemerkten Probleme zu beseitigen und das dargestellte Ziel zu erreichen:

- **Auflösen der Abhängigkeiten** – In einem ersten Schritt lösen wir die Abhängigkeiten zum Package `external` auf, indem wir einen `enum` namens `SupportedFrequencies` als Ersatz für `ComplexFrequency` einführen und anstelle der Klasse `ExtTimePeriod` die Klasse `LocalDateTime` aus dem Date and Time API (vgl. Kapitel 8) verwenden.
- **Vereinfachungen** – Einige der Berechnungen in der Methode sind etwas komplex und nicht gut zu lesen. Wir werden ein paar Vereinfachungen vornehmen.
- **Verlagern von Funktionalität** – Abschließend schauen wir, wie wir durch eine kleine Änderung von Zuständigkeiten für mehr Klarheit im Design sorgen.

17.3.2 Auflösen der Abhängigkeiten

Um den Sourcecode klarer und besser verständlich zu gestalten, werden wir folgende Refactorings (mit Windows-Tastaturkürzeln in Eclipse in Klammern) nutzen:[5]

- EXTRACT LOCAL VARIABLE (ALT+SHIFT+L)
- EXTRACT METHOD (ALT+SHIFT+M)
- INLINE (ALT+SHIFT+I)
- CHANGE METHOD SIGNATURE (ALT+SHIFT+C)

[5] Für Mac OS ist es statt ALT+SHIFT die Kombination ALT+COMMAND. Oftmals bietet die in Eclipse integrierte QUICK-FIX-Funktionalität, die man durch CTRL+1 aufruft, die Möglichkeit, die ersten drei Refactorings direkt auszuführen.

Schritt 1: Hilfsvariable einführen (EXTRACT LOCAL VARIABLE)

Zum leichteren Verständnis wird die zu bearbeitende Methode nochmals gezeigt:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                           final ComplexFrequency frequency)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    final int divisor = frequency == ComplexFrequency.P1M ? 1 : 3;
    final String addition = frequency == ComplexFrequency.P1M ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Als Erstes selektieren wir den Ausdruck `ComplexFrequency.P1M` und nutzen das Refactoring EXTRACT LOCAL VARIABLE (ALT+SHIFT+L), um die lokale Variable `isMonthly` zu extrahieren, wodurch sich der Sourcecode vereinfachen lässt:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                           final ComplexFrequency frequency)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Schritt 2: Abhängigkeit zur Frequenz entfernen (EXTRACT METHOD)

Als Nächstes wollen wir die Abhängigkeit zur Klasse `ComplexFrequency` auflösen und dazu eine überladene Variante der Methode `createTimeStampString()` erzeugen. Um diese aus der Originalmethode zu extrahieren, ordnen wir einige Zeilen um und nutzen dazu die Tastaturkürzel ALT+UP/DOWN. Damit verschieben wir die boolesche Variable `start` direkt zu der ersten Verwendung, also vor die Definition von `value`. Die Variable `isMonthly` schieben wir an den Methodenanfang:[6]

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                           final ComplexFrequency frequency)
{
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final LocalDateTime start = currentPeriod.getDateTime();
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

[6] Bitte beachten Sie, dass diese Umordnung hier zu keiner Verhaltensänderung führt, aber dass das in der Praxis z. B. wegen möglicherweise versteckter Seiteneffekte nicht immer so ist. Man spricht in dem Zusammenhang von ***Temporal Coupling*** (vgl. Abschnitt 17.6).

Danach selektieren wir alle Zeilen nach der Definition von `isMonthly` und setzen das Refactoring EXTRACT METHOD (ALT+SHIFT+M) ein. Damit entsteht eine gleichnamige Methode mit der Sichtbarkeit `public`, die als Parametertyp `boolean` statt `ComplexFrequency` besitzt.

```
@Deprecated
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                           final ComplexFrequency frequency)
{
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    return createTimeStampString(currentPeriod, isMonthly);
}

public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                           final boolean isMonthly)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final LocalDateTime start = currentPeriod.getDateTime();
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Die unerwünschte Abhängigkeit zur Klasse `ComplexFrequency` wurde damit aufgelöst – allerdings durch einen booleschen Parameter, was meistens kein gutes Design ist. Später komme ich darauf zurück und wir beheben auch diese Schwachstelle.

Indem wir die ursprüngliche Methode als `@Deprecated` markieren, signalisieren wir, dass zukünftige Nutzer stattdessen die neu erstellte Methode verwenden sollten. Für die bisherigen Aufrufer hat sich nichts geändert.

Schritt 3: Inlining des Methodenaufrufs (INLINE)

Da wir die Abhängigkeit zur Klasse `ComplexFrequency` in der Utility-Klasse eliminieren wollen, sollte überall die neu erstellte statt der alten Methode eingesetzt werden.

Die Aufrufstellen sind ähnlich zu folgender:

```
final String timeStamp = createTimeStampString(currentPeriod, frequency);
```

Die Aufrufstellen könnten wir zwar von Hand korrigieren, aber es ist sinnvoller und weniger fehleranfällig, dazu die in die IDE integrierten Refactorings zu nutzen.[7] Dazu markieren wir den Namen der ursprünglichen Methode und nutzen dann das Refactoring INLINE (ALT+SHIFT+I). Dieses transformiert alle Aufrufstellen folgendermaßen:

```
final boolean isMonthly = frequency == ComplexFrequency.P1M;
final String timeStamp = createTimeStampString(currentPeriod, isMonthly);
```

Optional kann die nicht mehr benötigte Methode automatisch gelöscht werden, wodurch nur noch die zuvor extrahierte, neue Methode in der Utility-Klasse verbleibt.

[7] Allerdings sollten wir uns dabei bewusst sein, dass sich die Verarbeitungsreihenfolge durch das INLINE ändern kann. Auch hier kann also Temporal Coupling eine Rolle spielen.

Die bisher durchgeführten Änderungen lassen erahnen, dass sich für Umgestaltungen die Nutzung von Refactoring-Automatiken anbietet, um die Wahrscheinlichkeit für Fehler zu reduzieren und für konsistente Änderungen zu sorgen.

Schritt 3a (optional): Inlining der Hilfsvariablen (INLINE)

Die Aufrufstellen sehen nun nach Schritt 3 – unter anderem durch den booleschen Übergabeparameter – etwas ungelenk aus, was wir später noch mit einer Designänderung adressieren werden. Zunächst könnte man in einem weiteren Schritt statt der lokalen Variablen den Ausdruck direkt als Methodenparameter angeben. Dabei hilft wiederum das Refactoring INLINE (ALT+SHIFT+I), diesmal für die Variablendeklaration. Zum Ausführen ist die Variable `isMonthly` zu selektieren:

```
final boolean isMonthly = frequency == ComplexFrequency.P1M;
final String timeStamp = createTimeStampString(currentPeriod, isMonthly);
```

Durch das Refactoring INLINE wird der Aufruf folgendermaßen abgewandelt:

```
final String timeStamp = createTimeStampString(currentPeriod,
                                               frequency == MyFrequency.P1M);
```

Jedoch reduziert dieser Schritt mitunter die Verständlichkeit und Lesbarkeit.

Schritt 4: Abhängigkeit zur Klasse `ExtTimePeriod` entfernen

Kommen wir wieder zu der eigentlichen Methode `createTimeStampString()` zurück. Wir wollen nun die Abhängigkeiten auf die Klasse `ExtTimePeriod` auflösen. Dabei hilft uns ein scharfer Blick auf die Methode und das Refactoring EXTRACT METHOD: Abgesehen von der ersten Zeile der Methode selektieren wir den Rest und extrahieren eine gleichnamige Methode. Die Utility-Klasse sieht wie folgt aus, nachdem wir die alte Methode noch als `@Deprecated` markiert haben:

```
@Deprecated
public static String createTimeStampString(final MyTimePeriod currentPeriod,
                                           final boolean isMonthly)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    return createTimeStampString(isMonthly, start);
}

public static String createTimeStampString(final boolean isMonthly,
                                           final LocalDateTime start)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Wie schon zuvor, hat das Refactoring keine Auswirkungen auf bisherige Nutzer, außer, dass diese nun durch das Hinzufügen von `@Deprecated` auf unsere Änderungen in der Utility-Klasse aufmerksam gemacht werden.

Schritt 5: Für konsistente Parameterreihenfolge sorgen

Beim Extrahieren der Methode fällt uns auf, dass durch die Automatik die Parameterreihenfolge vertauscht wurde und `isMonthly` nun der erste Parameter ist. Das wollen wir korrigieren. Dazu nutzen wir das Refactoring CHANGE METHOD SIGNATURE (ALT+SHIFT+C) und vertauschen die beiden Parameter, womit wir wieder eine konsistente Reihenfolge erzielen.

Schritte 6: Inlining des Methodenaufrufs

Wir führen weitere Aufräumarbeiten aus und selektieren die alte Methode und nutzen das Refactoring INLINE. Dadurch verdichten wir die Utility-Klasse:

```
public static String createTimeStampString(final LocalDateTime start,
                                           final boolean isMonthly)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Auch die Aufrufstelle wird automatisch durch die IDE angepasst:

```
final LocalDateTime start = currentPeriod.getDateTime();
final String timeStamp = createTimeStampString(start,
                                               frequency == EasyFrequency.P1M);
```

Schritte 6a (optional): Inlining der Hilfsvariablen

Wenn gewünscht, kann man nochmals das Refactoring INLINE ausführen, um die Hilfsvariable zu entfernen und direkt in den Methodenaufruf zu integrieren:

```
final String timeStamp = createTimeStampString(currentPeriod.getDateTime(),
                                               frequency == EasyFrequency.P1M);
```

Zwischenfazit

Die erstellte Methode besitzt keine Abhängigkeiten auf externe Klassen mehr oder zumindest nur auf Klassen, die Standardbibliotheken des JDKs entstammen. Damit lässt sich das Ganze viel einfacher mit Unit Tests überprüfen. Bisher haben wir allerdings keine Unit Tests ausgeführt, insbesondere weil es schlicht keine gab, was wir nun ändern wollen, und weil wir lediglich sichere Basis-Refactorings angewendet haben.

Tests

Die zuvor genutzten Refactorings haben das nach außen sichtbare Verhalten nicht geändert, was aber selbst für die Basis-Refactorings nicht immer gilt.[8] Allerdings sind diese deutlich sicherer, als die Refactorings von Hand auszuführen.

Bei Änderungen empfiehlt sich generell die Ausführung von Unit Tests. Weil es noch keine gibt, erstellen wir nachfolgend exemplarisch zwei einfache Testfälle:

```
@Test
void testCreateTimeStampString_Monthly()
{
    final boolean MONTHLY = true;
    assertEquals("2000-2", createTimeStampString(
                           LocalDateTime.of(2000, 2, 7, 0, 0), MONTHLY));
    assertEquals("2000-7", createTimeStampString(
                           LocalDateTime.of(2000, 7, 14, 0, 0), MONTHLY));
}

@Test
void testCreateTimeStampString_Quarterly()
{
    final boolean QUARTERLY = false;
    assertEquals("2000-Q1", createTimeStampString(
                            LocalDateTime.of(2000, 2, 7, 0, 0), QUARTERLY));
    assertEquals("2000-Q3", createTimeStampString(
                            LocalDateTime.of(2000, 7, 14, 0, 0), QUARTERLY));
}
```

Listing 17.4 *Ausführbar als* **'TIMESTAMPUTILSTEST'**

Wir führen die Tests als Programm TIMESTAMPUTILSTEST aus und sie zeigen – wie erwartet – Grün. Normalerweise würden wir noch ein paar mehr Testfälle ergänzen. In diesem Kontext sollen uns aber diese zwei reichen, um mögliche Probleme aufzuzeigen. Allerdings sollten Klassen in der Praxis umfangreicher getestet werden, als es der Platz hier erlaubt.

Unzulänglichkeit: Boolescher Parameter

Durch die Refactoring-Schritte haben wir zwar die Abhängigkeiten zum Package `external` gelöst, jedoch – wie schon zuvor erwähnt – auch eine Unschönheit in unsere öffentliche Schnittstelle eingefügt: einen booleschen Parameter. Was ist daran störend? Aufrufer müssen dadurch immer genau wissen, was die Werte `true` bzw. `false` ausdrücken sollen. Das lässt sich nur durch Betrachten der Implementierung der Methode ermitteln – leider nicht nur anhand der Aufrufstelle.

[8] Insbesondere gilt dies für das Refactoring CHANGE METHOD SIGNATURE, um Parameter umzuordnen, deren Typ zu ändern oder neue Parameter einzufügen, wodurch sich schnell Verhalten ändert. Ebenso kann ein Inlining problematisch sein, weil dadurch die Abarbeitungsreihenfolge leicht geändert wird. Man spricht bei dem Problem auch von Temporal Coupling (vgl. Abschnitt 17.6).

Die Verwendung der booleschen Konstanten `MONTHLY` und `QUARTERLY` in den Tests macht deutlich, dass sich ein weiteres Refactoring anbietet, um den booleschen Parameter in der öffentlichen Schnittstelle zu eliminieren. In unserem Beispiel hatten wir die Klassen im Zugriff und durften dort auch ändern. Das ist jedoch nicht immer der Fall, sodass man mitunter mit der Signatur leben muss. Wie kann man trotzdem für besser verständlichen Sourcecode sorgen? Schauen wir uns verschiedene Abhilfen an.

Abhilfen ohne Änderungen der Signatur Wie wir es beim Erstellen der Tests kennengelernt haben, kann man zwei Konstanten mit sprechenden Namen definieren:

```
public static final boolean MONTHLY = true;
public static final boolean QUARTERLY = false;
```

Oftmals besser lesbar ist es, eine Hilfsmethode wie folgt zu implementieren:

```
private static boolean isMonthly(final ComplexFrequency frequency)
{
    return frequency == ComplexFrequency.P1M;
}
```

Bei der zweiten Variante verbleibt allerdings die Abhängigkeit von Aufrufern an das Package `external`, was aber möglicherweise akzeptabel ist.

Es gibt eine weitere Möglichkeit, die so elegant in der Nutzung und offensichtlich ist, dass man sie leicht übersieht: Man definiert zwei Methoden mit sprechendem Namen, die jeweils den erwarteten Wert zurückgeben:[9]

```
private static boolean monthly()
{
    return true;
}

private static boolean quarterly()
{
    return false;
}
```

Die gezeigten Varianten lösen das eigentliche Problem nicht, lindern jedoch ein wenig die »API-Schmerzen«. Das ist für die Fälle praktisch, in denen man die Schnittstelle der Klasse nicht ändern kann, die Aufrufe aber klarer gestalten möchte. Der Aufruf für monatlich würde wie folgt aussehen:

```
final String timeStamp = createTimeStampString(currentPeriod.getDateTime(),
                                               monthly());
```

[9] Das kann man ganz hervorragend auch für andere Rückgabetypen außer `boolean` machen, solange die Wertemenge nicht zu groß wird.

Abhilfen mit Änderungen der Signatur Wenn man auf die Klassen Zugriff hat und die Methode ändern kann, bietet sich die Definition eines `enum`s an:

```
public enum SupportedFrequencies
{
    MONTHLY, QUARTERLY;
}
```

Damit können wir die Signatur der Methode wie folgt abändern und eine Hilfsvariable `isMonthly` einführen:

```
static String createTimeStampString(final LocalDateTime start,
                                    final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Diese Lösung ist für Aufrufer klarer, was ein sehr wichtiger Punkt ist, da damit auch die Benutzbarkeit verbessert wird. Allerdings fängt das Ganze intern an, unübersichtlich zu werden. Es wird höchste Zeit, nach ein paar Vereinfachungen Ausschau zu halten.

17.3.3 Vereinfachungen

In diesem Abschnitt sehen wir uns zwei Arten von Vereinfachungen an. Zunächst entzerren wir die Anweisungen und die etwas komplexere Logik. Daraus ergeben sich weitere Möglichkeiten, die Formel zur Berechnung an sich zu vereinfachen.

Vereinfachung der Anweisungen

Die Komplexität innerhalb der Methode entsteht vor allem dadurch, dass hier die zwei Fälle »Monatlich« und »Quartalsweise« ineinander verwoben behandelt werden:[10]

```
public static String createTimeStampString(final LocalDateTime start,
                                           final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Teilen wir das Ganze doch einfach so auf, dass wir abhängig von `isMonthly` zwei Wertebelegungen erhalten. Versuchen wir schrittweise dorthin zu kommen.

Als Vorbereitung führen wir das Refactoring INLINE für die Variable `value` aus, um den Ausdruck zur String- und Wertekonkatenation zusammenzuführen.

[10] Potenziell eine Verletzung des Single Responsibility Principle (SRP) (vgl. Abschnitt 3.5.3).

Den ternären Operator (?-Operator) kann man in eine `if-else`-Anweisung umwandeln. Dazu nutzen wir das Tastaturkürzel CTRL+1 für QUICK FIX und wählen dort REPLACE CONDITIONAL WITH 'IF-ELSE'. Das machen wir für beide ?-Operatoren. Damit ergibt sich folgende Variante der ursprünglichen Methode, wobei die entstehenden `if-else`-Anweisungen allerdings (noch) keine Blöcke sind:

```
public static String createTimeStampString(final LocalDateTime start,
                                           final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    if (isMonthly)
        divisor = 1;
    else
        divisor = 3;
    final String addition;
    if (isMonthly)
        addition = "";
    else
        addition = "Q";

    return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
         / divisor + 1);
}
```

Das Ergebnis sieht ein wenig chaotisch aus und scheint in die falsche Richtung zu gehen, da viel mehr Zeilen entstanden sind. Lassen Sie sich nicht entmutigen. Die Tests zeigen, dass sich das Verhalten der Methode nicht geändert hat. Wir sind wohl auf dem richtigen Weg. Insbesondere sind die einzelnen Abfragen viel weniger komplex.

Jetzt wollen wir die jeweiligen Zeilen für die beiden Bedingungen zusammen gruppieren. Voraussetzung dazu ist aber, dass wir die `if-else`-Anweisungen in Blöcke umwandeln. Dazu selektieren wir das `if` und nutzen wiederum das Tastaturkürzel CTRL+1, was uns nun die Option CHANGE 'IF-ELSE' STATEMENTS TO BLOCKS anbietet, die wir wählen. Danach ordnen wir die Zeilen um, indem wir die Zeilen aus den jeweiligen Bedingungen gruppieren. Als Folge entsteht im unteren Teil ein leeres `if-else`-Gebilde, das wir entfernen. Es ergibt sich folgende Methode:

```
public static String createTimeStampString(final LocalDateTime start,
                                           final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    final String addition;
    if (isMonthly)
    {
        divisor = 1;
        addition = "";
    }
    else
    {
        divisor = 3;
        addition = "Q";
    }
    return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
               / divisor + 1);
}
```

Der Sourcecode ist erneut länger geworden, aber zumindest sind die logischen Einheiten gruppiert. Bevor wir vereinfachen können, wird es noch etwas unübersichtlicher und wir benötigen auch etwas Handarbeit, um die inverse Variante des Basis-Refactorings CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENT[11] durchzuführen. Normalerweise will man damit duplizierte Elemente in `if`-Zweigen zu einer Anweisung am Ende zusammenfügen. Hier machen wir das Gegenteil und duplizieren die `return`-Anweisung, um sie dann in jedem `if-else`-Zweig bereitzustellen:

```
public static String createTimeStampString(final LocalDateTime start,
                                           final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    final String addition;
    if (isMonthly)
    {
        divisor = 1;
        addition = "";
        return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
               / divisor + 1);
    }
    else
    {
        divisor = 3;
        addition = "Q";
        return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
               / divisor + 1);
    }
}
```

Die Variablen `divisor` und `addition` sind eigentlich überflüssig, da sie jeweils nur einfache Konstanten enthalten. Wir können nun die Werte für beide Variablen direkt im Sourcecode ersetzen. Je nach verwendeter IDE müssen wir etwas Handarbeit leisten, um die in den Zweigen jeweils konstanten Werte in die `return`-Anweisungszeile zu integrieren sowie die überflüssigen Variablendeklarationen zu entfernen:

```
public static String createTimeStampString(final LocalDateTime start,
                                           final SupportedFrequencies frequency)
{
    if (frequency == SupportedFrequencies.MONTHLY)
    {
        return start.getYear() + "-" + ((start.getMonthValue() - 1) / 1 + 1);
    }

    return start.getYear() + "-Q" + ((start.getMonthValue() - 1) / 3 + 1);
}
```

[11] Details finden Sie in Martin Fowlers Buch »Refactoring: Improving the Design of Existing Code« [20].

Hinweis: Viele Zwischenschritte

Diese vielen kleinen Schritte für diesen einfachen Sourcecode-Abschnitt sehen möglicherweise übertrieben aus. Allerdings bietet das den Vorteil, dass man sich in jedem Einzelschritt auf das Wesentliche konzentrieren kann. Wenn die Programmabschnitte komplexer werden, dann profitiert man von kleinen Schritten, die im Falle eines Irrwegs leicht zurückgenommen werden können. Flüchtigkeitsfehler lassen sich dadurch eher vermeiden als bei »Freihand«-Refactorings.

Vereinfachung der Berechnung in mehreren Schritten

Wenn man sich die Ausdrücke anschaut, sollte zumindest im ersten Fall eine Vereinfachung möglich sein. Damit wir nicht abgelenkt werden, schauen wir hier wirklich nur auf den Ausdruck der Monatsberechnung an sich, wobei die äußere Klammerung wegen der Stringkonkatenation nicht weiter betrachtet wird:

```
(start.getMonthValue() - 1) / 1 + 1
```

Die Division / 1 ist nutzlos Eine Division durch 1 ändert das Ergebnis nicht, macht aber den Ausdruck komplizierter und den Sourcecode schlechter lesbar. Also entfernen wir diese Division und erhalten folgende Vereinfachung:

```
(start.getMonthValue() - 1) + 1
```

Weitere Schritte 1 Aufgrund der Vereinfachung ergeben sich neue Möglichkeiten. In der Praxis sieht man es immer mal wieder, dass Ausdrücke eher zu viel geklammert sind. Hier ist die äußere Klammerung um die Subtraktion überflüssig und wird entfernt:

```
start.getMonthValue() - 1 + 1
```

Weitere Schritte 2 In einem letzten Schritt kann die Berechnung `- 1 + 1` entfallen, weil sie den Wert `0` ergibt und hier somit nutzlos ist. Damit verbleibt für die Berechnung der Monate nur noch der Aufruf `start.getMonthValue()` und wir können die Methode wie folgt vereinfachen:

```
public static String createTimeStampString(final LocalDateTime start,
                                           final SupportedFrequencies frequency)
{
    if (frequency == SupportedFrequencies.MONTHLY)
    {
        return start.getYear() + "-" + start.getMonthValue();
    }

    return start.getYear() + "-Q" + ((start.getMonthValue() - 1) / 3 + 1);
}
```

Komplexität der Berechnung für Quartale Die Berechnung des Quartals ist komplexer, was sich kaum vermeiden lässt. In der hier genutzten Klasse `LocalDateTime` beginnt die Zählung der Tage und Monate jeweils bei 1, wie es der menschlichen Denkweise entspricht. Die gezeigten Berechnungen für das Quartal bilden die Werte von 1 – 12 durch die Subtraktion von 1 auf 0 – 11 ab, wodurch die Division durch 3 einen Wertebereich von 0 – 3 liefert. Die Addition von 1 ergibt dann den Wertebereich 1 – 4. Allerdings liest sich das `/ 3 + 1` potenziell falsch. Um die Berechnung klarer zu gestalten, kann man die Addition von `1` nach vorne ziehen:

```
return start.getYear() + "-Q" + (1 + (start.getMonthValue() - 1) / 3);
```

Fazit

Wenn man bedenkt, mit welch kompliziertem Ausdruck wir ins Rennen gestartet sind, ist das Erreichte beeindruckend. Bei einem Blick auf die ursprüngliche Berechnung wäre weder eine Aussage möglich gewesen, was das Resultat denn nun genau ist, noch, ob man die Berechnung vereinfachen kann. Durch unseren letzten Schritt ist keine Vereinfachung für die Monatsberechnung mehr nötig und das Ergebnis offensichtlich. Nur die Quartalsberechnung ist eben etwas komplizierter.

17.3.4 Verlagern von Funktionalität

Betrachten wir die letzte Version der Methode `createTimeStampString(LocalDateTime, SupportedFrequencies)` einmal genauer, dann erkennen wir, dass sie im `if-else` jeweils Funktionalität enthält, die stark mit der Aufzählung `SupportedFrequencies` verbunden ist. Diese haben wir beim Erstellen der Unit Tests zur Verbesserung des leicht unhandlichen APIs definiert. Es bietet sich nun an, noch mehr Funktionalität dorthin zu verlagern. Dabei nutzen wir, dass eine `enum`-Aufzählung Attribute und Methoden besitzen kann und Letztere sogar überschrieben werden können:

```
public enum SupportedFrequencies
{
   MONTHLY
   {
      public String createTimeStampString(final LocalDateTime start)
      {
         return start.getYear() + "-" + start.getMonthValue();
      }
   },

   QUARTERLY
   {
      public String createTimeStampString(final LocalDateTime start)
      {
         return start.getYear() + "-Q" + (1 + (start.getMonthValue() - 1) / 3);
      }
   };

   public abstract String createTimeStampString(final LocalDateTime start);
}
```

Die Utility-Klasse enthält nur noch folgende Methode:

```
public static String createTimeStampString(final LocalDateTime start,
                                           final SupportedFrequencies frequency)
{
    return frequency.createTimeStampString(start);
}
```

Tatsächlich sprechen nur noch Rückwärtskompatibilitätsgründe dafür, die Utility-Klasse überhaupt noch beizubehalten. Ansonsten könnte man die Funktionalität durch direkte Aufrufe an die jeweilige `enum`-Konstante realisieren.

17.4 Der Refactoring-Katalog

Dieser Abschnitt stellt einige Refactorings sowie Transformationen als Schritt-für-Schritt-Anleitungen vor. Diese sollen dabei helfen, ein spezielles Problem auf eine definierte Art und Weise zu beheben. Einige Refactorings lassen sich zu »High-Level-Refactorings« kombinieren. Ein Beispiel dafür ist die als MINIMIERE ZUSTANDSÄNDERUNGEN beschriebene Kombination aus Abschnitt 17.4.5.

Im folgenden Text spreche ich häufig der Einfachheit halber von `get()`- und `set()`-Methoden. Diese müssen nicht immer mit einem solchen Präfix anfangen, sondern es sind ganz allgemein Accessor- und Mutator-Methoden gemeint.

17.4.1 Reduziere die Sichtbarkeit von Attributen

Bekanntlich stellen Attribute den internen Zustand eines Objekts dar. Dieser sollte von außen nicht direkt sichtbar oder sogar änderbar sein. Dem OO-Grundgedanken der Datenkapselung (vgl. Abschnitt 3.1) folgend, ist es das Ziel bei diesem Refactoring, eine direkte Änderbarkeit von Attributen auf ein Minimum zu reduzieren. Im Idealfall können Details der Implementierung ohne Rückwirkungen auf Nutzer modifiziert werden. Beispielsweise können Daten entweder als Attribut gehalten, bei Bedarf berechnet oder aus einer externen Quelle gelesen werden. Als Voraussetzung sollte die Sichtbarkeit von Attributen möglichst weit eingeschränkt werden.

Schauen wir dazu auf die in Abbildung 17-2 als Klassendiagramm visualisierte Klasse `Person`, die die zwei Attribute `name` und `age` besitzt und die Ausgangsbasis für dieses Refactoring darstellt. Die Attribute sind zu Demonstrationszwecken `public` ('+') und `protected` ('#').

Abbildung 17-2 *Ausgangszustand*

Nachfolgend wird ein `Person`-Objekt erzeugt und direkt auf dessen beide Attribute zugegriffen:

```
final Person person = new Person();
person.name = "Meyer";
person.age = 27;

System.out.println("Name='" + person.name + "', Age=" + person.age);
```

Hier findet demnach keine Datenkapselung statt. Um eine solche zu erreichen, sollten möglichst viele Attribute `private` deklariert und Zugriff über Methoden bereitgestellt werden. Dazu prüfen wir iterativ für jedes Attribut, ob dies möglich ist, indem wir die im Folgenden beschriebenen Schritte ausführen.

Schritt 1: Erzeugen von Zugriffsmethoden

Wir erzeugen Zugriffsmethoden für die Attribute. In Eclipse können wir dazu das Refactoring ENCAPSULATE FIELD aus dem Menü REFACTOR nutzen. Beginnen wir mit dem Attribut `name`. In diesem Fall entstehen die öffentlichen Methoden `getName()` und `setName(String)`, wie es Abbildung 17-3 zeigt.

Abbildung 17-3 *Zwischenstand nach dem Einführen der Methoden*

Automatische Korrekturen der Verwendungsstellen Das Refactoring der IDE passt praktischerweise auch gleich die Aufrufstellen an: Alle Stellen, die bisher lesend auf das Attribut zugegriffen haben, nutzen nun die zuvor erzeugte `getName()`-Methode. Alle schreibenden Zugriffe werden mit der `setName(String)`-Methode ersetzt. Der Sourcecode verändert sich wie folgt:

```
final Person person = new Person();
person.setName("Meyer");
person.age = 27;

System.out.println("Name='" + person.getName() + "', Age=" + person.age);
```

Automatische Anpassung der Sichtbarkeit Ebenso wie die Zugriffe wird auch die Sichtbarkeit des Attributs `name` auf `private` reduziert, wenn man das Refactoring der IDE nutzt. Das Ergebnis zeigt Abbildung 17-4.

Abbildung 17-4 *Ergebnis von Schritt 1*

Schritt 2: Wiederhole Schritt 1 für alle Attribute

Wiederhole das Vorgehen aus Schritt 1 für alle Attribute. In diesem Beispiel ist dies lediglich das Attribut `age`. Dadurch entstehen die zwei neuen Zugriffsmethoden `getAge()` und `setAge(int)` mit der Sichtbarkeit `protected`. Deren Einsatz führt zu folgenden Änderungen in nutzendem Sourcecode:

```
final Person person = new Person();
person.setName("Meyer");
person.setAge(27);

System.out.println("Name='" + person.getName() + "', Age=" + person.getAge());
```

Fazit

Diese scheinbar rein kosmetischen Änderungen bewirken eine Verbesserung in der Datenkapselung. Diese erlaubt es, die Attribute zur Speicherung des Namens oder des Alters, wenn nötig, zu verändern. Der Vorteil einer guten Datenkapselung ist, dass sich interne Änderungen nicht in der Schnittstelle für andere Klassen auswirken und man dadurch Folgeänderungen bei Klienten vermeidet. ***Eine bessere Kapselung führt demnach auch zu einer loseren Kopplung***. Wieso ist das gut? Ganz deutlich ist dies für das Attribut `age` erkennbar: Die eingangs gewählte Modellierung des Alters als veränderliches `int`-Attribut ist ungünstig, da man das Alter nicht ändern kann. Vielmehr ergibt es sich aus dem Geburtstag und dem aktuellen Datum. Eine Verbesserung des Designs erreicht man, indem man das Geburtsdatum als Attribut speichert und das Alter dynamisch berechnet, statt dieses in Form eines Attributs zu halten. Dadurch entfällt auch die Methode `setAge(int)`. Stattdessen kann das Geburtsdatum gesetzt werden. Dazu bietet sich der Einsatz der Klasse `LocalDate` aus dem Date and Time API an (vgl. Kapitel 8) an. Abbildung 17-5 zeigt das Klassendiagramm.

Abbildung 17-5 *Mögliche Designänderungen durch Kapselung*

Sinnvolle weitere Schritte

Die neu entstandenen Zugriffsmethoden sollten in ihrer Sichtbarkeit in einem nachfolgenden Refactoring so weit wie möglich eingeschränkt werden. Das Refactoring REDUZIERE DIE SICHTBARKEIT VON METHODEN stellt in Abschnitt 17.4.3 die Anleitung dazu vor. Häufig können Attribute zudem `final` gemacht werden. Dazu müssen die `set()`-Methoden durch Initialisierungen zum Konstruktionszeitpunkt ersetzt und der Konstruktor um Parameter erweitert werden. Dies beschreibt das Refactoring MINIMIERE VERÄNDERLICHE ATTRIBUTE, das ich nun vorstelle.

17.4.2 Minimiere veränderliche Attribute

Wünschenswert ist es, die Zustandsänderungen eines Objekts möglichst gut zu kontrollieren und auf ein tatsächlich notwendiges Maß zu beschränken, um unerwünschte Objektzustände zu vermeiden. Dieses Ziel erreicht man mithilfe dieses Refactorings. Bestenfalls entsteht eine Klasse, die unveränderlich ist.

Die Ausgangssituation in der Realität sieht vielfach jedoch ganz anders aus. Aus Unachtsamkeit werden häufig öffentliche `set()`-Methoden für *eigentlich unveränderliche* Attribute angeboten. Das wurde in Abschnitt 16.2.1 als BAD SMELL: UNNÖTIGERWEISE VERÄNDERLICHE ATTRIBUTE beschrieben. Besonders leicht geschieht dies bei Erweiterungen einer Klasse um neue Attribute: Statt zu deren Initialisierung weitere Parameter in die Konstruktorsignatur aufzunehmen, werden `set()`-Methoden bereitgestellt. Dadurch können aber andere Klassen zu beliebigen Zeitpunkten Änderungen an den Werten der Attribute vornehmen. Das macht es deutlich schwieriger, für einen konsistenten und gültigen Objektzustand (vgl. Abschnitt 3.1.5) zu sorgen. Um die Zustandsänderungen auf wirklich notwendige einzugrenzen, sollte man die Veränderlichkeit von Attributen vermeiden. Dazu muss für alle *privaten* Attribute geprüft werden, ob man sie `final` deklarieren kann, wie es nachfolgend beschrieben wird. Ist dies der Fall, wird die zugehörende `set()`-Methode entfernt und die Konstruktorsignatur erweitert.

Betrachten wir das notwendige Vorgehen anhand der folgenden statischen inneren Klasse `TreeParameter` mit den zwei privaten Attributen `id` und `name` und öffentlichen Zugriffsmethoden. Oftmals muss man für eigene Klassen zuerst mithilfe des Re-

factorings REDUZIERE DIE SICHTBARKEIT VON ATTRIBUTEN möglichst viele Attribute und Zugriffsmethoden `private` machen, bevor man dieses Refactoring beginnt.

```
public static final class TreeParameter
{
    private long id = 0;
    private String name = null;

    public TreeParameter()
    {}

    public void setId(final long id)
    {
        this.id = id;
    }

    public long getId()
    {
        return this.id;
    }

    public void setName(final String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return this.name;
    }
}
```

Schritt 1: Prüfe auf Schreibzugriffe – Definiere ein Attribut `final`

Ziel ist es, alle Attribute im Konstruktor zu initialisieren. Wähle ein privates Attribut und definiere dieses zunächst `final`. Eine für das entsprechende Attribut vorhandene Defaultinitialisierung wird in diesem Schritt auskommentiert und als Zuweisung in den Konstruktor übernommen. Das folgende Listing zeigt Schritt 1 für das Attribut `id`:

```
public static final class TreeParameter
{
    private final long id;        // = 0;
    private String name = null;

    public TreeParameter()
    {
        this.id = 0;
    }
    // ...
}
```

Als Folge von Schritt 1 sieht man durch Fehlermeldungen beim Kompilieren, ob innerhalb der Klasse selbst noch schreibende Zugriffe durch Methoden oder Zuweisungen außerhalb des Konstruktors erfolgen. Besonders hilfreich für diesen Schritt ist die automatische Kompilierung in der IDE, weil man so direkt eine Rückmeldung erhält.

Treten keine Kompilierfehler auf, so erfolgte zuvor kein Schreibzugriff auf dieses Attribut und es wurde bislang nur lesend benutzt.[12] Schritt 1 wird dann für das nächste private Attribut ausgeführt. Bei Kompilierfehlern sind zwei Fälle zu unterscheiden. Erfolgt lediglich ein schreibender Zugriff durch eine `set()`-Methode, so ignorieren wir zunächst den Fehler »`The final field XYZ cannot be assigned`« zum unerlaubten Schreibzugriff in der `set()`-Methode und es geht weiter mit Schritt 2. Finden jedoch Zuweisungen innerhalb mehrerer Methoden statt, so erfolgt eine Rückkorrektur.

Schritt 1 – Rückkorrektur Man entfernt das zuvor eingeführte `final` wieder. Auch die gegebenenfalls entfernte Defaultinitialisierung wird wieder einkommentiert. Gehe vor zu Schritt 5.

Schritt 2: Prüfe, ob die `set()`-Methode entfernt werden kann

Erreicht man diesen Schritt, so verhindert lediglich eine `set()`-Methode, dass das entsprechende Attribut unveränderlich gemacht werden kann. Man prüft nun, ob überhaupt ein externer Aufruf durch andere Klassen an diese `set()`-Methode erfolgt. Dazu kann man entweder über das Tastaturkürzel CTRL+SHIFT+G oder über das Menü SEARCH –> REFERENCES –> WORKSPACE die Verwendungsstellen ausfindig machen. Alternativ wird dazu die Methode testweise `private` gekennzeichnet, und man nutzt wieder den Compiler, um Probleme oder Abhängigkeiten aufzudecken[13]:

```
public static final class TreeParameter
{
    private final long id;        // = 0;
    private String name = null;

    public TreeParameter()
    {
        this.id = 0;
    }

    // Verbliebener Schreibzugriff?
    private void setId(final long id)
    {
        this.id = id;
    }
    // ...
}
```

Die weiteren Schritte ergeben sich anhand der Ergebnisse der Kompilierung:

- Im besten Fall zeigt der Compiler bzw. die IDE lediglich Fehler in der eigenen Klasse an. Damit hat man externe Aufrufe der `set()`-Methode ausgeschlossen. Aufrufe innerhalb der eigenen Klasse können allerdings noch existieren. Schritt 3 beschreibt das Vorgehen für diesen Fall.

[12] Das gilt, sofern keine Zugriffe per Reflection erfolgen.

[13] Diese Aussage gilt allerdings nur unter der Voraussetzung, dass man allen Sourcecode vorliegen hat, der auf die Klasse zugreift.

- Gibt es Kompilierfehler außerhalb der eigenen Klasse, so wird die `set()`-Methode von anderen Klassen aufgerufen. Um kein Fass ohne Boden aufzumachen und die Änderungen lokal zu begrenzen, empfiehlt sich die nachfolgend angegebene Rückkorrektur. Mit Schritt 3 sollte man nur dann weitermachen, wenn nur sehr wenige externe Schreibzugriffe existieren, die zudem lediglich Änderungen direkt nach dem Konstruktionsprozess der betrachteten Klasse durchführen, ähnlich zu Folgendem:

```
final TreeParameter treeParams = new TreeParameter();
treeParams.setId(...);
treeParams.setName(...);
```

  In Schritt 4 wird der Konstruktor noch um einen Übergabeparameter erweitert.

Schritt 2 – Rückkorrektur Es müssen sowohl die ursprüngliche Sichtbarkeit der Methode wiederhergestellt als auch die Definition des Attributs als `final` entfernt werden. Zudem wird die gegebenenfalls entfernte Defaultinitialisierung wieder eingefügt, d. h., die Kommentarzeichen davor werden entfernt. Führe als Nächstes Schritt 5 aus.

Schritt 3: Prüfen interner Schreibzugriffe

Gibt es keine externen Zugriffe oder können die aufrufenden Programmstellen adäquat angepasst werden, so prüft man abschließend auf Schreibzugriffe innerhalb der Klasse selbst. Existieren mehrere davon, so ist das Attribut veränderlich und es erfolgt die in Schritt 2 angegebene Rückkorrektur. Gibt es jedoch nur einen Schreibzugriff, entfernt man die `set()`-Methode und es folgt Schritt 4.

Schritt 4: Anpassung des Konstruktors

Das Attribut muss jetzt zur korrekten Initialisierung im Konstruktor zugewiesen werden. Die Konstruktorsignatur wird dazu um einen korrespondierenden Übergabeparameter erweitert und der Konstruktorblock um eine entsprechende Zuweisung ergänzt. Diese ersetzt die in Schritt 1 eingeführte Zuweisung im Konstruktor.

```
public static final class TreeParameter
{
    private final long id;
    private String name = null;

    public TreeParameter(final long id)
    {
        this.id = id;
    }

    // ...
}
```

Schritt 5: Wiederhole Schritt 1 bis 4 für alle privaten Attribute

Wiederhole das Vorgehen der Schritte 1 bis 4 für alle verbliebenen privaten Attribute. Danach ergibt sich für die Klasse `TreeParameter` folgende Realisierung:

```
public static final class TreeParameter
{
    private final long id;
    private final String name;

    public TreeParameter(final long id, final String name)
    {
        this.id = id;
        this.name = name;
    }

    public long     getId()         { return this.id; }
    public String   getName()       { return this.name; }
}
```

Fazit

Als Ergebnis dieses Refactorings erhalten wir für dieses Beispiel eine Klasse, die keine Zustandsänderungen mehr erlaubt. Die Gefahr von unvorhersehbaren und unerwünschten Objektzuständen konnte vollständig eliminiert werden. Nicht in jedem Fall erreicht man eine unveränderliche Klasse, jedoch immer eine, deren Zustandsänderungen auf das wirklich notwendige Maß reduziert wurden.

Nach Abschluss dieses Refactorings sind möglicherweise einige Attribute noch öffentlich über Accessor-Methoden zugänglich, obwohl dies eventuell nicht notwendig ist. Es bieten sich zwei weitere Refactorings im Anschluss an.

Sinnvolle weitere Schritte

Methoden können in ihrer Sichtbarkeit eingeschränkt werden, um eine gute Datenkapselung und eine schlankere Schnittstelle zu erzielen. Man folgt dabei den im nachfolgenden Abschnitt 17.4.3 beschriebenen Schritten des Refactorings REDUZIERE DIE SICHTBARKEIT VON METHODEN. Darüber hinaus empfiehlt es sich, für öffentliche Methoden – zumindest für solche an Schnittstellen zu anderen Systemen oder Komponenten – mithilfe von Parameterprüfungen gültige Eingaben sicherzustellen. Eine Anleitung dazu wird im Refactoring ÜBERPRÜFE EINGABEPARAMETER in Abschnitt 17.5.2 beschrieben.

17.4.3 Reduziere die Sichtbarkeit von Methoden

Der OO-Kerngedanke der Kapselung ist, die Sichtbarkeit von Attributen und Methoden möglichst einzuschränken. Der Zustand eines Objekts kann folglich nicht mehr so sehr von außen bestimmt werden. Veränderungen sollten nur durch eine Reihe von verhaltensdefinierenden Business-Methoden erfolgen. Die Schritte dazu beschreibt das

Refactoring ERSETZE MUTATOR- DURCH BUSINESS-METHODE in Abschnitt 17.4.4. Alle nicht in dieser Schnittstelle enthaltenen Methoden versucht man, `private` oder `protected` zu definieren. Dazu sind folgende zwei Vorgehensweisen möglich – bitte beachten Sie aber die später genannten Einschränkungen, bevor Sie überall im Programm die Sichtbarkeit ändern.

1. **Statische Sourcecode-Analyse einsetzen** – Man nutzt Tools zur statischen Sourcecode-Analyse, um unbenutzte Methoden zu ermitteln. Während diese Funktionalität bereits in IntelliJ IDEA integriert ist, bietet es sich für Eclipse an, das Plugin UCDETECTOR[14] im Marketplace nachzuinstallieren. Dieses Tool findet unbenutzten Sourcecode sowie Methoden, deren Sichtbarkeit verringert werden kann. Es ermittelt auch, ob Methoden oder Attribute `final` sein können. Nützlich ist zudem, dass Methoden gefunden werden, die lediglich von Tests aufgerufen werden und damit nicht zur Anwendungsfunktionalität beitragen. Als praktisches Feature werden QUICK-FIX-Vorschläge bereitgestellt.
2. **Methodensichtbarkeit auf Verdacht ändern** – Hierbei gibt es die beiden folgenden Varianten:
   - Ausgehend von der aktuellen Sichtbarkeit verringert man diese schrittweise, also etwa von `public` -> `protected` -> Package-private -> `private`, bis entweder das Kompilieren fehlschlägt oder `private` erreicht ist. Man nimmt die kleinste Sichtbarkeit, bei der kein Kompilierfehler auftritt.
   - Man setzt die Methodensichtbarkeit zunächst auf `private`. Bei Bedarf erhöht man schrittweise die Sichtbarkeit, bis das Programm wieder kompiliert.

Anekdote: Positive Auswirkungen dieses Refactorings

Mir wurde ein umfangreicheres Programmpaket übertragen, in dem vieles zu kompliziert realisiert war. Das lag unter anderem daran, dass nahezu keine Kapselung stattfand und es vor `public`-Methoden wimmelte. Nach einigen Tagen hatte ich die Sichtbarkeit vieler Methoden gemäß diesem Refactoring reduzieren können. Dadurch konnten ca. 20 % der Methoden als unbenutzt erkannt werden und entfallen.

Aufdecken komplexer und nutzloser Zyklen Beim vorherigen Analysieren der Klassen hatte ich bereits das Gefühl, dass noch mehr Optimierungspotenzial existierte. Daher verfolgte ich die Aufrufe einiger Methoden und erkannte, dass diese eine ringförmige Aufrufhierarchie über einige Delegationen in verschiedenen Klassen enthielten. Durch aufwendiges Nachverfolgen und ein wenig Glück konnte ich etwa weitere 20 % nutzlosen Sourcecode entfernen.

Verbesserungen Die verbliebenen Klassen waren nun wesentlich übersichtlicher. An vielen Stellen konnten Attribute `final` definiert werden, wodurch wiederum die Anzahl der Zustandsänderungen begrenzt und die Klassen somit besser handhabbar wurden.

[14] `http://www.ucdetector.org/`

Einschränkungen

Die Reduktion der Sichtbarkeit kann man nur *sicher* einsetzen, wenn man alle nutzenden Klassen im Zugriff hat und außerdem weiß, dass keine Aufrufe mit Reflection erfolgen. Im ersten Fall würde man durch Änderungen der Sichtbarkeit eventuell Kompilierfehler in externen Komponenten verursachen. Im zweiten Fall kommt es eventuell zu Laufzeitfehlern, da benötigte Informationen über Reflection nicht zugreifbar sind.[15]

Dieses Refactoring ist außerdem nur für solche Methoden sinnvoll, die keine Kernfunktionalität der Klasse darstellen und daher auch nicht in der öffentlichen Schnittstelle für andere Klassen bereitgestellt werden sollen. Wendet man dieses Vorgehen wahllos für alle Methoden der Schnittstelle an, so führt dies nicht unbedingt sofort zu Problemen, weil die Reduktion der Sichtbarkeit momentan noch unbenutzter Methoden keinen Kompilierfehler verursacht. Allerdings wird später für Nutzer der Einsatz der Klasse erschwert, weil einige Methoden dann nicht in der Schnittstelle ansprechbar sind. Deswegen ist dieses Refactoring bevorzugt dann einzusetzen, wenn das System schon (fast) vollständig implementiert ist. Andernfalls weiß man z. B. lediglich, dass eine gegebene Methode momentan nicht außerhalb der Klasse benötigt wird.

17.4.4 Ersetze Mutator- durch Business-Methode

Dieses Refactoring hilft dabei, den Zustand eines Objekts vor feingranularen Änderungen zu schützen, und hat den objektorientierten Gedanken der Definition von Verhalten durch Business-Methoden im Blick.

Benutzen wir hier nochmals grafische Figuren als Beispiel. Zum Setzen der X- und Y-Koordinaten könnte man die Methoden `setX(int)` und `setY(int)` anbieten. Analog dazu könnten die Methoden `setEndX(int)` und `setEndY(int)` die Endposition und somit indirekt Breite und Höhe setzen. Die Schnittstelle ist akzeptabel, suggeriert aber einen Datenbehältercharakter und erlaubt feingranulare Zustandsänderungen von außen. Insbesondere dann, wenn Aufrufer mehrere Methoden am Zielobjekt hintereinander aufrufen, ist es mitunter besser verständlich, Methoden zu bündeln. Im Beispiel würde man Methoden anbieten, mit denen die Startposition oder die Größe einer Figur geändert werden kann (vgl. Abbildung 17-6). Dabei ist aber zu bedenken, dass z. B. in einem Zeichenprogramm die feingranulareren Aufrufe besser geeignet sind.

Abbildung 17-6 *Einführen von Business-Methoden*

[15] Reflection wird oft von Frameworks genutzt. Vielfach kommt es dann erst zur Laufzeit zu Problemen, wenn die aufzurufenden Methoden nicht die erwartete und benötigte Sichtbarkeit aufweisen.

17.4.5 Minimiere Zustandsänderungen

Je weniger Zustandsinformationen in einem Objekt variabel sind, desto besser lassen sich Modifikationen daran kontrollieren und nachvollziehen. Um die Veränderlichkeit eines Objekts möglichst gering zu halten, kann man die zuvor vorgestellten Refactorings kombinieren und führt folgende Schritte durch:

1. Wir reduzieren die Anzahl und die Sichtbarkeit der veränderlichen Attribute. Dazu nutzen wir die Refactorings REDUZIERE DIE SICHTBARKEIT VON ATTRIBUTEN aus Abschnitt 17.4.1 und MINIMIERE VERÄNDERLICHE ATTRIBUTE aus Abschnitt 17.4.2.
2. Wir halten die Anzahl und die Sichtbarkeit von `set()`-Methoden möglichst gering. Dazu wenden wir das Refactoring REDUZIERE DIE SICHTBARKEIT VON METHODEN aus Abschnitt 17.4.3 auf die `set()`-Methoden an. Zudem ersetzen wir Zugriffsmethoden durch Business-Methoden, wie dies im Refactoring ERSETZE MUTATOR- DURCH BUSINESS-METHODE in Abschnitt 17.4.4 dargestellt ist.

17.4.6 Führe ein Interface ein

Dieses Refactoring beschreibt, wie wir aus einer bestehenden Klasse ohne Interface eine Klasse machen, die ein Interface anbietet. Zuvor möchte ich aber kurz darauf hinweisen, dass dies nicht in jedem Fall sinnvoll ist, etwa für nur interne Klassen, für Klassen mit öffentlicher Schnittstelle dagegen schon.

Bei diesem Refactoring bündeln wir die gewünschten öffentlichen Methoden in einem Interface. In Eclipse können wir dazu das Refactoring EXTRACT INTERFACE aus dem Menü REFACTOR nutzen.

In folgendem Beispiel enthält das neu entstehende Interface `NewInterface` der Einfachheit halber alle `public`-Methoden der ursprünglichen Klasse `ClassWithoutInterface`. Das zugehörige UML-Diagramm zeigt Abbildung 17-7.

Abbildung 17-7 *Einführen eines Interface*

Für komplexere Klassen ist dies häufig nur ein erster Schritt, um die nach außen sichtbare Funktionalität zu definieren. Ergänzend kann das folgende Refactoring SPALTE EIN INTERFACE AUF genutzt werden, um eine semantische Strukturierung vorzunehmen.

Achtung: Interfaces sind kein Selbstzweck

Ein Interface ist ***kein Selbstzweck*** und nicht immer hilfreich: Für den Fall, dass eine Klasse nur intern in einem Package benutzt wird, sorgt ein Interface oftmals lediglich für mehr Komplexität und mehr Sourcecode, allerdings ohne größeren Nutzen. An Systemgrenzen oder weil das Design eine Abstraktion erfordert, ***können Interfaces für eine klare Trennung zwischen Angebot und Realisierung von Funktionalitäten sorgen***. Dadurch erhält man mehr Klarheit über die angebotene Funktionalität und erzielt auch eine losere Kopplung. Zudem lässt sich auf diese Weise eine konkrete Realisierung verbergen oder austauschbar gestalten. Vorteilhaft ist dies insbesondere für Klassen, die aus anderen Packages verwendet werden sollen.

17.4.7 Spalte ein Interface auf

Manchmal stellt man im Verlauf der Entwicklung fest, dass ein Interface besser in zwei oder mehr Interfaces aufgeteilt werden sollte. Dazu dient dieses Refactoring.

Als Beispiel soll im Folgenden das Interface `GlobalInterface` aufgespalten werden, da es neben reinen Lesezugriffen auch Business-Funktionalität anbietet. Mithilfe dieses Refactorings entstehen dann ein Datenbehälter-Interface (`DataProviderInterface`) mit `get()`-Methoden und ein Interface (`BusinessInterface`) mit Business-Methoden (siehe Abbildung 17-8).

Abbildung 17-8 *Einführen mehrerer Interfaces*

Mit diesem Refactoring kann man eine Trennung von Zuständigkeiten realisieren: Die ehemals auf unterschiedlichen semantischen Ebenen stattfindenden Zugriffe werden dann durch zwei Interfaces mit jeweils verschiedenen Spezialisierungen ausgedrückt. In diesen Interfaces gruppieren wir die Methoden nach ihrer Zugehörigkeit.

17.4.8 Führe ein Read-only-Interface ein

In komplexeren Systemen kann es zum Schutz vor ungewollten Änderungen durch andere Komponenten sinnvoll sein, Read-only-Interfaces anzubieten. Stellen wir uns folgendes Szenario vor: Eine Klasse wird von Package-externen Klassen lesend genutzt und von Package-internen Klassen schreibend mit Daten versorgt.

Ohne Zugriffsschutz könnten die Modelldaten nicht nur Package-intern, sondern ebenso durch Package-externe Klienten verändert werden. Um dies zu verhindern, kann man ein Read-only-Interface einführen, das durch `get()`-Methoden ausschließlich lesenden Zugriff bietet und den Package-externen Klassen bereitgestellt wird. Die Transformation ist in Abbildung 17-9 visualisiert.

Abbildung 17-9 *Einführen eines Read-only-Interface*

Hinweis Bedenken Sie, dass es manchmal bereits ausreicht, wenn man die modifizierenden Methoden Package-private definiert. Das gilt ebenfalls für Methoden, die nicht für Package-externe Klienten zugreifbar sein sollten. Das Read-only-Interface ist insbesondere für lose gekoppelte Komponenten sinnvoll.

17.4.9 Führe ein Read-Write-Interface ein

Wir haben gesehen, dass man mithilfe von Read-only-Interfaces schreibende Zugriffe durch Package-externe Klassen verhindern kann. Manchmal sollen aber einige Komponenten doch schreibend zugreifen, allerdings unter der Prämisse, diesen keinen direkten Zugriff auf konkrete Objekte anzubieten. Dann kann man ein spezielles Interface definieren, das lediglich Schreibzugriffe bereitstellt.

Problematisch an einer strikten Trennung zwischen rein lesenden und rein schreibenden Zugriffen ist, dass dies zu unhandlich wird, weil man dann jeweils eine Referenz unterschiedlichen Typs auf die gleiche Klasse benötigt. Vielfach bietet sich ein Read-Write-Interface an, das das Read-only-Interface um die Möglichkeit von Schreibzugriffen erweitert. Dadurch hat man im Read-Write-Interface immer lesenden und schreibenden Zugriff. Die Verwaltung und Nutzung wird einfacher, da man nur noch eine Referenz auf das Read-Write-Interface benötigt, statt, je nach Aufgabe, zwei Referenzen auf das Read-only- bzw. Write-Interface. Abbildung 17-10 zeigt diese Varianten.

Abbildung 17-10 *Einführen eines Read-Write-Interface*

Einschränkung Wie schon zuvor argumentiert, benötigt man derartige Konstrukte insbesondere dann, wenn Komponenten möglichst unabhängig voneinander sein sollen und eventuell von verschiedenen Teams oder sogar Firmen entwickelt werden. Für derartige Fälle sind solche Schnittstellen mit klaren Verantwortlichkeiten hilfreich.

17.4.10 Lagere Funktionalität in Hilfsmethoden aus

Dieses Refactoring dient dazu, den Sourcecode zu strukturieren, indem ähnliche und wiederkehrende Programmstücke erkannt und in Hilfsmethoden ausgelagert werden. In Eclipse können wir dazu das Refactoring EXTRACT METHOD nutzen. Nach einer solchen Überarbeitung folgt der Sourcecode dadurch besser dem DRY-Prinzip (»Don't Repeat Yourself«, vgl. »Der Pragmatische Programmierer« [34]).

Die folgende Methode `getFilterInfo(InformationProvider)` prüft verschiedene Bedingungen und baut korrespondierende Informationstexte zusammen:

```
public static String getFilterInfo(final InformationProvider infoProvider)
{
    final StringBuilder builder = new StringBuilder();

    if (isSelected(Option.NAME_CONTAINS))
    {
        if (builder.length() > 0)
        {
            builder.append(", ");
        }
        builder.append(getLangString("label.name") + "='" +
                       infoProvider.getName() + "'");
    }

    // weitere Auswertungen ...

    if (isSelected(Option.JOB_ID_CONTAINS))
    {
        if (builder.length() > 0)
        {
            builder.append(", ");
        }
        builder.append(getLangString("label.jobid") + "='" +
                       infoProvider.getJobID() + "'");
    }

    return builder.toString();
}
```

Die hier genutzte Methode `isSelected(Option)` prüft, ob eine Filterbedingung aktiviert ist. Anschließend werden die Informationen zu dieser Filterbedingung in ein `StringBuilder`-Objekt eingefügt. Die Methode `getFilterInfo(InformationProvider)` ist bereits relativ übersichtlich, besitzt aber kleinere Sprünge in den Abstraktionsebenen. Darüber hinaus existieren einige ähnliche Zeilen.

Schritt 1: Erzeugen von Hilfsmethoden

Wir erkennen zwei immer wiederkehrende Sourcecode-Abschnitte. Zum einen wird jeweils ein Komma und ein Leerzeichen an die Variable `builder` angehängt, sofern dort bereits Zeichen gespeichert sind. Das lässt sich einfach in eine Methode `appendCommaWhenNotEmpty(StringBuilder)` herausfaktorieren:

```
private static void appendCommaWhenNotEmpty(final StringBuilder builder)
{
    if (builder.length() > 0)
    {
        builder.append(", ");
    }
}
```

Zum anderen werden die Informationstexte zu den Filterbedingungen immer wieder auf ähnliche Art und Weise dem `StringBuilder`-Objekt hinzugefügt. Es erfordert etwas mehr Aufwand, diese Funktionalität in eine Methode auszulagern, weil die Unterschiede jeweils im Zugriffsschlüssel auf die Textressourcen und dem entsprechenden Wert liegen. Wir schreiben folgende Methode:

```
private static void appendFilterText(final StringBuilder builder,
                                     final String resourceID,
                                     final Object value)
{
    builder.append(getLangString(resourceID) + "='" + value + "'");
}
```

Schritt 2: Anpassen der Aufrufstellen

Nutzt man diese beiden Hilfsmethoden, so lässt sich der Sourcecode kürzer und in einer einheitlichen Abstraktionsebene darstellen. Damit verbessert sich die Struktur und die Lesbarkeit des Sourcecodes.

```
public static String getFilterInfo(final InformationProvider infoProvider)
{
    final StringBuilder builder = new StringBuilder();

    if (isSelected(Option.NAME_CONTAINS))
    {
        appendCommaWhenNotEmpty(builder);
        appendFilterText(builder, "label.name", infoProvider.getName());
    }

    // weitere Auswertungen ...

    if (isSelected(Option.JOB_ID_CONTAINS))
    {
        appendCommaWhenNotEmpty(builder);
        appendFilterText(builder, "label.jobid", infoProvider.getJobID());
    }

    return builder.toString();
}
```

17.4.11 Trenne Informationsbeschaffung und -verarbeitung

Vielfach ist es vorteilhaft, die Beschaffung von Informationen von ihrer Verarbeitung und Speicherung zu trennen. Diese Teilaufgaben lassen sich im Allgemeinen recht gut in separate Methoden auslagern, wodurch die Orthogonalität (Kombinierbarkeit) und Wiederverwendbarkeit erhöht sowie ungültige Zwischenzustände vermieden werden.

Betrachten wir eine Methode, in der Informationsbeschaffung und -verarbeitung noch miteinander verknüpft sind:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            model.setState(elementVO.getQueueStatus());     // 1
        }
        else
        {
            model.setState(QueueStates.UNKNOWN);            // 2
        }
        model.setQueuedJobs(elementVO.getEntryCount());     // 3
    }
    else
    {
        model.setState(QueueStates.UNREACHABLE);            // 4
        model.setQueuedJobs(0);                             // 5
    }
}
```

Daten werden aus einem DATA TRANSFER OBJECT (DTO) bzw. VALUE OBJECT (VO) (vgl. Abschnitt 3.4.5) `elementVO` ausgelesen und in einem Modell `model` gespeichert. Der Sourcecode enthält dazu fünf verschiedene Zeilen mit `set()`-Methoden. Die Informationsbeschaffung ist in den restlichen Methoden über `get()`-Aufrufe an `elementVO` verteilt. Das wirkt harmlos. Allerdings sollte man bedenken, dass prinzipiell bei jedem Methodenaufruf ein Fehler auftreten und als Folge eine Exception ausgelöst werden kann. Dadurch kann es zu Inkonsistenzen durch teilweise initialisierte Objekte kommen, wie es der folgende Praxistipp weiter ausführt. Die obige Art der Informationsbeschaffung kann auch beim Einsatz von Multithreading zu Inkonsistenzen führen (vgl. Kapitel 10).

Unser Ziel ist es, eine bessere Trennung zu erreichen. Im Idealfall können zunächst alle Informationen ermittelt, in einem Immutable-Objekt gespeichert und anschließend verarbeitet werden. Eine Umsetzung in Pseudocode sieht wie folgt aus:

```
final DataValueContainer values = retrieveValues();
processValues(values);
storeValues(values);
```

Achtung: Probleme durch zu feingranulare `set()`-Methoden

Einfluss auf Objektintegrität Eine schrittweise Veränderung von Attributen kann zu Zwischenzuständen führen, die keinen gültigen Objektzustand darstellen. Besonders groß ist die Gefahr dafür beim Einsatz von Methoden, die Exceptions auslösen können, etwa in verteilten Systemen mit Remote Calls. Treten Exceptions auf, so kann ein Objekt sehr schnell in einen inkonsistenten Zustand versetzt werden, etwa weil die Verarbeitung durch eine Exception abrupt beendet wird. Eine Trennung von Informationsbeschaffung und -verarbeitung ist dann vorteilhaft: Treten Fehler bei der Informationsbeschaffung auf, kann auf ein Setzen der Werte verzichtet und das Objekt somit in einem konsistenten Zustand belassen werden.

Einfluss auf BEOBACHTER Werden Zustandsänderungen gemäß dem BEOBACHTER-Muster (vgl. Abschnitt 18.3.7) verarbeitet, so bereiten feingranulare `set()`-Methoden neben ständigen Zustandsmeldungen zusätzlich das Problem, dass sie ungültige Zwischenzustände sichtbar machen.

Analogie aus dem realen Leben Warum potenziell teilinitialisierte Objekte nicht optimal sind, kann man sich mit folgender Analogie aus dem realen Leben verdeutlichen: Morgens verlässt man das Haus ja auch erst, nachdem man sich vollständig angezogen hat und den Rasierschaum oder die Zahnpasta aus dem Gesicht gewischt hat, also nur in einem validen Zustand. Tut man dies nicht, so wirkt das im besten Fall befremdlich auf die Umwelt. Denken Sie an diese Analogie, um sich die Wichtigkeit eines sinnvoll initialisierten Objekts zu vergegenwärtigen und möglichst sparsam zustandsändernde `set()`-Methoden für eigene Klassen anzubieten.

Schritt 1: Erkennen von veränderlichen Elementen und Einführen von Hilfsvariablen

Wir untersuchen den Sourcecode nach veränderlichen Werten und erkennen, dass im `if`- bzw. `else`-Zweig jeweils zwei Variablen durch `set()`-Aufrufe verändert werden. Wir führen daher entsprechende Hilfsvariablen `state` und `entryCount` ein. Dabei hilft uns das Basis-Refactoring EXTRACT LOCAL VARIABLE mit dem Tastaturkürzel ALT+SHIFT+L. Dies erfolgt getrennt für beide Blöcke. Ein derart kleinteiliges Vorgehen wäre hier eventuell nicht notwendig, wenn man sehr geübt ist. ***Aber je komplexer der Ausschnitt des Sourcecodes und je mehr verschiedene Variablen modifiziert werden, desto besser ist es, kleine überschaubare und am besten durch die IDE unterstützte Schritte zu machen***. Durch EXTRACT LOCAL VARIABLE werden die direkten Aufrufe der `set()`-Methoden durch Zuweisungen an die neu eingeführten lokalen Variablen und einen anschließenden Aufruf an die jeweilige `set()`-Methode des Modells ersetzt. Danach ordnen wir noch mit ALT+UP/DOWN ein paar Zeilen um und fassen im oberen Block die Variablen für `state` und `entryCount` zusammen:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    if (elementVO != null)
    {
        QueueStates state;
        int entryCount;

        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
            model.setState(state);
        }
        else
        {
            state = QueueStates.UNKNOWN;
            model.setState(state);
        }
        entryCount = elementVO.getEntryCount();
        model.setQueuedJobs(entryCount);
    }
    else
    {
        QueueStates state = QueueStates.UNREACHABLE;
        model.setState(state);

        int entryCount = 0;
        model.setQueuedJobs(entryCount);
    }
}
```

Trotz dieser umfangreicheren Modifikationen haben wir am Programmverhalten nichts geändert. Der Sourcecode ist allerdings länger geworden. Diesen offensichtlichen Nachteil akzeptieren wir zunächst, da eine entscheidende Verbesserung durch diese Umformung erreicht wurde: ***Die Informationsbeschaffung und der Transfer ins Modell erfolgen nun in zwei Schritten***. Diese Transformation bildet die Grundlage für die weiteren Modifikationen.

Schritt 2: Zusammenfassen der `set()`-Methoden zu Blöcken

Um die Informationsbeschaffung und -verarbeitung weiter zu entkoppeln, wollen wir die Aufrufe der `set()`-Methoden zu Einheiten am jeweiligen Ende eines Blocks zusammenfassen. Dies ist immer dann möglich, wenn der Kontrollfluss den Block vollständig durchläuft. Durch das Verlagern (Zeilen mit ALT+UP/DOWN bewegen) und das Zusammenfassen von `set()`-Methoden erhält man folgenden Zwischenstand:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    if (elementVO != null)
    {
        final QueueStates state;
        int entryCount;

        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
        }
```

```
        else
        {
            state = QueueStates.UNKNOWN;
        }
        entryCount = elementVO.getEntryCount();

        model.setState(state);
        model.setQueuedJobs(entryCount);
    }
    else
    {
        final QueueStates state = QueueStates.UNREACHABLE;
        final int entryCount = 0;

        model.setState(state);
        model.setQueuedJobs(entryCount);
    }
}
```

Schwierigkeit: Mehrere Ausgänge aus Blöcken bzw. in einer Methode

Mehrere `return`-Anweisungen innerhalb komplexer und geschachtelter `if`-Anweisungen erschweren häufig das Verständnis und hätten den gezeigten Schritt deutlich schwieriger – eventuell sogar unmöglich – gemacht. Die negative Aussage zu mehrfachen `return`-Anweisungen gilt jedoch nicht für Shortcut-Returns am Methodenanfang im Rahmen von Fehlerabfragen. Diese tragen fast immer zu einer Vereinfachung der Logik bei.

Schritt 3: Auftrennen von Informationsbeschaffung und -verarbeitung

Zur Trennung von Informationsbeschaffung und -verarbeitung strukturieren wir den Sourcecode um. Wir wollen die Blöcke mit den `set()`-Aufrufen zusammenfassen und definieren dazu Hilfsvariablen im äußersten Block. Nachdem der gesamte Zustand ermittelt wurde, werden dann die `set()`-Methoden ausgeführt. Gleichzeitig nehmen wir eine Vereinfachung vor: Die Initialisierung aus dem `else`-Zweig für den Fehlerfall (`elementVO == null`) nutzen wir als Startbelegung der Variablen:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    QueueStates state = QueueStates.UNREACHABLE;
    int entryCount = 0;

    // Informationsbeschaffung
    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
        }
        else
        {
            state = QueueStates.UNKNOWN;
        }
```

```
        entryCount = elementVO.getEntryCount();
    }

    // Verarbeitung
    model.setState(state);
    model.setQueuedJobs(entryCount);
}
```

Schritt 4: Kapseln von Informationsbeschaffung und -verarbeitung

Informationsbeschaffung und -verarbeitung sind nun getrennt und können in eigene Methoden ausgelagert werden. Dazu bietet es sich an, die lokalen Variablen zu Objekten zusammenzufassen. Doch schauen wir uns dies schrittweise an.

Zunächst nutzen wir das Basis-Refactoring EXTRACT METHOD, um die per Kommentar markierten Teile jeweils als Methode zu extrahieren. Wir beginnen mit den beiden Zeilen zur Verarbeitung, die wir in die Methode `storeInModel()` auslagern:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    QueueStates state = QueueStates.UNREACHABLE;
    int entryCount = 0;

    // Informationsbeschaffung
    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
        }
        else
        {
            state = QueueStates.UNKNOWN;
        }
        entryCount = elementVO.getEntryCount();
    }

    // Verarbeitung
    storeInModel(state, entryCount);
}

private void storeInModel(final QueueStates state, final int entryCount)
{
    model.setState(state);
    model.setQueuedJobs(entryCount);
}
```

Wenn wir nun die Zeilen zur Informationsbeschaffung startend vom Kommentar markieren und ebenso extrahieren wollen, dann liefert uns Eclipse die Fehlermeldung, dass der Rückgabewert nicht eindeutig bestimmbar ist und mehr als eine lokale Variable als Kandidat existiert. Was nun?

Wir hatten bereits vor, die beiden semantisch zusammenhängenden Variablen in einer Klasse zu vereinen. Praktischerweise gibt es dazu das Basis-Refactoring INTRODUCE PARAMETER OBJECT. Als Vorbereitung selektieren wir die Methode `storeInModel()` und führen danach das Refactoring aus. Es erscheint ein Dialog. Dort müs-

sen wir lediglich noch den gewünschten Klassen- und Parameternamen eintragen, etwa `QueueStateVO`. Dadurch entsteht eine öffentliche Klasse gemäß dem Muster VALUE OBJECT (vgl. Abschnitt 3.4.5) mit zwei öffentlichen Attributen `state` und `entryCount`. Der Aufruf wird automatisch in Folgendes umgewandelt:

```
storeInModel(new QueueStateVO(state, entryCount));
```

Um uns die Befüllungsarbeit zu erleichtern, extrahieren wir wieder eine lokale Variable per EXTRACT LOCAL VARIABLE:

```
final QueueStateVO queueStateVO = new QueueStateVO(state, entryCount);
storeInModel(queueStateVO);
```

In unserem Kontext benötigen wir keine öffentliche Klasse, sondern eher eine Package-private Hilfsklasse mit ebensolchen Attributen, die direkten Zugriff darauf erlauben. Wir modifizieren die generierte Klasse wie folgt:

```
class QueueStateVO
{
    // ACHTUNG: sind modifizierbar, werden durch andere Klasse geändert
    QueueStates state;
    int         entryCount;

    QueueStateVO(final QueueStates state, final int entryCount)
    {
        this.state = state;
        this.entryCount = entryCount;
    }
}
```

Hinweis: Value Objects und Records

Zunächst könnte man denken, dass sich dafür seit Java 14 ein `record` (vgl. Abschnitt 14.1.3) anbietet:

```
record QueueStateVO(QueueStates state, int entryCount)
{}
```

Das ist jedoch hier nicht der Fall, da die Attribute im Verlaufe der Verarbeitung geändert werden. Records sind nur dann für Data Transfer Objects oder Value Objects geeignet, wenn diese unveränderlich sind.

Nun kann man dann die Zeilen

```
QueueStates state = QueueStates.UNREACHABLE;
int entryCount = 0;
```

durch einen Konstruktoraufruf eines `QueueStateVO`s wie folgt ersetzen:

```
final QueueStateVO stateVO = new QueueStateVO(QueueStates.UNREACHABLE, 0);
```

Der Einsatz der Klasse `QueueStateVO` ist relativ selbsterklärend. Überall dort, wo früher eine Zuweisung an eine lokale Variable erfolgte, verwenden wir nun den Zugriff auf das korrespondierende Attribut des Value Object:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    final QueueStateVO stateVO = new QueueStateVO(QueueStates.UNREACHABLE, 0);

    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            stateVO.state = elementVO.getQueueStatus();
        }
        else
        {
            stateVO.state = QueueStates.UNKNOWN;
        }
        stateVO.entryCount = elementVO.getEntryCount();
    }

    storeInModel(stateVO);
}
```

Es bietet sich häufig an, Hilfsmethoden zur Datenbeschaffung und Verarbeitung einzuführen, statt nur einige Zeilen umzuordnen. Die Methode `updateModelFromElement(DataElementVO)` ändern wir wie folgt:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    final QueueStateVO stateVO = createQueueStateVOFromElement(elementVO);

    storeInModel(stateVO);
}
```

Dazu extrahieren wir folgende Methode `createQueueStateVOFromElement(DataElementVO)`, in der wir das `QueueStateVO`-Objekt mit Werten befüllen. Der entsprechende Sourcecode umfasst alle Zeilen zur Informationsbeschaffung der ursprünglichen Methode exklusive des `storeInModel()`-Aufrufs zur Verarbeitung:

```
QueueStateVO createQueueStateVOFromElement(final DataElementVO elementVO)
{
    final QueueStateVO stateVO = new QueueStateVO(QueueStates.UNREACHABLE, 0);

    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            stateVO.state = elementVO.getQueueStatus();
        }
        else
        {
            stateVO.state = QueueStates.UNKNOWN;
        }
        stateVO.entryCount = elementVO.getEntryCount();
    }

    return stateVO;
}
```

Fazit

Durch die Untergliederung in Informationsbeschaffung und Verarbeitung wurde eine Trennung von Zuständigkeiten erreicht. Das führt zu strukturellen Verbesserungen, die sich positiv auf die Lesbarkeit und Verständlichkeit des Sourcecodes auswirken. Zudem kann auch bei möglichen Fehlern während der Informationsbeschaffung ein konsistenter Objektzustand sichergestellt werden.

17.4.12 Wandle Konstantensammlung in `enum` um

Zum Teil findet man im Sourcecode einige Konstanten, die semantisch zusammengehören, aber nicht als eigenständiger Typ definiert sind. Dies wurde als BAD SMELL: ZUSAMMENGEHÖRENDE KONSTANTEN NICHT ALS TYP DEFINIERT in Abschnitt 16.1.3 besprochen. Dieses Refactoring hilft bei der Problemlösung.

Nehmen wir an, in der folgenden Klasse `BusinessClass` wären einige `int`-Konstanten definiert, zu denen teilweise korrespondierende `String`-Konstanten als Beschreibung vorhanden sind. Glücklicherweise sind die Konstanten über die Namenspräfixe `JOBSTATUS` und `ERRORCODE` gegliedert und lassen sich leicht unterscheiden:

```
public class BusinessClass
{
    public static final int     JOBSTATUS_UNDEFINED      = -1;
    public static final int     JOBSTATUS_ACTIVE         = 1;
    public static final int     JOBSTATUS_FINISHED       = 2;

    private static final String JOBSTATUS_UNDEFINED_NAME = "UNDEFINED";
    private static final String JOBSTATUS_ACTIVE_NAME    = "Active";
    private static final String JOBSTATUS_FINISHED_NAME  = "Finished";

    public static final int     ERRORCODE_OK             = 0;
    public static final int     ERRORCODE_WARN           = 1;
    public static final int     ERRORCODE_ERROR          = 2;

    private final BusinessJob job;

    public int getJobState()
    {
        if (job.isActive())
            return JOBSTATUS_ACTIVE;
        if (job.isFinished())
            return JOBSTATUS_FINISHED;

        return JOBSTATUS_UNDEFINED;
    }
    // ...
```

Schritt 1: Sammlung in einer eigenen Konstantenklasse

Anhand der `JOBSTATUS`-Konstanten zeige ich den Ablauf bei diesem Refactoring. Die bisher lose zusammenhängenden `JOBSTATUS`-Konstanten werden in einer eigenen Klasse gesammelt. Bei der Benennung ist es sinnvoll, einen Ober- oder Gliederungsbegriff zu verwenden. Besitzen die Konstanten ein gemeinsames, aussagekräftiges Prä-

fix, ist dies ein guter Anhaltspunkt für den Namen der Konstantenklasse. Wir nutzen hier den Namen `JobStates`. Die Klasse selbst deklarieren wir `final`, da sie nicht als Basis für weitere Klassen dienen soll:

```
public final class JobStates
{
    public static final int     JOBSTATUS_UNDEFINED      = -1;
    public static final int     JOBSTATUS_ACTIVE         = 1;
    public static final int     JOBSTATUS_FINISHED       = 2;

    private static final String JOBSTATUS_UNDEFINED_NAME = "UNDEFINED";
    private static final String JOBSTATUS_ACTIVE_NAME    = "Active";
    private static final String JOBSTATUS_FINISHED_NAME  = "Finished";
}
```

Schritt 2: Erzeugen eines Konstruktors und von Zugriffsmethoden

Die Konstanten bestehen aus einem Wert und einer Beschreibung. Daher führen wir in der Klasse `JobStates` zwei Attribute ein. Die Klasse wird entsprechend dem Muster IMMUTABLE-KLASSE (vgl. Abschnitt 3.4.2) konstruiert. Der private Konstruktor verhindert das Erzeugen weiterer Objekte außerhalb dieser Konstantenklasse. Zudem bieten wir nur lesenden Zugriff auf die Attribute.

```
public final class JobStates
{
    public static final int     JOBSTATUS_UNDEFINED      = -1;
    public static final int     JOBSTATUS_ACTIVE         = 1;
    public static final int     JOBSTATUS_FINISHED       = 2;

    private static final String JOBSTATUS_UNDEFINED_NAME = "UNDEFINED";
    private static final String JOBSTATUS_ACTIVE_NAME    = "Active";
    private static final String JOBSTATUS_FINISHED_NAME  = "Finished";

    private final String        name;
    private final int           value;

    private JobStates(final String name, final int value)
    {
        this.name = name;
        this.value = value;
    }

    public String getName()         { return this.name; }
    public int getValue()           { return this.value; }
}
```

Der Einsatz dieser Konstantenklasse hilft dabei, die Funktionalitäten der Business-Klasse von der Definition benötigter Konstanten zu trennen. Darüber hinaus können die Konstanten dann von anderen Klassen benutzt werden, ohne dass diese Kenntnis von der Business-Klasse besitzen müssen.

Schritt 3: Umwandlung in eine `enum`-Aufzählung

Als letzten Schritt wandeln wir die Konstantendefinition in eine `enum`-Aufzählung um:

```
enum JobStates
{
    UNDEFINED("UNDEFINED", -1), ACTIVE("Active", 1), FINISHED("Finished", 2);

    private final String name;
    private final int    value;

    private JobStates(final String name, final int value)
    {
        this.name = name;
        this.value = value;
    }

    public String getName()         { return this.name; }
    public int getValue()           { return this.value; }
}
```

Fazit

Konstanten aus Business-Klassen auszulagern, ist häufig bereits ein wichtiger Schritt für mehr Struktur und Verständlichkeit. Die nachfolgende Definition von Werten innerhalb einer `enum`-Aufzählung führt zu Typsicherheit und einer loseren Kopplung – hier besteht dadurch keine Abhängigkeit zur Business-Klasse mehr. Durch eine derartige Realisierung sind die Konstanten außerdem automatisch sortierbar und auch serialisierbar. Darüber hinaus erfolgt eine typsichere, objektorientierte Definition der Konstanten, wodurch Anwendungsfehler wesentlich unwahrscheinlicher als bei der Verwendung reiner `int`-Werte sind. Auch nachfolgende Refactorings können ohne größere Auswirkungen auf nutzende Klienten durchgeführt werden: Konstanten müssen ausschließlich in der `enum`-Aufzählung gepflegt werden. Der verwendende Sourcecode nutzt lediglich Referenzen auf die Konstanten.

17.4.13 Entferne Exceptions zur Steuerung des Kontrollflusses

Dieses Refactoring dient dazu, den BAD SMELL: EXCEPTIONS ZUR STEUERUNG DES KONTROLLFLUSSES zu korrigieren, wie er in Abschnitt 16.3.4 beschrieben ist.

Betrachten wir dazu die folgende, simple Methode `isAlive()`:

```
private void isAlive() throws DbException
{
    // Teste Existenz in Datenbank
    session.getDbObject(id);
}
```

Diese Methode wird in der folgenden `toString()`-Methode eingesetzt, um die Existenz des eigenen Objekts in der Datenbank vor der Erzeugung einer XML-

Repräsentation sicherzustellen. Ist das Objekt dort nicht vorhanden, so wird im `catch`-Block eine Fehlermeldung zurückgeliefert:

```
public String toString()
{
    try
    {
        isAlive();
        return createXmlStringInfo();
    }
    catch (final DbException e)
    {
        return "Object of class " + getClass().getSimpleName() + " is dead!";
    }
}
```

Achtung: Abgrenzung gegenüber Methoden zur Zustandsprüfung

Der Einsatz der Methode `isAlive()` erinnert an Methoden zur Zustandsprüfung. Allerdings gibt es einen entscheidenden Unterschied: `isAlive()` überprüft einen möglichen Zustand des Objekts, nämlich den, dass ein assoziiertes Objekt, hier ein Verweis in die Datenbank, nicht mehr gültig ist. Dieser Zustand ist zwar selten, stellt aber keinen ungültigen Objektzustand dar. Daher sollte man hier besser einen booleschen Rückgabewert statt einer Exception verwenden. Eine Zustandsprüfung sichert dagegen einen korrekten und erwarteten Objektzustand ab. Eine Parameterprüfung soll vor unvollständigen Initialisierungen bewahren. Ungültige Objektzustände stellen einen Fehler und eine ungewünschte Ausnahmesituation dar. Deshalb ist dafür der Einsatz von Exceptions sinnvoll.

Schritt 1: Einführen eines Rückgabewerts

Der Methodenname `isAlive()` impliziert die Rückgabe eines booleschen Werts. Daher wandeln wir die Methode dementsprechend um. Dazu fangen wir die zuvor in der Signatur definierte Exception ab und geben im korrespondierenden `catch`-Block `false` zurück. Eine erfolgreiche Prüfung führt zur Rückgabe des Werts `true`. Abschließend wird die Exception aus der Signatur entfernt:

```
private boolean isAlive()
{
    try
    {
        session.getDbObject(id);  // Teste Existenz in Datenbank
        return true;
    }
    catch (final DbException e)
    {
        return false;
    }
}
```

Es kommt hier zu einer Änderung der Signatur, weil die Exception daraus entfernt wird. In diesem Fall ist dies allerdings problemlos möglich, da es sich um eine private Methode handelt und wir dadurch die komplette Kontrolle über den verwendenden Sourcecode haben. Weitere Hinweise liefert folgender Praxistipp.

Achtung: API-Änderungen durch Refactorings

Bei Änderungen an der Methodensignatur muss man vorsichtig vorgehen, um keine Inkompatibilitäten einzuführen. Änderungen an öffentlichen Methoden erfordern Änderungen in nutzenden Klienten. Häufig sind daher Änderungen am API nicht folgenlos durchführbar. Eine Methode sollte dann besser per Annotation als `@Deprecated` markiert und durch eine Version mit verbessertem API ersetzt werden. ***Private Methoden können immer (abgesehen von Reflection) ohne Auswirkungen für andere externe Klassen geändert werden***.

Nach den Transformationen wird offensichtlich, dass der Rückgabewert der `getDbObject()`-Methode nicht ausgewertet wird. Dies ist generell unschön, für dieses Beispiel allerdings unbedeutend. Ähnliches wurde in Abschnitt 16.1.8 als BAD SMELL: UNVOLLSTÄNDIGE BETRACHTUNG ALLER ALTERNATIVEN diskutiert.

Schritt 2: Auswertung des Rückgabewerts statt Exception Handling

Das Exception Handling mit `try-catch` ist nun überflüssig und wird durch eine `if-else`-Auswertung des Rückgabewerts ersetzt.[16] Dadurch ergibt sich folgender, besser lesbarer Sourcecode:

```
public String toString()
{
    if (isAlive())
    {
        return createXmlStringInfo();
    }
    return "Object of class " + getClass().getSimpleName() + " is dead!";
}
```

Fazit

Die vermeintliche »Fehlerbehandlung« (Abfrage der Existenz) erfolgt nun so lokal wie möglich, nämlich an der Stelle des Datenbankzugriffs in der `isAlive()`-Methode. Für die Aufrufer der Methode bleibt dies verborgen. Dadurch lässt sich nutzender Sourcecode klarer gestalten. Falls jedoch lokal innerhalb von Verarbeitungsmethoden nicht sinnvoll auf Probleme reagiert werden kann, sollten Exceptions propagiert werden.

[16] Je höher die Anzahl der `if`-Abfragen oder Verschachtelungen, desto eher sind mehrere `return`-Anweisungen zu vermeiden, weil sie dann den Programmfluss meistens schwieriger erkennbar machen. Für Shortcut-Returns am Methodenanfang oder kurze Methoden bis etwa 20 Zeilen können mehrere `return`-Anweisungen aber sogar für mehr Klarheit sorgen.

17.4.14 Wandle in Utility-Klasse mit statischen Hilfsmethoden um

Utility-Klassen sollten kein Objektverhalten anbieten und daher lediglich aus einer Sammlung von Hilfsmethoden bestehen. Manchmal besitzen diese Klassen aus Unachtsamkeit allerdings doch einen öffentlichen Konstruktor und einige Zustandsattribute sowie nicht statische Hilfsmethoden. Oftmals müssen dem Konstruktor dazu Parameter übergeben werden, die später innerhalb der Hilfsmethoden genutzt werden.

Als Beispiel dient eine Methode `writeDoc(IDocument, String, String)`, die Objekte vom Typ `IDocument` speichern soll. Dazu wird eine Instanz der Klasse `DocumentToFilesystem` genutzt und zuvor erzeugt, um dann eine parameterlose `doSave()`-Methode aufzurufen. Die benötigten Parameter werden aber bereits an den Konstruktor der Klasse `DocumentToFilesystem` übergeben:

```
public void writeDoc(final IDocument document,
                     final String path, final String fileName)
{
   final DocumentToFilesystem docToFileSystem = new DocumentToFilesystem(
                                                    document, path, fileName);

   docToFileSystem.doSave();
}
```

Obwohl es sich bei der Klasse `DocumentToFilesystem` eigentlich um eine reine Utility-Klasse handelt, muss immer (künstlich) ein entsprechendes Objekt erzeugt und korrekt initialisiert werden, um die Hilfsmethoden, hier etwa die Methode `doSave()`, aufrufen zu können. Diese vom Design her missglückte Utility-Klasse – weil sie statt statischer Hilfsmethoden lediglich Objektmethoden anbietet – ist in Abbildung 17-11 als Klassendiagramm dargestellt.

Abbildung 17-11 *Falsche Realisierung einer Utility-Klasse*

Zum Aufruf von Objektmethoden erfolgt zunächst immer eine Parameterübergabe im Konstruktor. Sinnvoller wäre es jedoch, benötigte Parameter direkt an die jeweiligen Methoden zu übergeben. Im Folgenden prüfen wir, ob für eine Utility-Klasse tatsächlich ein Zugriff auf die Zustandsinformationen notwendig ist. Ziel bei diesem Refactoring ist es, die Zustandsinformationen aus der Utility-Klasse möglichst vollständig zu entfernen und stattdessen Aufrufparameter an statische Methoden zu verwenden.

Schritt 1: Umwandlung in statische Hilfsmethode

Um eine Methode vom Objektzustand unabhängig zu machen, kann man diese zunächst statisch definieren. In Form von Kompilierfehlern werden automatisch mögliche Abhängigkeiten zu verwendeten Attributen sichtbar. Existieren keine Abhängigkeiten, so kann man eine weitere Methode wählen und erneut diesen Schritt 1 ausführen. Ansonsten löst man die erkannten Abhängigkeiten, wie in Schritt 2 beschrieben, auf.

Schritt 2: Erweiterung der Methodensignatur

Damit die Methode unabhängig von den Attributen einer Instanz der Klasse wird, müssen alle Abhängigkeiten durch Übergabeparameter aufgelöst werden. Man erweitert die Methodensignatur um die zur Ausführung erforderlichen Parameter. Für die Methode `doSave()` werden die Informationen aus den Attributen `file`, `path` und `filename` benötigt. In Abbildung 17-12 ist dies als Klassendiagramm verdeutlicht.[17]

Abbildung 17-12 *Utility-Klasse nach Schritt 2*

Durch die Parameterübergabe entfällt die Notwendigkeit zur Konstruktion der Utility-Klasse, nur um eine Hilfsmethode nutzen zu können. Dies erleichtert die Nutzbarkeit:

```
public void writeDoc(final IDocument document,
                     final String path, final String fileName)
{
    DocumentToFilesystemStep2.doSave(document, path, fileName);
}
```

Schritt 3: Umwandlung aller Methoden in entsprechende Hilfsmethoden

Durch Wiederholen der Schritte 1 und 2 werden im besten Falle alle Methoden bearbeitet und in statische Hilfsmethoden transformiert, sodass dann nur noch statische Hilfsmethoden in der Utility-Klasse definiert sind, wie es Abbildung 17-13 zeigt.
Allerdings existieren noch die statischen Attribute sowie eine Abhängigkeit zum jeweiligen Konstruktor. Dies wird, sofern möglich, im letzten Schritt des Refactorings entfernt.

[17] Statische Methoden werden in der UML unterstrichen.

Abbildung 17-13 *Utility-Klasse nach Schritt 3*

Schritt 4: Entfernen von Zustandsinformationen und Anpassen von Konstruktoren

Wenn alle Methoden erfolgreich bearbeitet wurden, müssen keine Zustandsinformationen mehr gespeichert werden. Sowohl die Konstruktoren mit Parametern als auch die korrespondierenden Attribute sind damit obsolet und können entfallen.

Wir entfernen alle öffentlichen Konstruktoren und erstellen einen privaten Defaultkonstruktor. Dadurch werden Objektkonstruktionen der Utility-Klasse durch andere Klassen unterbunden. Diese Klasse wird anschließend in `DocumentToFilesystemUtils` umbenannt, was zur Lesbarkeit beiträgt. Das in Abbildung 17-14 dargestellte Klassendiagramm verdeutlicht dies.

Abbildung 17-14 *Utility-Klasse nach Schritt 4*

Fazit

Durch die erfolgten Transformationen ist nun der Utility-Charakter der Klasse offensichtlich. Zudem werden alle Abhängigkeiten und Parametrierungen klar. Außerdem werden Anwendungsfehler vermieden, die bei einer Konstruktion von Objekten der ursprünglichen Klasse auftreten konnten: Ist beispielsweise die Wertebelegung für eine `doSave()`-Aktion ausgelegt, so ist diese meistens für den Aufruf anderer Hilfsmethoden unpassend und führt vermutlich zu Fehlern. Gibt es mehrere Instanzen der Utility-Klasse, so steigt die Wahrscheinlichkeit, eine nicht passend initialisierte Instanz einzusetzen.[18]

[18] Natürlich kann man nicht ausschließen, dass die Werte beim Methodenaufruf auch falsch belegt sind. Die Fehler werden jedoch schneller klar.

17.4.15 Löse `if-else` / `instanceof` durch Polymorphie auf

In der Praxis findet man des Öfteren Programmstellen die verschiedene Bedingungen prüfen und abhängig davon Aktionen ausführen. Mitunter sind dies aber lediglich einfache Zuweisungen, was wir uns im Anschluss separat im Praxishinweis »Bedingte Zuweisungen übersichtlich gestalten« anschauen.

Hier wollen wir einen anderen Fall betrachten, nämlich den, dass diverse Typprüfungen erfolgen und basierend darauf Aktionen ausgelöst werden:

```
for (final Object obj : figures)
{
    if (obj instance Rect)
    {
        ((Rect)obj).drawRect();
    }
    else if (obj instance Line)
    {
        ((Line)obj).drawLine();
    }
    else if (obj instance Circle)
    {
        ((Circle)obj).drawCircle();
    }
}
```

Mit etwas Programmiererfahrung erkennt man, dass das ein merkwürdiges Konstrukt ist und hier ein Indiz für den Einsatz von Polymorphie vorliegt. Dazu wandeln wir den Sourcecode in folgenden drei Schritten um:

1. Schnittstellenvereinheitlichung – gemeinsame Methode `draw()` einführen
2. Gemeinsamen Basistyp `BaseFigure` einführen
3. Anpassung der Aufrufstelle

Schritt 1: Schnittstellenvereinheitlichung

Wir suchen einen allgemeingültigen Namen für die jeweiligen Fälle, hier etwa `draw()`, und benennen die jeweiligen Methoden mithilfe des Basis-Refactorings REFACTOR –> RENAME um:

```
for (final Object obj : figures)
{
    if (obj instance Rect)
    {
        ((Rect)obj).draw();
    }
    else if (obj instance Line)
    {
        ((Line)obj).draw();
    }
    else if (obj instance Circle)
    {
        ((Circle)obj).draw();
    }
}
```

Nach diesem Renaming besitzen alle Klassen konsistente Methodennamen. Dadurch wird noch deutlicher, dass sich ein gemeinsamer Basistyp anbietet.

Schritt 2: Einführen eines gemeinsamen Basistyps

In diesem Schritt 2 führen wir einen gemeinsamen Basistyp ein, hier etwa eine abstrakte Klasse oder ein Interface namens `BaseFigure`:

```
public interface BaseFigure
{
    public abstract void draw();
}
```

Schritt 3: Anpassung der Aufrufstelle

Mit dem neu eingeführten Typ `BaseFigure` lässt sich unter Ausnutzung der Polymorphie die Aufrufstelle massiv vereinfachen, weil sämtliche Typprüfungen entfallen können. Statt des zuvor allgemeinsten Typs `Object` nutzen wir nun `BaseFigure` als Basis in der `for`-Schleife. Zudem nennen wir die Variable von `obj` in das aussagekräftigere `baseFigure` um. Dadurch ergibt sich folgendes Konstrukt:

```
for (final BaseFigure baseFigure : figures)
{
    baseFigure.draw();
}
```

Fazit

Die durch dieses Refactoring erzielte Lösung besitzt diverse positive Eigenschaften und erfüllt zudem einige SOLID-Prinzipien (vgl. Abschnitt 3.5.3):

- Sie ist objektorientiert und typsicher.
- Sie ist erweiterbar und folgt dem Open Closed Principle (OCP).
- Sie nutzt eine klare, schmale Schnittstelle und folgt dem Interface Segregation Principle (ISP).
- Die jeweilige Funktionalität wird in Objekten gekapselt, wodurch die Kohäsion hoch ist und Separation of Concerns eingehalten wird.
- Die Umsetzung basiert auf einer guten Abstraktion, sodass nicht die konkreten Ausprägungen oder gar deren Besonderheiten durchscheinen. Somit wird auch das Dependency Inversion Principle (DIP) erfüllt.

Hinweis: Bedingte Zuweisungen übersichtlich gestalten

Neben dem gerade gezeigten vermeintlichen Nachbau von Polymorphie, wie ich es auch schon in Abschnitt 3.1.3 diskutiert habe, findet man ab und an auch Programmzeilen ähnlich zu folgenden, die jeweils Zustandsänderungen abhängig von einer Variablenbelegung vornehmen:

```
String var1 = null;
String var2 = null;

if (cmdVar == value1)
{
    var1 = "a1";
    var2 = "b1";
}
else if (cmdVar == value2)
{
    // var1 unbelegt
    var2 = "b2";
}
else if (cmdVar == value3)
{
    var1 = "a3";
    // var2 unbelegt
}
else
{
    var1 = "";
    var2 = "";
}
```

Sofern die Variablen `var1` und `var2` semantisch zusammengehören, kann man diese zu einem PARAMETER VALUE OBJECT (vgl. Abschnitt 3.4.5) zusammenfassen und die jeweiligen Kombinationen in einer Lookup-Map speichern, wodurch die Wertebelegungen klarer und näher beieinander abgelegt werden können:

```
lookupMap.put(value1, new ValueVO("a1", "b1");
lookupMap.put(value2, new ValueVO(null, "b2");
lookupMap.put(value3, new ValueVO("a3", null);
```

Zugriffe lassen sich dann wie folgt realisieren und sogar elegant mit einem Default versehen. Dazu nutzen wir die mit Java 8 eingeführte Methode `getOrDefault()` aus dem Interface `Map<K,V>` wie folgt:

```
final ValueVO defaultValue = new ValueVO("", "")
final ValueVO values = lookupMap.getOrDefault(cmdVar, defaultValue);
var1 = values.first;
var2 = values.second;
```

17.5 Defensives Programmieren

In komplexeren Systemen mit vielen Komponenten und externen Aufrufen empfiehlt es sich, insbesondere an den System- bzw. Komponentengrenzen vorsichtig oder defensiv zu programmieren. Damit ist gemeint, dass man Eingaben validiert und korrekte Initialisierungen und Zustände überprüft.

17.5.1 Führe eine Zustandsprüfung ein

Diese Umbaumaßnahme dient dazu, eine korrekte Initialisierung von Attributen, die zum Einsatz der Klasse wichtig sind, sicherzustellen. Dabei nutzen wir das Refactoring LAGERE FUNKTIONALITÄT IN HILFSMETHODEN AUS (vgl. Abschnitt 17.4.10).

Beispiel

Im folgenden Beispiel sehen wir drei Methoden, die jeweils auf ein statisches Attribut `parameterAccessService` zugreifen. Nur nach erfolgreicher Initialisierung durch den Aufruf der `initialize()`-Methode kann diese Utility-Klasse korrekt arbeiten. Daher findet in allen Methoden jeweils eine Initialisierungsprüfung statt und es wird eine `IllegalStateException` ausgelöst, falls die benötigte Initialisierung noch nicht erfolgt ist:

```
public static final String getExternalNameServiceName()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly" +
                 " initialized. Use initialize() before any other method call.");

   return parameterAccessService.getValue("SYSPARAM_EXTERNAL_NAME_SERVICE");
}

public static final String getExternalORBHost()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly" +
                 " initialized. Use initialize() before any other method call.");

   return parameterAccessService.getValue("SYSPARAM_EXTERNAL_ORB_HOST");
}

public static final String getInternalORBHost()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly" +
                 " initialized. Use initialize() before any other method call.");

   return parameterAccessService.getValue("SYSPARAM_INTERNAL_ORB_HOST");
}
```

Je mehr Methoden solche identischen Zustandsprüfungen durchführen, desto mehr explodiert der Sourcecode und scheint nur aus der Prüfung der Initialisierung zu bestehen.

Schritt 1: Implementieren einer Prüfmethode

Zur Prüfung der Initialisierung faktorieren wir die Methode `checkParameterAccessServiceInitialized()` heraus:

```
private static void checkParameterAccessServiceInitialized()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly" +
                " initialized. Use initialize() before any other method call.");
}
```

Schritt 2: Einsatz der Prüfmethode

Der Einsatz der obigen Methode geschieht gemäß dem DRY-Prinzip und macht die einsetzenden Methoden besser lesbar:

```
public static final String getExternalNameServiceName()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_NAME_SERVICE");
}

public static final String getExternalORBHost()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_ORB_HOST");
}

public static final String getInternalORBHost()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_INTERNAL_ORB_HOST");
}
```

Fazit

Eine derartige zentrale Initialisierungsprüfung ermöglicht es, mehrfach gleichen oder ähnlichen Sourcecode zu vermeiden, und sorgt dadurch für eine bessere Lesbarkeit.

17.5.2 Überprüfe Eingabeparameter

Die Prüfung von Eingabeparametern hilft dabei, unerwartete Werte zurückzuweisen. Dadurch bleibt die Objektintegrität erhalten und eine Methodenausführung erfolgt nur unter sicheren Umgebungsbedingungen. Des Weiteren können wir unsere Software im Fehlerfall informativer gestalten. Als Reaktion auf ungültige Parameterwerte sollte man Exceptions auslösen, da diese im Gegensatz zu Rückgabewerten einem Aufrufer eine Menge an Informationen über eine mögliche Fehlerursache mitliefern können.

Zumindest an Systemgrenzen, aber auch für öffentliche Business-Methoden empfiehlt es sich, die eingehenden Parameter auf Gültigkeit zu prüfen. Auf diese Weise

vermeidet man, ungültige Werte ins Programm zu lassen und dadurch Merkwürdigkeiten oder Fehler in Berechnungen zu erhalten. Dabei finde ich folgende Analogie aus dem realen Leben hilfreich: Wenn die dreckigen Schuhe vor der Wohnungstür bleiben, muss man nicht andauernd den Boden der ganzen Wohnung säubern, wenn man mal im Wald spazieren war.[19]

Bei dieser Umbaumaßnahme lassen sich die Prüfung von Referenzparametern sowie von Parametern primitiver Datentypen unterscheiden. Für Letztere können wir gültige Werte leichter prüfen, für Referenzparameter ist dies – abgesehen von einer Prüfung der Referenz auf `null` – meistens etwas komplizierter. Da Referenzparameter zum Teil eine Ausnahme bei der Prüfung darstellen, beschränke ich hier die Diskussion auf primitive Datentypen. Der folgende Praxistipp geht auf die Behandlung von `null`-Werten ein.

Stilfrage: `null`-Werte explizit behandeln oder nicht?

Eine Parameterprüfung ist immer dann zwingend notwendig, wenn ansonsten der Fehler »verschleppt« würde oder es zu einem nicht sofort erkennbaren Fehlverhalten käme, etwa zu falschen Berechnungen. Wird direkt auf eine übergebene Referenzvariable zugegriffen, die normalerweise nicht `null` sein darf, so führt dies ohne Zutun zu einer `NullPointerException`. In derartigen Fällen kann auf eine explizite Prüfung eines solchen Parameters verzichtet werden – allerdings wird dann nicht deutlich, ob nur versehentlich keine Prüfung erfolgt. Deshalb prüfe ich aus Gründen der Nachvollziehbarkeit und Konsistenz in der Regel alle Parameter und behandle `null`-Referenzen. Zudem bevorzuge ich als Reaktion auf unerwartete `null`-Werte, explizit eine `IllegalArgumentException` zu werfen und weitere Informationen zur Fehlersituation im Text der Exception zu übergeben. Seit JDK 7 ist der Einsatz der Hilfsmethode `Objects.requireNonNull()` empfehlenswert.

Beispiel

Betrachten wir folgenden Konstruktor, in dem drei Eingabeparameter vom Typ `int` entgegengenommen und ungeprüft gleichnamigen Attributen zugewiesen werden:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final int registrationStrategy)
{
    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

[19] Oder: Wenn man schon an der Grenze Ausweise und Waren kontrolliert, dann muss man das nicht mehr im ganzen Land immer und überall wieder tun.

Tatsächlich sind dabei aber folgende Randbedingungen zu beachten:

- Der Wert für `minExecutions` darf nicht negativ sein, d. h., er muss mindestens den Wert 0 haben. Die maximale Anzahl an Ausführungen `maxExecutions` muss kleiner oder gleich einem nicht gezeigten Wert `MAX_EXECUTIONS` sein.
- Der Wert `maxExecutions` muss größer gleich dem Wert `minExecutions` sein.
- Als Übergabewerte für die Registrierungsstrategie sind nur folgende durch `int`-Konstanten definierte Werte zulässig:

```
public static final int REPLACE_OLD_OR_ADD_AS_LAST = 0;
public static final int ADD_AS_FIRST               = 1;
public static final int ADD_AS_LAST                = 2;
```

Schritt 1: Referenz- oder Bereichsprüfung

Eine einfache Bereichsprüfung, die zunächst keine detaillierten Hinweistexte zur besseren Nachvollziehbarkeit bereitstellt, hilft dabei, den in Abschnitt 16.3.8 vorgestellten BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN zu eliminieren:

```
public CommandExecutor(final int minExecutions, final int maxExecutions, final
      int registrationStrategy)
{
    if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
    {
        throw new IllegalArgumentException("parameter 'minExecutions' or " +
                                           "'maxExecutions' is out of valid range");
    }
    if (minExecutions > maxExecutions)
    {
        throw new IllegalArgumentException("parameter 'minExecutions' must be" +
                                           " <= 'maxExecutions'");
    }
    // Achtung: Hier korrekter, aber potenziell fehlerträchtiger und fragiler
    // Vergleich, weil das Ganze stark vom Mapping und Werten abhängig ist!
    if (registrationStrategy < REPLACE_OLD_OR_ADD_AS_LAST ||
        registrationStrategy > ADD_AS_LAST)
    {
        throw new IllegalArgumentException("parameter 'registrationStrategy'" +
                                           " is invalid");
    }

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Dieser erste Schritt gibt im Fehlerfall einen vagen Hinweis (auslösender Parameter) auf die Ursache (Fehlerbeschreibung). ***Allerdings sind sowohl die Fehlerbeschreibung als auch die Wertebereichsprüfung fragil***. Ziemlich unangenehm ist der Einsatz von Vergleichsoperatoren auf möglicherweise willkürlich gewählten Werten von Konstanten. Im Speziellen gilt dies hier für die Verwendung der Konstanten der Registrierungsstrategie. Diese hätten theoretisch auch durch beliebige andere Werte abgebildet werden

können. Dann wären die Vergleiche eventuell nicht mehr ausreichend oder korrekt gewesen, etwa wenn `ADD_AS_LAST` dem Wert `4` entspricht. In Abschnitt 16.1.3 diskutiert BAD SMELL: ZUSAMMENGEHÖRENDE KONSTANTEN NICHT ALS TYP DEFINIERT dadurch verursachte Probleme und mögliche Lösungen. Wir nutzen als Abhilfe das Refactoring WANDLE KONSTANTENSAMMLUNG IN `ENUM` UM (vgl. Abschnitt 17.4.12).

Schritt 2: Einsatz von Aufzählungen, wenn sinnvoll und möglich

Häufig bietet sich für semantisch zusammengehörende Konstanten der Einsatz eines `enum`-Aufzählungstyps oder des ENUM-Musters (vgl. Abschnitt 3.4.4) an. In diesem Fall definiert der Typ `RegistrationStrategy` die Konstanten und wird als Referenzparameter übergeben. Eine Wertebereichsprüfung erfolgt implizit durch die Angabe des Typs. Allerdings ist nun die Referenz auf `null` zu prüfen:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final RegistrationStrategy registrationStrategy)
{
   // ... zwei Prüfungen ausgelassen ...
   Objects.requireNonNull(registrationStrategy,
                          "parameter 'registrationStrategy' must not be null");

   this.minExecutions = minExecutions;
   this.maxExecutions = maxExecutions;
   this.registrationStrategy = registrationStrategy;
}
```

Schritt 3: Angabe von übergebenen Werten und von Gültigkeitsbereichen

Anschließend korrigieren wir die unpräzise Fehlermeldung, indem wir den übergebenen Wert in den Exception-Text aufnehmen. Dadurch liefern wir bereits einen guten Hinweis auf mögliche Fehlerursachen. Wir können die Informationen aber noch weiter verbessern, indem wir gegebenenfalls zusätzlich gültige Wertebereiche angeben:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final RegistrationStrategy registrationStrategy)
{
   if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
   {
      throw new IllegalArgumentException("parameter 'minExecutions'=" +
                minExecutions + " or 'maxExecutions'=" + maxExecutions +
                " is out of valid range: [" + 0 + " - " + MAX_EXECUTIONS + "]");
   }
   if (minExecutions > maxExecutions)
   {
      throw new IllegalArgumentException("parameter 'minExecutions'=" +
                minExecutions + " must be <= 'maxExecutions'=" + maxExecutions);
   }

   Objects.requireNonNull(registrationStrategy,
                          "parameter 'registrationStrategy' must not be null");

   this.minExecutions = minExecutions;
   this.maxExecutions = maxExecutions;
   this.registrationStrategy = registrationStrategy;
}
```

Schritt 4: Herausrefaktorieren von Prüfmethoden

Wie man sieht, wird der Sourcecode der Parameterprüfung immer umfangreicher und komplizierter. Hier dominiert die Wertebereichsprüfung bereits den eigentlichen Programmcode. Ähnliches habe ich im einleitenden Beispiel von Kapitel 16 für Logging-Code als negativ angesprochen. Ein Herausrefaktorieren von Prüfmethoden ist unter dem Aspekt der Wiederverwendbarkeit in anderen Klassen oftmals eher wenig sinnvoll, da die Prüfungen stark vom jeweiligen Kontext abhängig sind. Innerhalb der eigenen Klasse kann sich die Lesbarkeit durch diese Prüfmethoden allerdings enorm erhöhen:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final RegistrationStrategy registrationStrategy)
{
    assertExecutionsInValidRange(minExecutions, maxExecutions);
    assertMinExecutionsLessOrEqualToMax(minExecutions, maxExecutions);
    Objects.requireNonNull(registrationStrategy,
                           "parameter 'registrationStrategy' must not be null");

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Die beiden Prüfmethoden werden wie folgt definiert:

```
static void assertExecutionsInValidRange(final int minExecutions,
                                         final int maxExecutions)
{
    if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
              minExecutions + " or 'maxExecutions'=" + maxExecutions +
              " is out of valid range: [" + 0 + " - " + MAX_EXECUTIONS + "]");
    }
}

static void assertMinExecutionsLessOrEqualToMax(final int minExecutions,
                                                final int maxExecutions)
{
    if (minExecutions > maxExecutions)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
              minExecutions + " must be <= 'maxExecutions'=" + maxExecutions);
    }
}
```

Fazit

Die Prüfung von Eingabeparametern verhindert mögliche Inkonsistenzen und sorgt für einen gültigen Objektzustand. Im Fehlerfall werden aussagekräftige Fehlermeldungen produziert, die eine spätere Analyse und Fehlersuche erleichtern.

17.6 Fallstricke bei Refactorings

Nachfolgend wollen wir uns zum Abschluss dieses Kapitels eine unglückliche Kombination von temporären Abhängigkeiten (***Temporal Coupling***) und Seiteneffekten anschauen, die dazu führt, dass selbst vermeintlich sichere IDE-basierte Basis-Refactorings Fehler verursachen.

Initial sei nochmals darauf hingewiesen, dass Sie bei Refactorings immer schrittweise vorgehen sollten und idealerweise möglichst oft auch immer alle Unit Tests ausführen, um sicherzustellen, keine Fehler einzubauen. Zudem sollten Sie die von den IDEs angebotenen vielfältigen Refactorings nutzen und Free-Style-Umbauten eher vermeiden.

Gefahren durch Temporal Coupling und Seiteneffekte

Ich möchte Ihnen an einem einfachen Beispiel verdeutlichen, warum Seiteneffekte und temporäre Abhängigkeiten (Temporal Coupling) zwei Dinge sind, die Sie in Ihrer täglichen Arbeit behindern können. Zunächst machen es Seiteneffekte schwierig, Zustandsänderungen gut nachvollziehbar und kontrollierbar zu halten – insbesondere im Kontext von Multithreading. Temporal Coupling sorgt dafür, dass meist verdeckte Abhängigkeiten bestehen, die unerwartet Einfluss auf den Programmzustand und auf die Ausführungsreihenfolge haben. Beides ist schon für sich schlimm genug, erschwerend kommt jedoch hinzu, dass auch Aufräumarbeiten und Refactorings behindert werden. Nach dieser längeren Einleitung schauen wir uns drei kurze Methoden an, wobei das Temporal Coupling zwischen der Berechnung in `doSomething()` und dem Seiteneffekt in `someOtherStuff()` besteht:

```
int x = 2;
int y = 7;

private void affectedMethod()
{
    int result = doSomething(x,y);    // Berechnung basiert auf x und y

    someOtherStuff();                 // ändert x als Seiteneffekt

    if (result == 14)
    {
       System.out.println("expected 14");
       ...
    }
}

int doSomething(int x, int y)
{
    return x * y;
}

void someOtherStuff()
{
     x = 0;                           // unerwarteter Seiteneffekt ...
}
```

Bei der Belegung der Attribute `x` und `y` mit den Werten `2` und `7` wird für `result` der Wert `14` berechnet und somit kommt es zur Ausgabe:

```
expected 14
```

Inline von `result`

Nehmen wir an, wir wollten den Sourcecode klarer gestalten und würden dazu die künstliche Hilfsvariable `result` durch das Basis-Refactoring INLINE auflösen. Ohne Temporal Coupling und ohne Seiteneffekte im Sourcecode ist das Vorgehen absolut sicher. Schauen wir jedoch unter diesen ungünstigen Randbedingungen auf das Ergebnis der Transformation und deren Folgen:

```
int x = 2;
int y = 7;

private void affectedMethod()
{
    someOtherStuff();               // ändert x als Seiteneffekt

    if (doSomething(x,y) == 14)
    {
       System.out.println("expected 14");
       ...
    }
}
```

Würden wir diese Variante der Methode ausführen, so käme es nicht zur erwarteten Ausgabe des Werts 14. Vielmehr wird durch den Seiteneffekt in `someOtherStuff()` nun der Wert 0 berechnet, da jetzt der zwischenzeitlich geänderte Werte von `x` in die Berechnung eingeht. Demnach haben wir einen Applikationsfehler durch ein eigentlich wasserdichtes Basis-Refactoring eingeführt. Im Beispiel scheint dies nicht so schwerwiegend zu sein. Wenn wir uns jedoch vorstellen, dass statt einer Konsolenausgabe etwa eine Bestellung oder Flugbuchung ausgelöst würde oder eben nicht, wäre das wohl ein dramatischer Fehler.

Fazit

An einem einfachen Beispiel habe ich mögliche Probleme durch Seiteneffekte und Temporal Coupling dargestellt. Vermeiden Sie Seiteneffekte und dokumentieren Sie zeitliche Abhängigkeiten von Methoden, sodass spätere Bearbeiter des Sourcecodes nicht in solche Fallen, wie die gerade präsentierte, tappen.

17.7 Weiterführende Literatur

Das umfangreiche Themengebiet Refactorings lässt sich in einem Kapitel nicht umfassend behandeln. Ich hoffe aber, Ihr Interesse geweckt zu haben. Empfehlenswerte Bücher, die sich ausführlich mit dem Thema Refactorings beschäftigen, sind:

- »**Refactoring: Improving the Design of Existing Code**« von Martin Fowler [20]
  Einige Tricks und Kniffe, Sourcecode zu verbessern, lernt man von Kollegen oder durch Erfahrung. Martin Fowler fasst dieses Wissen in dem genannten Buch zusammen und stellt ein systematisches Vorgehen zur Sourcecode-Transformation vor.
- »**Refactorings to Patterns**« von Joshua Kerievsky [45]
  Dieses Buch verknüpft das Standardwerk zu Refactorings von Martin Fowler mit den Ideen der Entwurfsmuster.
- »**Refactorings in großen Softwareprojekten**« von Stefan Roock und Martin Lippert [63]
  Neben den Refactorings, die einfache Sourcecode-Transformationen beschreiben, kann man Refactorings auch auf Architektur- bzw. Designebene übertragen. Dieses Buch stellt einige Architekturprobleme, Refactorings und Tools vor.
- »**Working Effectively With Legacy Code**« von Michael Feathers [18]
  In diesem Buch werden Strategien aufgezeigt, wie man mit Altlasten behafteten Sourcecode überarbeiten kann, ohne in einem Wartungsalbtraum zu enden.

18 Entwurfsmuster

Vielfach erprobte und regelmäßig eingesetzte allgemeingültige Verfahren zur Lösung eines Entwurfsproblems werden ***Entwurfsmuster*** oder auch ***Designpattern*** genannt. Populär wurden diese durch das Buch »Design Patterns – Elements of Reusable Object Oriented Software« von der sogenannten Gang of Four (GoF): Gamma, Helm, Johnson, Vlissides [22]. Bereits Mitte der 90er-Jahre löste dieses Buch einen Run aus, der bis heute anhält. Viele Bücher sind zu diesem Thema erschienen, ohne jedoch das Original als *die* Referenz zu verdrängen.

Dieses Kapitel stellt einige in der Praxis häufig eingesetzte Entwurfsmuster vor, zeigt aber auch mögliche Probleme auf und entkräftet den Glauben, allein durch Einsatz von Entwurfsmustern erfolgreiche Softwareentwicklung betreiben zu können. Naheliegende Fehlverwendungen werde ich als ***Anti-Pattern*** beschreiben und erläutern.

Die von der GoF vorgenommene Gliederung in die drei Kategorien Erzeugungsmuster, Strukturmuster und Verhaltensmuster übernehme ich in diesem Buch. Die Vorstellung eines Musters beginnt mit einer kurzen Beschreibung und Motivation für dessen Einsatz. Daran schließt sich eine Darstellung der zugrunde liegenden Struktur in Form eines UML-Diagramms sowie eine Beschreibung des jeweiligen Musters anhand vereinfachter Praxisbeispiele mit abschließender Bewertung an.

Abschnitt 18.1 beschreibt, wie der Einsatz von Erzeugungsmustern den Konstruktionsprozess von Objekten vereinfachen kann. Die in Abschnitt 18.2 vorgestellten Strukturmuster helfen, Funktionalitäten flexibel erweitern und anpassen zu können. Verhaltensmuster werden in Abschnitt 18.3 behandelt. Durch ihren Einsatz strukturiert und vereinfacht man komplexe Abläufe oder Interaktionen zwischen Objekten.

Design mit Entwurfsmustern

Die durch Entwurfsmuster beschriebenen prototypischen Beispielumsetzungen helfen, ein Problem auf eine dokumentierte Art und Weise zu lösen. Die Verwendung von Entwurfsmustern kann zu qualitativ hochwertiger Software führen – es kommt jedoch nicht automatisch zu einem guten Design und sinnvollen Lösungen. Zwar erreicht man häufig größere Flexibilität, allerdings entsteht auf Sourcecode-Ebene oft weitere Komplexität. Der Einsatz von Entwurfsmustern sollte daher immer auf die jeweilige Situation abgestimmt werden.

Jedes Entwurfsmuster trägt einen eindeutigen Namen und beschreibt eine Lösungsidee für ein Entwurfsproblem. Dadurch fällt die Kommunikation von Designentschei-

dungen unter Entwicklern leichter. Es entsteht eine ***Entwurfssprache***, die dabei hilft, in Diskussionen länger auf der verständlichen Ebene von Konzepten und des Designs mögliche Alternativen besprechen zu können, ohne sich mit Implementierungsdetails beschäftigen zu müssen. Wollte man früher einem anderen Softwareentwickler eine Realisierung beschreiben, so waren dafür viele Worte und Zeichnungen nötig. Etwa: Diese Klasse ist so entworfen, dass sie nur einmal instanziiert werden kann, bzw. diese Klasse ist ein Datenbehälter und definiert für jedes Attribut öffentliche `get()`- und `set()`-Methoden. Heutzutage reicht es, z. B. zu sagen, diese Klasse ist als SINGLETON oder als VALUE OBJECT entworfen. Eine ausführliche Darstellung der Vorteile einer gemeinsamen Designsprache finden Sie in dem ausgezeichneten Buch »Design Patterns Explained« [66] von Alan Shalloway und James R. Trott.

Beispielapplikation: Grafischer Editor

In diesem Kapitel werden zur Verdeutlichung einige Entwurfsmuster im Kontext einer Beispielanwendung vorgestellt. Pro Entwurfsmuster wird jeweils ein Aspekt der im Folgenden beschriebenen Anforderungen an einen grafischen Editor abgedeckt.

Anforderungen Es sollen Zeichenfunktionen für Figuren, wie Linien, Rechtecke und Kreise, angeboten werden. Zudem ist die Anzeige von Bildern gefordert. Die zu erstellende Applikation soll über Menüs, Kontextmenüs und Toolbars gesteuert werden. Ein erster Skizzenentwurf der Bedienoberfläche kann neben Bleistift und Papier auch mithilfe eines Zeichenprogramms erstellt werden. Häufig ist Letzteres aber mühseliger als eine Freihandskizze. Für Prototyp-Entwürfe von GUIs kann ich das Tool Balsamiq Mockups empfehlen, das als Demoversion unter `http://www.balsamiq.com/` verfügbar ist. Damit entstand der in Abbildung 18-1 dargestellte Entwurf.

Abbildung 18-1 *UI-Entwurfskizze der Beispielapplikation*

Auf der Arbeitsfläche können sowohl Zeichenoperationen ausgeführt als auch Bilder eingefügt werden. Als Layouthilfe sind ein Lineal sowie ein Raster gewünscht. Als Randbedingung sollen alle Aktionen relativ zügig ausgeführt werden bzw. bei länger andauernden Aktionen ein paralleles Weiterarbeiten oder der Abbruch einer Aktion möglich sein. Obwohl diese letzten beiden Anforderungen eigentlich natürlich sind und für nahezu jede interaktive Anwendung zutreffen, werden diese in diversen Applikationen jedoch nur unzureichend umgesetzt.

Entwurfsideen Die zu zeichnenden Elemente, also die grafischen Figuren, wollen wir einheitlich behandeln. Daher definieren wir ein gemeinsames Interface sowie eine abstrakte Basisklasse, wie dies in Abschnitt 3.2 vorgestellt wurde.

Wir werden zur Konstruktion des grafischen Editors folgende Muster[1] nutzen:

- **BEFEHL (COMMAND)** – In Menüs und Toolbars sind vielfach Aktionen mit gleicher Funktionalität definiert. Diese Aktionen, wie etwa Laden oder Speichern, können über einen Menüeintrag, Shortcut, Toolbar-Button usw. ausgelöst werden. Um diese Funktionalität nicht mehrfach auszuprogrammieren, also um Sourcecode-Duplikation zu vermeiden, werden die Aktionen in Form von speziellen Befehlsobjekten implementiert. Diese werden jeweils an der aufrufenden Stelle erzeugt und ausgeführt. Eine Undo-Funktionalität wird durch Einsatz dieses Musters nachträglich leichter realisierbar.
- **BEOBACHTER (OBSERVER)** – Änderungen im Modell werden an verschiedene Views propagiert und führen zu Modifikationen in der Übersichtsdarstellung und auf der Zeichenfläche.
- **ERZEUGUNGSMETHODE** und **FABRIKMETHODE (FACTORY METHOD)** – Beide helfen beim Erzeugen von Objekten.
- **ITERATOR** – Dieses Muster verwenden wir häufig, etwa zum Durchlaufen des Datenmodells beim Zeichnen verschiedener Figuren.
- **MODEL-VIEW-CONTROLLER** (MVC)[2] – Es werden Views (Arbeitsfläche und Übersichtsdarstellung) mit verschiedenen Ansichten gleicher Modelldaten realisiert.
- **PROTOTYP (PROTOTYPE)** – Dieses Muster hilft dabei, Copy-Paste-Operationen zu erleichtern.
- **SCHABLONENMETHODE (TEMPLATE METHOD)** – Das Zeichnen auf der Arbeitsfläche erfolgt nach einem genau definierten Algorithmus, der zunächst Lineale sowie Gitterstützpunkte und anschließend Figuren zeichnet. Durch die Basisklasse wird die Infrastruktur (Lineal und Raster) gezeichnet, jedoch ist dort keine Methode zum Zeichnen der Figuren implementiert. Dies geschieht erst in einer Spezialisierung.

[1] Englische Schreibweise ist in Klammern angegeben, falls sie von der deutschen abweicht.
[2] MVC ist im strengen Sinn kein Entwurfsmuster, sondern ein Architekturmuster.

18.1 Erzeugungsmuster

Erzeugungsmuster dienen dazu, einen potenziell aufwendigen Konstruktionsprozess von Objekten zu vereinfachen. Wenn Methoden statt Konstruktoren zum Erzeugen von Objekten verwendet werden, so entspricht dies den Ideen der Muster ERZEUGUNGSMETHODE bzw. FABRIKMETHODE. Auch mithilfe des Musters ERBAUER kann man die Objektkonstruktion einfacher gestalten. Durch ein SINGLETON wird sichergestellt, dass lediglich eine Instanz einer Klasse erzeugt und ein definierter Zugriffspunkt darauf angeboten wird. Mit einem PROTOTYP können Objekte leicht aus einem definierten Mustersatz durch Kopie und Parametrierung erstellt werden.

18.1.1 Erzeugungsmethode

Beschreibung und Motivation

Man kann sich die ERZEUGUNGSMETHODE etwa wie eine Bestellung in einem Restaurant vorstellen. Man sagt, welches Gericht man essen möchte, nennt jedoch nicht jede Zutat. Auch die Art der Zubereitung bleibt verborgen.

Auf ein Programm übertragen bedeutet dies Folgendes: Wenn der Konstruktionsprozess komplex oder fehleranfällig ist, weil beispielsweise viele Parameter übergeben werden müssen, so wird eine Erzeugung durch einen Konstruktoraufruf von außen vermieden. Stattdessen wird die Klasse selbst für die Objekterzeugung verantwortlich gemacht und stellt zur Objektkonstruktion eine spezielle, statische Konstruktionsmethode, die Erzeugungsmethode, bereit. Um Objekterzeugungen von außen ohne Aufruf der Erzeugungsmethode sicher zu verhindern, dürfen lediglich private Konstruktoren definiert werden.

Struktur

Betrachten wir eine vereinfachte Klasse `DrawingPad`, die entweder als Applet oder als Applikation gestartet werden kann. Dies wird durch den Wert des Konstruktorparameters `runAsApplet` entschieden. Die Signatur des Konstruktors sieht wie folgt aus:

```
DrawingPad(final String title, final boolean runAsApplet,
           final JApplet applet, final ConsoleParams appParams)
```

Die beiden Parameter `applet` bzw. `appParams` sind jeweils nur in einem der beiden Modi sinnvoll zu belegen. Für Applikationen bleibt daher der `applet`-Parameter unbelegt (`null`). Für Applets wird für `appParams` der Wert `null` übergeben. Damit käme es beispielsweise zu folgenden Konstruktoraufrufen:

```
new DrawingPad("Application", false, null, consoleParams);
new DrawingPad("Applet", true, applet, null);
```

Betreiben wir eine kurze Analyse, um einige Probleme bei der Konstruktion eines `DrawingPad`-Objekts aufzudecken.

1. Wirklich lesbar und verständlich sind die gezeigten Konstruktoraufrufe mit `null`-Werten und den booleschen Literalen `true` und `false` nicht.
2. Existieren optionale Parameter, so steigt die Gefahr, an falscher Stelle `null`-Werte zu übergeben und dadurch unerwartete Fehlinitialisierungen vorzunehmen.
3. Es ist keine Kontrolle möglich, dass alle Werte konsistent übergeben werden. Im Speziellen können sowohl eine `JApplet`-Referenz als auch eine Referenz auf `ConsoleParams` übergeben werden. Zudem muss darauf geachtet werden, dass der boolesche Wert `runAsApplet` passend gesetzt ist. Man kann sich schnell folgende Fragen stellen: Welcher Wert soll Vorrang haben? Sollen beide Werte ausgewertet werden? Und wie drückt man Letzteres durch einen booleschen Parameter aus?
4. Der boolesche Wert `runAsApplet` ist redundant, da er sich anhand der Aussage `applet != null` erschließen lässt. Schlecht an dieser Redundanz ist zweierlei: Erstens ist eine widersprüchliche Eingabe möglich, da `runAsApplet` als `false` in Kombination mit einer gültigen `JApplet`-Referenz übergeben werden kann. Gleiches gilt für die Kombination mit einer `ConsoleParams`-Referenz und dem Wert `true`. Zweitens wird dieses Implementierungsdetail auch noch in der öffentlichen Schnittstelle für alle möglichen Aufrufer sichtbar. Eine nachträgliche Korrektur würde somit Änderungen an allen verwendenden Stellen bedingen.

Als Verbesserung bietet sich an, zwei Erzeugungsmethoden zu nutzen: Eine für die Konstruktion von Applets und eine für Applikationen. Diese beiden Methoden bekommen die sprechenden Namen `createAsApplet()` und `createAsApplication()`. Sie werden wie folgt realisiert:

```
private static final boolean RUN_AS_APPLET = true;

public static DrawingPad createAsApplet(final String title,
                                        final JApplet applet)
{
    final ConsoleParams NO_APP_PARAMS = null;
    return new DrawingPad(title, RUN_AS_APPLET, applet, NO_APP_PARAMS);
}

public static DrawingPad createAsApplication(final String title,
                                             final ConsoleParams appParams)
{
    final JApplet NO_APPLET = null;
    return new DrawingPad(title, !RUN_AS_APPLET, NO_APPLET, appParams);
}
```

Anhand dieses Beispiels lernen wir eine weitere, nützliche Technik kennen: Statt Magic-Konstanten[3] zu benutzen, werden hier für die Werte `null` und `true` jeweils Konstanten mit sprechendem Namen (hier `NO_APP_PARAMS` bzw. `RUN_AS_APPLET`) definiert. Dadurch erhöht sich die Lesbarkeit und die Intention wird verdeutlicht. Durch Einsatz der Erzeugungsmethoden benötigt man zudem weniger Übergabeparameter für

[3] Verallgemeinerung von Magic Numbers: Literale primitiver Typen, Strings und `null`-Referenzen.

Aufrufe durch Klienten – die zur Objektkonstruktion intern erforderliche Anzahl bleibt natürlich gleich. Allerdings werden die Details der ausgelassenen Parameter und der Redundanz in der Klasse selbst versteckt. Eine spätere Korrektur wird unabhängig von externen Aufrufern möglich.

Erweiterungen

Der Einsatz einer Erzeugungsmethode ermöglicht häufig, die Klasse gemäß dem in Abschnitt 3.4.2 vorgestellten IMMUTABLE-Muster unveränderlich zu machen und dementsprechend Zustandsänderungen nach der Konstruktion zu verhindern. Selbst wenn dies nicht vollständig gelingt, kann eine Reduktion der veränderlichen Attribute auf ein tatsächlich benötigtes Minimum die Übersichtlichkeit fördern. Eine schrittweise Anleitung zum Reduzieren variabler Anteile einer Klasse beschreibt das Refactoring MINIMIERE VERÄNDERLICHE ATTRIBUTE in Abschnitt 17.4.2.

Verwendet man eine Erzeugungsmethode, so ist es bei Bedarf leicht möglich, eine Fehlerbehandlung zu integrieren. Man kann dadurch gewährleisten, dass ein Objekt immer vollständig initialisiert ist, bevor es verwendet wird. Treten Fehler während der Konstruktion auf, kann alternativ der Wert `null` oder ein NULL-OBJEKT (vgl. Abschnitt 18.3.2) zurückgeliefert werden. Seit Java 8 bietet es sich an, Instanzen der Klasse `Optional<T>` (vgl. Abschnitt 6.5) als Rückgabewert für potenziell fehlgeschlagene Erzeugungsmethoden zu nutzen. Dabei modelliert der Wert `Optional.empty` den Fehlerfall.

Stellt ein Fehler allerdings eine außergewöhnliche Situation dar, sollte dies statt mit einem Rückgabewert besser mit einer Exception ausgedrückt werden. Eine Diskussion verschiedener Arten der Fehlerbehandlung finden Sie in Abschnitt 5.6.

Bewertung

Der Einsatz einer ERZEUGUNGSMETHODE bewirkt Folgendes:

+ **Lesbarkeit** – Die Details des Konstruktionsprozesses werden versteckt, was für mehr Lesbarkeit sorgt: Die Signatur einer Erzeugungsmethode ist in der Regel kürzer als die Parameterliste des Konstruktors. Dies ist möglich, weil lediglich einige der Parameter des Konstruktors entgegengenommen und die restlichen beim Aufruf des Konstruktors mit Defaultwerten belegt werden. Zudem kann man für eine Erzeugungsmethode einen sprechenden Namen wählen, der die erzeugten Objekte besser charakterisiert als der Konstruktor mit dem Namen der Klasse.

+ **Konstruktionssicherheit** – Man kann garantieren, dass vollständig und korrekt initialisierte Objekte erzeugt werden. Dazu werden zu Beginn einer Erzeugungsmethode alle Parameter vor der Übergabe und Konstruktion des Objekts validiert. Dadurch wird es möglich, die Instanzerzeugung abzusichern oder zu erleichtern und angemessen auf Fehlersituationen zu reagieren. Weiterhin kann man eine Art Transaktion bei der Objekterzeugung erreichen: Ein Objekt wird entweder vollständig erzeugt oder es wird im Fehlerfall kein Objekt, sondern der Wert `null`, ein NULL-OBJEKT oder `Optional.empty` zurückgeliefert. Daher leistet eine Erzeugungsmethode bei der Behandlung von Fehlern im Konstruktionsprozess gute Dienste.
o **Mehraufwand** – Es ist minimal mehr Sourcecode erforderlich.

18.1.2 Fabrikmethode (Factory Method)

Beschreibung und Motivation

Die Idee hinter dem Muster FABRIKMETHODE ist ähnlich einer Produktion in einer realen Fabrik, die aufgrund einer Bestellung gewünschte Dinge produzieren kann. Bei einer solchen Bestellung sind lediglich die Artikelnummern oder Bezeichnungen der Teile, nicht aber die konkreten Ausprägungen und Realisierungen bekannt. Es findet demnach analog zum Muster ERZEUGUNGSMETHODE eine Kapselung der Objekterzeugung statt: Objekte werden nicht direkt per Konstruktoraufruf erzeugt, sondern dieser Vorgang wird an ein spezielles Objekt, eine sogenannte Fabrik, delegiert. Die dort definierten Fabrikmethoden erzeugen Objekte verschiedenen Typs.

Der Unterschied zum Muster ERZEUGUNGSMETHODE, das den Konstruktionsprozess einer speziellen Klasse vereinfacht, liegt darin, dass beim Muster FABRIKMETHODE eine andere Klasse, die Fabrik, die Konstruktion von Objekten eines gewünschten konfigurierbaren Typs durchführt.

Struktur

Es wird eine Fabrikklasse `Factory` mit einer Methode `createProduct(ProductType)` zum Erzeugen von Objekten definiert. Dort wird anhand eines Parameters `productType` entschieden, welcher konkrete Typ erzeugt wird. Der gewünschte zu erzeugende Typ wird dabei entweder, wie in diesem Fall, als Parameter übergeben oder aus einer Konfiguration gelesen. Damit überhaupt verschiedene Produkte von einer Methode erzeugt werden können, müssen diese Produkte ein gemeinsames Interface `IProduct`, eine gemeinsame Basisklasse `AbstractProduct` oder beides besitzen. Klienten arbeiten immer mit einer dieser Abstraktionen des konkreten Produkts, wodurch es automatisch zu einer besseren Kapselung und loseren Kopplung zum verwendenden Applikationscode kommt. Abbildung 18-2 zeigt eine mögliche Realisierung als UML-Diagramm.

Abbildung 18-2 *Fabrikmethode*

Beispiel

Die Fabrikmethode `createGraphicsElement(...)` wird in einer Klasse `MenuActionHandler` definiert. Dort wird anhand des übergebenen `enum`-Werts entschieden, von welchem Typ eine neue Instanz erzeugt wird. Abbildung 18-3 zeigt eine mögliche Realisierung eines Teils unserer Grafikanwendung als UML-Diagramm.

Abbildung 18-3 *Fabrikmethode konkret*

Abhängig von dem übergebenen Aufzählungswert `type` wird ein Objekt eines korrespondierenden grafischen Elements erzeugt, beispielsweise `ImageGraphicsElement`. Das zeigt der folgende Sourcecode-Ausschnitt:

```
public static AbstractGraphicsElement createGraphicsElement(final
      GraphicsElementTypes type, final Point insertPos)
{
    switch (type)
    {
        case IMAGE:
            return new ImageGraphicsElement(insertPos.x, insertPos.y);
        case PDF:
            return new PdfGraphicsElement(insertPos.x, insertPos.y);

            // ...
    }
    throw new IllegalStateException("Unexpected 'GraphicsElementTypes' '" +
                                    type + "'");
}
```

Bewertung

Der Einsatz des Musters FABRIKMETHODE besitzt durch die Ähnlichkeit zum Muster ERZEUGUNGSMETHODE nahezu dieselben Implikationen, weshalb diese in der folgenden Aufzählung nur kurz genannt werden. Als Erweiterung wird beim Muster FABRIKMETHODE allerdings nicht nur der Konstruktionsprozess von Klassen vereinfacht, sondern der Konstruktionsprozess an sich gekapselt: Die Konstruktion von Objekten eines gewünschten, konfigurierbaren Typs wird durch eine externe Klasse realisiert. Es findet somit eine Abstraktion von den konkret erzeugten Klassen statt.

Der Einsatz einer FABRIKMETHODE hat folgende Auswirkungen:

+ **Lesbarkeit** – Die Details des Konstruktionsprozesses werden versteckt, was für mehr Lesbarkeit sorgt.
+ **Abstraktion** – Der konkret erzeugte Objekttyp kann vor dem Aufrufer versteckt werden, indem lediglich eine Referenz auf ein Interface oder eine abstrakte Klasse des erzeugten Objekttyps zurückgegeben wird. Dies verstärkt zudem den Effekt der Kapselung und führt zu einer loseren Kopplung.
+ **Kapselung** – Die Kapselung verstärkt sich: Ein Klient nutzt lediglich eine Schnittstelle zur Erzeugung, wodurch die konkrete Realisierung einer Fabrikklasse, der Erzeuger, ausgetauscht werden kann. Durch die zuvor erwähnte Abstraktion vom konkret erzeugten Objekt können unterschiedliche Fabrikklassen verschiedene konkrete Typen eines gemeinsamen Basistyps zurückliefern.
+ **Konstruktionssicherheit** – Durch eine Konsistenzprüfung können garantiert vollständig initialisierte Objekte erzeugt werden bzw. Fehler entsprechend korrigiert oder behandelt werden.
o **Mehraufwand** – Es entsteht ein wenig mehr Sourcecode, da hier zumindest eine Fabrikklasse und eine Basisklasse für die Produktklassen benötigt werden.
o **Mehr Komplexität** – Die zu realisierenden Klassen erhöhen die Komplexität geringfügig.

18.1.3 Erbauer (Builder)

Beschreibung und Motivation

Das Vorgehen beim ERBAUER-Muster lässt sich mit der Bestellung einer Pizza oder eines Autos vergleichen. Auf einer Bestellliste kreuzt man aus vielen verschiedenen Bestandteilen und Extras genau diejenigen an, die man bekommen möchte. Man gibt also eine spezielle Bestellung auf und konfiguriert sich damit sein Objekt mit den gewünschten Eigenschaften. Erst nach Abschluss der Auswahl wird mit der eigentlichen Herstellung begonnen.

Das ERBAUER-Muster kann immer dann sinnvoll eingesetzt werden, wenn ein zu erzeugendes Objekt viele optionale oder komplex zu konstruierende Bestandteile besitzt. Auch wenn zur Objektkonstruktion benötigte Werte iterativ ermittelt werden, lässt sich eine Folge von `set()`-Methoden zur Objektinitialisierung durch den Einsatz des ERBAUER-Musters vermeiden. Dadurch kann man meistens einen gültigen Objektzustand viel leichter sicherstellen (vgl. Abschnitt 3.1.5).

Im Unterschied zum Muster FABRIKMETHODE, bei dem eine abstrakte Bezeichnung, beispielsweise eine Artikelnummer, das zu erzeugende Produkt beschreibt, liegt der Fokus beim Einsatz des ERBAUER-Musters darauf, die einzelnen Bestandteile eines Objekts auf einfache Weise zu spezifizieren. Dabei werden zudem die Ideen des ERZEUGUNGSMETHODE-Musters aufgegriffen, die eine sichere Konstruktion adressieren. Betrachten wir dies nun etwas konkreter.

Struktur

Eine Klasse `Builder` definiert eine Fabrikmethode `create()` oder `build()`, die später zur Konstruktion des gewünschten `Product`-Objekts aufgerufen werden muss. Um die Erzeugung von Objekten mittels Konstruktoraufruf durch externe Klassen zu verhindern, definiert man den Konstruktor der `Product`-Klasse mit der Sichtbarkeit `private`. Dies zeigt Abbildung 18-4.

Abbildung 18-4 *Erbauer*

Die Klasse `Builder` sammelt zunächst alle Informationen, um die zu konstruierenden Teile zu ermitteln. Das Auswählen einer Eigenschaft erfolgt durch Aufruf spezieller Methoden mit aussagekräftigem Namen. Nachdem alle gewünschten Eigenschaften spezifiziert wurden, wird die Erstellung des eigentlichen Produkts veranlasst, wodurch auch eine Konstruktion der zu den gewünschten Eigenschaften korrespondierenden Teile erfolgt. Dies geschieht durch Aufruf der `create()`-Methode. Folgendes Sourcecode-Fragment zeigt das zuvor Beschriebene:

```
new Builder().chooseOption1().chooseOption2().create();
```

In diesem Fall ergibt sich durch die Hintereinanderausführung von Methoden eine lesbare und kurze Schreibweise. Voraussetzung dazu ist, dass die Auswahlmethoden jeweils eine Referenz auf die `Builder`-Instanz selbst zurückliefern.

Tipp: Hintereinanderausführung von Methoden

In vielen Situationen ist eine Hintereinanderausführung von Methoden zu vermeiden, da man sich nicht auf korrekte Rückgabewerte verlassen kann und die Verkettung bei `get()`-Methoden eher unleserlich wird.

Für die Methoden im ERBAUER mit sprechenden Namen gilt diese Kritik nicht, da hier garantiert ein gültiges Objekt zurückgeliefert wird und eine Verkettung häufig die Lesbarkeit sogar erhöhen kann.

Beispiel

Im folgenden Beispiel wird die Klasse `PizzaBuilder` genutzt, um die Eigenschaften zweier verschiedener `Pizza`-Objekte zu spezifizieren und diese anschließend zu konstruieren:

```
public static void main(final String[] args)
{
   final PizzaBuilder builder = new PizzaBuilder();
   builder.mitExtraSardellen();
   System.out.println("Normale Pizza mit extra Sardellen: " + builder.create());

   final PizzaBuilder builder2 = new PizzaBuilder();
   builder2.mitSalami().small().bitteBeachten("Ohne Mais!");
   System.out.println("Kleine Salami-Pizza ohne Mais: " + builder2.create());
}
```

Listing 18.1 *Ausführbar als* '**BUILDEREXAMPLE**'

Die (statische innere) Klasse `PizzaBuilder` ist ähnlich zum Muster VALUE OBJECT (vgl. Abschnitt 3.4.5) realisiert und enthält die Wunschliste der Optionen:

```
public static final class PizzaBuilder
{
    // Defaultwerte, gelten wenn keine korrespondierende Methode aufgerufen wird
    boolean mitSalami         = false;
    boolean mitExtraSardellen = false;
    String  info              = "";
    Size    size              = Size.MEDIUM;

    public PizzaBuilder mitSalami()
    {
        this.mitSalami = true;
        return this;
    }

    public PizzaBuilder mitExtraSardellen()
    {
        this.mitExtraSardellen = true;
        return this;
    }

    public PizzaBuilder small()
    {
        this.size = Size.SMALL;
        return this;
    }

    public PizzaBuilder bitteBeachten(final String info)
    {
        this.info = info;
        return this;
    }

    public Pizza create()
    {
        return new Pizza(this);
    }
    // ...
```

Bewertung

Der Einsatz des ERBAUER-Musters hat folgende Auswirkungen:

+ **Lesbarkeit und Vereinfachung** – Die Details des Konstruktionsprozesses werden versteckt, was für mehr Lesbarkeit sorgt: Es werden lediglich gewünschte Eigenschaften spezifiziert. Die eigentliche Konstruktion wird intern geregelt und bleibt verborgen. Zudem kann man sprechende Methodennamen wählen, die die Eigenschaften des zu erzeugenden Objekts besser beschreiben als die Namen von Parametern.
+ **Konstruktionssicherheit** – Es lassen sich vollständig und korrekt initialisierte Objekte erzeugen: Die Parametrierung erfolgt ausschließlich über einige Methoden, die die korrespondierenden Attribute setzen und später als Eingabewerte für den Konstruktor des zu erzeugenden Produkts dienen.

+ **Unterstützung optionaler Attribute** – Der Umgang mit vielen optionalen Attributen wird erleichtert: Defaultwerte müssen nicht explizit gesetzt bzw. im Konstruktor übergeben werden.
- **Mehr Komplexität** – Der Einsatz führt zu mehr Komplexität und Sourcecode, da gewünschte Wertebelegungen von Attributen über den Aufruf von Methoden spezifiziert werden müssen. Der für Aufrufer verborgene Konstruktionsprozess sorgt für mehr Komplexität innerhalb der Klassen des ERBAUER-Musters.

18.1.4 Singleton

Beschreibung und Motivation

Möchte man sicherstellen, dass höchstens eine Instanz einer bestimmten Klasse erzeugt werden kann, so bietet sich die Verwendung des SINGLETON-Musters an. Außerdem wird dadurch ein globaler Zugriffspunkt auf die Instanz bereitgestellt.

Beispiele für Einsatzgebiete sind jede Art von zentraler Registrierung: Wenn ein Programm über Plugins erweiterbar ist, werden diese zweckmäßigerweise über einen zentralen Plugin-Manager verwaltet. Ebenso werden Fabrikklassen häufig gemäß dem SINGLETON-Muster implementiert, da man Klienten nicht zusätzlich die Erzeugung von Fabrikklassen auferlegen möchte.

Struktur

Damit lediglich eine Instanz erzeugt werden kann, ist es wichtig, den Konstruktionsprozess in der Klasse selbst zu kapseln und diese für die Verwaltung ihres einzigen Exemplars zuständig zu machen. Es kommt eine spezielle Form des ERZEUGUNGSMETHODE-Musters zum Einsatz, für die folgende Realisierungsanforderungen existieren:

1. Um eine Objekterzeugung außerhalb der Klasse zu verhindern, verwenden wir einen privaten Konstruktor.
2. Zur Speicherung der einzigen Instanz wird ein statisches Attribut `INSTANCE` verwendet. Der Zugriff auf dieses erfolgt ausschließlich über eine statische Zugriffsmethode, typischerweise `getInstance()` genannt.

Die Umsetzung dieser Forderungen ist in Abbildung 18-5 als Klassendiagramm dargestellt. Dabei werden stellvertretend für beliebige Attribute einer `Singleton`-Klasse zwei Attribute `attribute1` und `attribute2` in die Modellierung aufgenommen.

Abbildung 18-5 *Singleton*

Beispiel

Ein erster intuitiver Entwurf, den man häufig so oder ähnlich in Sourcecode findet, sieht wie folgt aus:

```
public final class BadSingleton
{
    private static BadSingleton INSTANCE = null;

    private int attribute1;
    private int attribute2;

    // ACHTUNG: SCHLECHT !!!
    public static BadSingleton getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new BadSingleton();
        }
        return INSTANCE;
    }

    private BadSingleton()
    {
    }

    public void businessMethod1() { /* ... */ }
    public void businessMethod2() { /* ... */ }
}
```

Bei einem flüchtigen Blick darauf scheint diese Umsetzung schon ganz gut zu sein. Aber wie bereits in Jon Bentleys »Perlen der Programmierkunst« [3] erwähnt, ist die erste Idee meistens nicht die beste.

In Singlethreading-Umgebungen ist die obige Implementierung akzeptabel. Probleme offenbaren sich beim Einsatz in Multithreading-Umgebungen: Wie bereits in Kapitel 10 erwähnt, ist die gezeigte Realisierung nicht Thread-sicher. Rufen mehrere Threads nahezu zeitgleich diese `getInstance()`-Methode auf, so kann es zu verschiedenen Race Conditions kommen. Im einfachsten Fall führen mehrere Threads die Prüfung auf `null` durch und werden anschließend unterbrochen. Alle Threads erzeugen als Folge »ihr« eigenes Singleton.

Eine gängige Lösung ist das Synchronisieren der `getInstance()`-Methode. Durch das Schlüsselwort `synchronized` wird der Zugriff durch mehrere Threads unterbunden und zusätzlich eine konsistente Sicht aller Threads auf den aktuellen Zustand der Variablen garantiert:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

Diese Realisierung gemäß Lazy Initialization (vgl. Abschnitt 22.3) sorgt bei jedem Zugriff auf die synchronisierte Methode für einen gegenseitigen Ausschluss und die Ausführung als kritischer Bereich. Dieses Verhalten wird eigentlich nur für die erste Initialisierung des statischen Attributs `INSTANCE` benötigt. Für alle weiteren Zugriffe ist dieser Aufwand weder erforderlich noch gewünscht, weil dadurch Nebenläufigkeit verhindert wird: Rufen andere Komponenten häufig die `getInstance()`-Methode auf, so kann das Locking einen Flaschenhals darstellen, weil Zugriffe nicht parallel erfolgen können, sondern sequenziell nacheinander ausgeführt werden.

Als Singleton realisierte Klassen stellen in der Regel zentrale und vielfach benötigte Komponenten dar. Deswegen ist die zuvor gezeigte Lazy Initialization zur Konstruktion eines Singletons eher fragwürdig, weil sie vor allem für optionale und aufwendig zu konstruierende Komponenten gedacht ist, was auf Singletons eher seltener zutrifft. Außerdem verursacht gerade Lazy Initialization unter Umständen verschiedenste Probleme beim Multithreading. Und tatsächlich lässt sich der Sourcecode viel einfacher, übersichtlicher und kürzer formulieren, wenn man auf Lazy Initialization verzichtet. Man nutzt stattdessen eine direkte, Thread-sichere, statische Initialisierung des statischen Attributs `INSTANCE`:

```
public final class Singleton
{
    private static final Singleton INSTANCE = new Singleton();

    private int attribute1;
    private int attribute2;

    public static Singleton getInstance()
    {
        return INSTANCE;
    }

    private Singleton()
    {
    }

    public void businessMethod1() { /* ... */ }
    public void businessMethod2() { /* ... */ }
}
```

Thread-Sicherheit ist bei dieser Art der Realisierung dadurch gegeben, dass die Klassenbeschreibung einmal geladen wird und anschließend eine Initialisierung erfolgt, bevor die Klasse für andere Klassen zugreifbar wird.

Folgende Vorteile ergeben sich zusätzlich: Im Unterschied zum ersten Ansatz wird das Objekt immer erzeugt und steht bei der ersten Benutzung der Klasse direkt zur Verfügung. Dadurch entfällt der Bedarf für die Synchronisation. Die Deklaration als `final` sorgt zudem dafür, dass sich die Referenz `INSTANCE` nicht mehr ändern kann und somit auch keine Prüfungen auf `null` notwendig sind.

Besondere Variante basierend auf `enum` Die zuvor gezeigte Lösung war bis JDK 1.4 die kürzeste und eleganteste Realisierung eines SINGLETONs. Seit JDK 5 kann man eine Besonderheit von `enum`-Aufzählungen nutzen. Diese sind immer eindeutig und nur einmal in einer JVM vorhanden. Somit lässt sich ein SINGLETON als eine einelementige Aufzählung wie folgt definieren:

```
public enum Singleton
{
   INSTANCE;

   private int attribute1;
   private int attribute2;

   public void businessMethod1() { /* ... */ }
   public void businessMethod2() { /* ... */ }
}
```

So elegant diese Lösung zunächst vielleicht auch aussehen mag, sie besitzt doch ein paar Nachteile. Einer davon ist, dass `enum`-Aufzählungen implizit von der Basisklasse `Enum<E extends Enum<E>>` abgeleitet sind. ***Für Singletons sind Basisklassen aber in der Regel nicht wünschenswert***. Man erbt im obigen Beispiel nämlich Funktionalität in Form der Methoden `name()`, `ordinal()` sowie `compareTo(Singleton)`. Im besten Fall ist das überraschend, meistens aber eher verwirrend und lenkt von den eigentlichen Business-Methoden ab. Es kommt noch schlimmer: Für ein Singleton erwartet man nur eine statische Methode, nämlich `getInstance()`. Obige Realisierung basierend auf `enum` bietet allerdings diverse statische Methoden, beispielsweise `values()`, `valueOf(String)` usw.

Bewertung

Der Einsatz eines SINGLETONs hat folgende Auswirkungen:

+ **Zentraler Zugriffspunkt** – Es gibt einen zentralen Zugriffspunkt auf benötigte Funktionalität, der eine sichere Initialisierung erlaubt.
+ **Strukturierung** – Die Zugriffe auf das SINGLETON werden strukturiert und Implementierungsdetails lassen sich verstecken.
o **Realisierungsprobleme** – Der Einsatz ist komplizierter, als man meint: Intuitive Lösungen unter Verwendung von Lazy Initialization führen zu Problemen in

Multithreading-Umgebungen. In der Vergangenheit wurden dazu komplexe, zum Teil fehlerhafte und schlecht lesbare Realisierungen entwickelt, unter anderem das sogenannte ***Double Checked Locking***.[4] Einfacher und zudem korrekt ist die zuvor vorgestellte Lösung. Ein subtileres Problem lässt sich dadurch jedoch nicht lösen: die im folgenden Punkt beschriebene fehlende Eindeutigkeitsgarantie!

- **Keine Eindeutigkeitsgarantie** – Der Einsatz ist problematisch, wenn man mehrere `ClassLoader`[5] verwendet. Es gibt dann nämlich keine Eindeutigkeitsgarantie mehr und es sind somit mehrere Instanzen eines Singletons möglich – pro `ClassLoader` eine. Daher müssen zusätzliche Betrachtungen angestellt werden. Dies wird im folgenden Hinweis ausführlicher beschrieben.

Hinweis: Fallstricke und Probleme durch mehrere `ClassLoader`

Das SINGLETON-Muster soll dafür sorgen, dass nur eine einzige Instanz einer Klasse innerhalb einer Applikation erzeugt werden kann. Allerdings garantiert der Einsatz dieses Musters selbst innerhalb einer JVM nicht die Einzigartigkeit einer Instanz. ***Das SINGLETON-Muster sorgt in Java nur dafür, dass höchstens eine Instanz pro `ClassLoader` existiert. In einer JVM kann es also durchaus eben so viele Instanzen des »Singletons« geben, wie es `ClassLoader` gibt***.

Das kann z. B. bei einem Webserver der Fall sein, da dort zum Teil mehrere `ClassLoader`-Instanzen eingesetzt werden. Diesen Sachverhalt sollte man kennen, wenn ein SINGLETON eine Ressource kapseln soll, die nur einmal vorhanden ist. In komplexen Applikationen verschärfen sich die Probleme, wenn hier mit mehreren JVMs gearbeitet wird. Dasselbe gilt beim Einsatz von Applikationsservern. Wie man an dieser kurzen Diskussion erkennt, sind beim Einsatz des SINGLETON-Musters in komplexen Applikationen einige Vorüberlegungen notwendig. Verwendet man allerdings nur einen `ClassLoader`, dann sind die hier angesprochenen Punkte nicht relevant, und man muss lediglich die im Text gegebenen Empfehlungen zur Realisierung beachten.

Achtung: SINGLETON als Anti-Pattern

Ich kenne Applikationen, die das SINGLETON-Muster als Allheilmittel ansehen: Dort sind viele Klassen als ein solches realisiert. In diesem Fall kann der Einsatz des Musters aber mit einer Ansammlung von statischen Variablen verglichen werden. Es ist jedoch nicht sinnvoll, überall Zugriff auf alle Klassen anzubieten. Im Normalfall sollte es nur einige wenige Klassen geben, die man zentral bereitstellen muss. Hält man sich nicht daran, so wird das SINGLETON schnell zu einem Anti-Pattern.

[4] Hintergründe finden Sie online unter `http://www.ibm.com/developerworks/java/library/j-dcl.html`.

[5] Ein sogenannter Klassenlader vom Typ `ClassLoader` lädt benötigte Klassenbeschreibungen bei Bedarf in den Speicher und entscheidet vorher, ob eine Klasse überhaupt geladen werden darf. Zudem stellt er sicher, dass Anwendungen keine Systemklassen überschreiben.

18.1.5 Prototyp (Prototype)

Beschreibung und Motivation

Das Vorgehen beim PROTOTYP-Muster besteht darin, basierend auf speziellen Vorlagenobjekten neue Objekte zu generieren. Man nutzt dieses Muster, wenn Instanzen einer Klasse einen großen gemeinsamen Grundstock an gleichen Werten ihrer Attribute haben und durch wenige Modifikationen gebrauchsfertig gemacht werden können. Außerdem kann der Einsatz zur Performance-Steigerung und zur ressourcenschonenden Objekterzeugung verwendet werden, wenn diese per Konstruktor zeitaufwendig ist (z. B. durch Datenbankzugriffe zum Ermitteln der Werte der Attribute) und somit ein Kopieren der Ergebnisse und anschließendes Parametrieren günstiger ist.

Dieses Muster besitzt zwei Varianten: Bei der ***statischen*** Variante dienen eine oder mehrere Kopiervorlagen (Prototypen) als Basis für neue Objekte. Bei der ***dynamischen*** Variante kann sogar der Typ der Kopiervorlage zur Laufzeit unterschiedlich sein.

Struktur

Die Struktur für die statische Variante des Musters wird in Abbildung 18-6 gezeigt. Es existiert ein Objekt vom Typ `Prototype`, der eine `makeCopy()`-Methode anbietet, um sich selbst zu kopieren. Damit kann jedes Objekt dieses Typs als Kopiervorlage agieren. Hier wird bewusst nicht der Name `clone()` verwendet, um eine klare Abgrenzung zum `Cloneable`-Interface aus dem JDK zu gewährleisten.

Abbildung 18-6 *Prototyp*

Nach einem Aufruf von `makeCopy()` können bei Bedarf weitere Anpassungen der Attributwerte durch `set()`-Methoden erfolgen, um das neu erstellte Objekt wie gewünscht zu parametrieren.

Beispiel

Die dynamische Variante stelle ich im Folgenden am vereinfachten Beispiel eines grafischen Editors vor, der über Buttons verschiedene Aktionen, wie Figuren zeichnen, kopieren und einfügen, erlaubt. Mehrfache Kopien sind in Abbildung 18-7 für einige Rechtecke gezeigt, wobei die aktuell selektierte Figur fett dargestellt ist.

Abbildung 18-7 *Beispielapplikation unter Verwendung des* PROTOTYP*-Musters*

Wird der Copy- bzw. Paste-Button gewählt, so werden spezielle Methoden aufgerufen und das momentan auf der Zeichenfläche selektierte Element kopiert bzw. eingefügt. Die beteiligten Klassen und deren Zusammenwirken zeigt das Klassendiagramm in Abbildung 18-8.

Abbildung 18-8 *Prototyp am Beispiel*

Zur Realisierung der Copy- bzw. Paste-Aktionen werden die Methoden `handleCopyElement()` und `handlePasteElement()` aufgerufen, um das momentan auf der Zeichenfläche selektierte Element zu kopieren. Dieses kann je nach Auswahl von unterschiedlichen Laufzeittypen sein. Damit eine einheitliche Verarbeitung möglich ist, müssen alle zu verarbeitenden Objekte einen gemeinsamen Basistyp besitzen. In diesem Beispiel ist dies die abstrakte Klasse `AbstractGraphicsElement` mit den konkre-

ten Subklassen `LineFigure`, `RectFigure` und `OvalFigure`. Eine Copy- bzw. Paste-Aktion nutzt lediglich die abstrakte Basisklasse und muss dadurch die Spezialisierungen nicht kennen. Auf diese Weise können sogar beliebige, abgeleitete Elemente kopiert werden, die zum Kompilierzeitpunkt noch unbekannt sind.

Schauen wir uns nun die Realisierung der Methode `handleCopyElement()` an. Es wird hier die zuvor beschriebene Funktionalität umgesetzt, indem eine Kopie des momentan selektierten grafischen Elements erzeugt und im Attribut `clipboardElement` gespeichert wird:

```
public void handleCopyElement()
{
   final AbstractGraphicsElement selected = contentModel.getSelectedElement();

   if (selected != null)
   {
       insertPos = selected.getPosition();
       clipboardElement = selected.makeCopy();
   }
   else
   {
       throw new IllegalStateException("Copy must only be activated if an " +
                                       "element is selected!");
   }
}
```

Ein Detail dieser Methode ist noch erwähnenswert: Für den Fall, dass im Modell kein Element selektiert ist (`getSelectedElement()` demnach `null` liefert), wird eine `IllegalStateException` ausgelöst. Zunächst könnte eine solche Reaktion auf diese zwar besondere, aber nicht ungewöhnliche Situation verwundern. Selbstverständlich darf zu gewissen Zeitpunkten durchaus kein Element selektiert sein. Als Folge sollte eine Statusverwaltung die Copy-Aktion jedoch deaktivieren. und damit auch der Button nicht anwählbar sein, um einen ansonsten sinnlosen Aufruf dieser Funktionalität zu verhindern. Folgt man dieser Regel, so kann der genannte Fall (`selected == null`) nur dann eintreten, wenn die Statusverwaltung der Aktionen fehlerhaft arbeitet. Der hier gezeigte, offensive Umgang mit unerwarteten Situationen ist ratsam und hilfreich, weil dadurch Denk- oder Realisierungsfehler noch während der Entwicklungsphase gefunden werden können.

Wird eine Paste-Operation ausgeführt, so sollen die meisten Objekteigenschaften beibehalten werden, lediglich die Objektposition wird angepasst und das zwischengespeicherte Objekt als Kopie eingefügt:

```
public void handlePasteElement()
{
    if (clipboardElement != null)
    {
        // Kaskadierende Position sicherstellen
        increaseImageInsertPos();

        // Kopie anpassen ...
        final Point insertPos = getImageInsertPos();
        clipboardElement.setPosition(insertPos.x, insertPos.y);
```

```
        // ... und dem Modell hinzufügen
        contentModel.addElement(clipboardElement);

        // hier nochmal kopieren, damit wir mehrfach einfügen können
        clipboardElement = clipboardElement.makeCopy();
    }
    else
    {
        throw new IllegalStateException("Paste must only be activated if an " +
                                        "element is selected!");
    }
}
```

Nachdem wir nun einige wichtige Bestandteile der Applikation kennengelernt haben, zeige ich abschließend die `main()`-Methode und das Programm PROTOTYPEEXAMPLE zum Start der Applikation:

```
public static void main(final String[] args)
{
   final ContentModel  contentModel  = new ContentModel();
   final ActionHandler actionHandler = new ActionHandler(contentModel);

   final AppFrame appFrame =
                 new AppFrame("PrototypeExample", contentModel, actionHandler);
   appFrame.setVisible(true);
}
```

Listing 18.2 *Ausführbar als* '**PROTOTYPEEXAMPLE**'

Bewertung

Der Einsatz des PROTOTYP-Musters besitzt folgende Implikationen:

+ **Vereinfachung** – Die Objektvervielfältigung und -konstruktion wird vereinfacht. Statt neue Objekte mit umfangreicher Parametrierung neu erzeugen zu müssen, wird hier auf eine Vorlage zurückgegriffen, die als Basis dient und lediglich einige Modifikationen zum Einsatz benötigt.
+ **Flexibilität** – Bei der dynamischen Variante ist das Kopieren von zur Kompilierzeit unbekannten Typen möglich. Dies erlaubt das Hinzufügen und Entfernen von kopierbaren Produkten zur Laufzeit.
+ **Nachvollziehbarkeit** – Der Ablauf ist im Gegensatz zum in die JVM integrierten `clone`-Mechanismus besser nachvollziehbar. In diesem Fall wird explizit eine Implementierung der `makeCopy()`-Methode gefordert. Beim Erfüllen des `Cloneable`-Interface gibt es nur implizite Annahmen über die Existenz von `clone()`-Methoden: `Cloneable` ist lediglich ein Marker-Interface (vgl. Abschnitt 3.4.3) und besitzt keine Methoden.
o **Fallstrick Referenzsemantik** – Bei Containerklassen ist schwierig zu entscheiden, ob durch die Referenzsemantik Probleme entstehen können. Details dazu finden Sie in der Diskussion zu Referenzierungen und Parameterübergaben per Call-by-Value, die in Abschnitt 3.1.1 geführt wurde.

18.2 Strukturmuster

Strukturmuster helfen dabei, Funktionalitäten leichter konfigurierbar oder handhabbar zu machen sowie Verhalten flexibel erweitern und anpassen zu können. Eine FASSADE kapselt und versteckt die Komplexität eines Subsystems. Mit einem ADAPTER lassen sich inkompatible Softwarestücke verbinden. Ein DEKORIERER erlaubt es, Klassen neue Funktionalität hinzuzufügen. Mit dem KOMPOSITUM lassen sich Einzelelemente und Gruppen aus Baumstrukturen einheitlich behandeln.

18.2.1 Fassade (Façade)

Beschreibung und Motivation

Das Entwurfsmuster FASSADE kann dabei helfen, die Komplexität eines Subsystems zu verbergen und den Zugriff darauf für externe Klassen zu vereinfachen bzw. zu steuern. Eine Fassadenklasse definiert dazu ein »High-Level«-Interface, um komplizierte, sehr feingranulare Interaktionen und Beziehungen zwischen Package-internen und -externen Klassen zu vermeiden. Dazu delegiert ein Fassadenobjekt die Aufrufe durch Klienten entsprechend der Zuständigkeit an spezielle Klassen des Subsystems. Es herrscht eine gerichtete Abhängigkeit: Die Fassadenklasse kennt die Klassen des Subsystems, aber nicht umgekehrt, d. h., keine Klasse des Subsystems kennt die Fassadenklasse.

Struktur

Betrachten wir zunächst ein System mit zwei Klienten `Client1` und `Client2`, die Datenbankzugriffe durchführen wollen. In Abbildung 18-9 erkennt man viele Abhängigkeiten von den eingesetzten Klassen. Dies liegt daran, dass die Logik und Steuerung jeweils in den Klienten erfolgt.

Abbildung 18-9 *Abhängigkeiten ohne Fassade*

Mit der Anzahl der beteiligten Klienten steigen diese Abhängigkeiten immer weiter. Dies ist der Punkt, an dem man über den Einsatz des FASSADE-Musters nachdenken sollte. Führt man im Beispiel eine Fassadenklasse `DBAccess` ein, so lassen sich dadurch die Zugriffe auf die Datenbankzugriffsklassen strukturieren. Abbildung 18-10 zeigt dies eindrucksvoll.

Abbildung 18-10 *Strukturierung durch Fassade*

Die Fassadenklasse verbirgt den Verbindungsaufbau und die Komplexität der Datenbankzugriffe vor aufrufenden Klienten. Diese nutzen nur noch eine Verwaltungsklasse und damit eine einzige Schnittstelle, die es ihnen ermöglicht, ihre Aufgaben auszuführen. Sämtliche Details der Infrastruktur sind für Aufrufer nicht mehr von Interesse. Die meisten Klassen des Subsystems können und sollten folglich Package-private definiert werden, um Implementierungsdetails nach außen auch tatsächlich zu verbergen.

Bewertung

Der Einsatz des FASSADE-Musters hat folgende Auswirkungen:

+ **Vereinfachung des APIs** – Der Zugriff auf die Funktionalität eines Subsystems fällt leichter, weil eine einfachere und oftmals verständlichere Schnittstelle zum Zugriff auf dessen Klassen angeboten wird.
+ **Trennung zwischen Klienten und Subsystem** – Die Zugriffe werden strukturiert und Implementierungsdetails lassen sich verstecken. Klienten müssen somit weniger Kenntnis über die Komponenten des Subsystems haben. Dadurch kommt es zu einer besseren Trennung und loseren Kopplung zwischen Klienten und dem Subsystem.
+ **Zentralisierung von Funktionalität und Updates** – Die Steuerung von sogenannten ***Cross-Cutting Concerns*** (vgl. folgenden Praxistipp »Cross-Cutting Concern und aspektorientierte Programmierung«) erfolgt einheitlich an einer Stelle. Außerdem kann man eine Fassade zu einer Art Transaktionsverwaltung bei der Kommunikation von einem GUI mit einem komplexeren Modell nutzen. Im besten Fall

greift das GUI dann nur über ein Fassadenobjekt auf ein Modell zu. Dies erlaubt es, die Propagation von Modelländerungen nicht feingranular, sondern gebündelt und gegebenenfalls gepuffert durchzuführen. Dadurch vermeidet man beispielsweise ständiges Neuzeichnen und »Geflacker«. Erst nachdem Änderungen vollständig im Modell verarbeitet wurden, wird das GUI durch die Fassadenklasse über Änderungen informiert und dann aktualisiert.

- **Gefahr eines breiten Interface** – Die Kapselung der Funktionalität des Subsystems führt schnell zu einem sehr breiten Interface mit vielen Methoden.
- **Keine Nutzungsgarantie** – Es ist nicht gewährleistet, dass Klienten die Fassadenklasse auch tatsächlich benutzen. Stattdessen können Klienten daran vorbei programmieren, wodurch die zuvor genannten positiven Effekte verloren gehen. Bei der GoF wird dieser Punkt als Feature angesehen, basierend auf folgender Begründung: Für den Regelfall liefert die Fassadenklasse eine gute Standardimplementierung, für Sonderfälle bleibt der Weg zur Spezialimplementierung offen. Dies ist stichhaltig. Allerdings entstehen in der Praxis aus Bequemlichkeit häufig immer wieder neue »Schleichwege« an der Fassadenklasse vorbei. Dadurch leidet die Wartbarkeit, da der Sourcecode unübersichtlich und schlechter nachvollziehbar wird. Um derartige »Fehlverwendungen« aus anderen Packages zu unterbinden, kann man Klassen des Subsystems Package-private definieren.
- **Mehr Indirektionen** – Es werden Weiterleitungen von Methodenaufrufen, sogenannte ***Indirektionen***, eingeführt.

Tipp: Cross-Cutting Concern und aspektorientierte Programmierung

Unter einem sogenannten Cross-Cutting Concern versteht man eine orthogonale, vom eigentlichen Programmcode unabhängige Funktionalität, die in mehreren Programmteilen immer wieder benötigt wird (etwa Logging oder Transaktionsverwaltung) und die durch normale Modularisierungstechniken nicht adäquat abgebildet werden kann. Daher sind diese Funktionalitäten immer wieder an benötigten Stellen aufzurufen oder auszuprogrammieren. Die sogenannte ***aspektorientierte Programmierung*** (***AOP***) adressiert diese Probleme und ermöglicht es, dass derartige Funktionalität nicht direkt in der Anwendung selbst realisiert werden muss. Allerdings unterstützt Java die aspektorientierte Programmierung nicht von Hause aus. Mit AspectJ (`http://www.eclipse.org/aspectj/`) ist eine Erweiterung verfügbar, die den vom Java-Compiler erzeugten Bytecode modifiziert und an speziellen Stellen die durch Aspekte definierte Zusatzfunktionalität einbindet.

18.2.2 Adapter

Beschreibung und Motivation

Dieses Muster lässt sich sehr anschaulich am Beispiel von zwei Steckern oder Puzzleteilen motivieren, die nicht ineinander passen. Man nimmt ein Adapterstück und nutzt dieses zur Verbindung, um die nicht passenden Teile zu verbinden. Bei Software handelt man gleichermaßen: Man verwendet ein Softwarestück, das die Schnittstellen ansonsten inkompatibler Software aufeinander abbildet, um diese miteinander zu verbinden. Der entscheidende Vorteil gegenüber einem direkten Ansprechen einer anderen Klasse ist, dass durch den Einsatz eines Adapters keine Änderungen an den Schnittstellen der bereits vorhandenen Implementierungen nötig sind. Dafür muss allerdings ein neues Softwarestück, der Adapter, entwickelt werden.

Struktur

Ein Klient `Client` nutzt ein Objekt vom Typ `Adapter`, um die Funktionalität einer vorhandenen Klasse `OriginalClass` mit einer für den Klienten passenden Schnittstelle verwenden zu können. Diese wird durch den `Adapter` zur Verfügung gestellt (vgl. Abbildung 18-11). Die Referenz auf die Originalklasse wird häufig `adaptee` genannt.

Abbildung 18-11 *Adapter in UML*

Beispiel

In diesem Beispiel ist eine Implementierung eines Adapters gezeigt, die eine beliebige Realisierung des Interface `List<E>` (etwa `ArrayList<E>`, `LinkedList<E>` etc.) mit einem Listenmodell für Swing-Komponenten kompatibel macht.

```
public static class ListToListModelAdapter<E> extends AbstractListModel<E>
{
    private final List<E> adaptee = new ArrayList<>();

    public ListToListModelAdapter(final List<E> data)
    {
        this.adaptee.addAll(data);
    }

    public int getSize()
    {
        return adaptee.size();
    }
```

```
    public E getElementAt(final int i)
    {
        return adaptee.get(i);
    }
}
```

Listing 18.3 *Ausführbar als* **'SIMPLEADAPTEREXAMPLE'**

Die `OriginalClass` aus dem UML-Strukturdiagramm ist die das `List<E>`-Interface erfüllende Referenz `data`. Die gewünschte Klasse bzw. das gewünschte Interface wird durch die abstrakte Klasse `AbstractListModel<E>` und das Interface `ListModel<E>` repräsentiert. Die verbindende Adapterklasse ist die Klasse `ListToListModelAdapter<E>`. Diese nutzt die abstrakte Klasse `AbstractListModel<E>`, um viele Standardaufgaben aus dem Interface `ListModel<E>` zu erfüllen: Dadurch müssen nur an zwei Stellen die Funktionalitäten von `List<E>` auf `ListModel<E>` abgebildet werden. Dazu nutzen die Methoden `getSize()` und `getElementAt(int)` die beiden Methoden `size()` bzw. `get(int)` des Interface `List<E>`. Das benötigt einen Zugriff auf die `OriginalClass`, hier in Form der `List<E>`. In der Regel wird dies durch Speicherung als Attribut in der Adapterklasse gelöst. In diesem Fall ist dies die Referenz `adaptee`, die eine flache Kopie der Eingabeliste erstellt.

Bewertung

Der Einsatz des ADAPTER-Musters hat folgende Auswirkungen:

+ **Sicherstellung von Kompatibilität** – Zwei unabhängige Klassen (oder Systeme) werden durch den Adapter kompatibel zueinander gemacht, wodurch es möglich wird, die Funktionalität einer anderen Klasse zu nutzen, ohne deren Implementierung ändern zu müssen. Zwar könnte man dies auch durch Änderungen in den bestehenden Klassen erreichen, dadurch könnten aber Inkompatibilitäten entstehen, und die Wahrscheinlichkeit für Fehler steigt.
+ **Keine Änderungen am bestehenden System** – Es ist nur die Realisierung des Adapters erforderlich. Dadurch wird das Risiko für Fehler minimiert, da die bisherige (getestete) Funktionalität unverändert bleibt.
o **Aufwand durch Delegation** – Es wird Delegation genutzt und es ist zum Teil einiges an Codierungsaufwand notwendig.

18.2.3 Dekorierer (Decorator)

Beschreibung und Motivation

Mithilfe des DEKORIERER-Musters kann man vorhandenes Objektverhalten um zusätzliches Verhalten erweitern, ohne dass dafür der Sourcecode der ursprünglichen Klasse modifiziert werden muss. Dies ist dann praktisch, wenn entweder eine zu erweiternde Klasse nicht als Sourcecode vorliegt oder dieser nicht verändert werden darf. Ein erster Gedanke ist häufig, eine Subklasse zu bilden und dort die gewünschten Erweiterungen vorzunehmen. Wie bereits in Abschnitt 3.3.2 diskutiert, löst diese Art der Realisierung teilweise eine kombinatorische Explosion von Klassen aus, wenn verschiedenste Funktionalitäten miteinander verknüpft werden sollen. Das DEKORIERER-Muster bietet hierzu eine Alternative: Erweiterungen in der Funktionalität werden wie ein Mantel um die vorhandene Funktionalität gelegt. Jede Dekoriererklasse realisiert nur einen Teil der Gesamtfunktionalität. Daher wird das Muster manchmal auch ***Wrapper*** genannt.

Struktur

Grundlage dieses Musters ist ein gemeinsamer Basistyp (Interface oder abstrakte Klasse), der die öffentliche Schnittstelle für Dekoriererklassen und für zu dekorierende Klassen beschreibt. In diesem Beispiel definiert das Interface `CommonIF` eine spezielle Funktionalität, die durch die zwei konkreten Realisierungen `ConcreteImpl1` bzw. `ConcreteImpl2` erfüllt wird. Soll nun weitere Funktionalität zur Verfügung gestellt werden, so kann dies mithilfe der beiden Dekoriererklassen `ConcreteDecorator1` bzw. `ConcreteDecorator2` geschehen. Zur Realisierung der Ummantelung hält jedes Dekoriererobjekt eine Referenz auf das zu dekorierende Objekt in Form des Interface `CommonIF`. Werden verschiedene Dekoriererklassen benötigt, um unterschiedliche Erweiterungen unabhängig einsetzen zu können, so kann man eine abstrakte Basisklasse `AbstractDecorator` einführen. Diese hält eine Referenz auf das Interface `CommonIF` und kann damit entweder konkrete Ausprägungen oder wiederum Dekoriererklassen referenzieren. Abbildung 18-12 zeigt das entsprechende Klassendiagramm.

Abbildung 18-12 *Klassendiagramm mit mehreren Dekoriererklassen*

Bei Aufruf von Methoden eines Dekoriererobjekts ruft dieses gleichlautende Methoden des referenzierten, zu dekorierenden Objekts auf und delegiert somit diesen Auftrag, um vorher oder nachfolgend gewünschte Erweiterungen durchzuführen. Problemlos können auch mehrere Dekoriererobjekte hintereinander geschaltet werden, d. h., es findet dann eine mehrfache Ummantelung statt.

Existiert lediglich eine Dekoriererklasse bzw. ist nur eine geplant, kann man auf die Basisklasse verzichten. Eine Mehrfachummantelung ist in einem solchen Fall lediglich mit gleicher Funktionalität möglich. Dies ist meistens nicht sinnvoll.

Beispiel

Ein Beispiel für dieses Muster ist die Umkehrung einer Sortierreihenfolge, was durch eine spezielle Realisierung des Interface `Comparator<T>` umgesetzt wird. Wir definieren eine Dekoriererklasse `ReverseComparator<T>`, die das Interface `Comparator<T>` implementiert und in deren `compare(T, T)`-Methode die Eingabeparameter tauscht:

```
public final class ReverseComparator<T> implements Comparator<T>
{
    private final Comparator<T> originalComparator;

    public ReverseComparator(final Comparator<T> originalComparator)
    {
        this.originalComparator = Objects.requireNonNull(originalComparator,
                                  "originalComparator must not be null!");
    }

    @Override
    public int compare(final T o1, final T o2)
    {
        return originalComparator.compare(o2, o1);
    }
}
```

In diesem Fall besteht die Bearbeitung durch das Dekoriererobjekt demnach lediglich im Tausch der Eingabeparameter und somit dem Aufruf von `compare(o2, o1)`. Das zugehörende Klassendiagramm ist in Abbildung 18-13 dargestellt.

Abbildung 18-13 *Klassendiagramm des ReverseComparator*

Ähnliches im JDK Im JDK findet sich in der Klasse `Collections` folgende Methode:

```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

Diese erzeugt eine Invertierung einer Sortierung auf die gleiche Weise, wie wir es getan haben. Für eine Instanz eines `Comparator<T>` kann man Gleiches durch Aufruf von dessen Defaultmethode `reversed()` erreichen.

Bewertung

Der Einsatz des DEKORIERER-Musters besitzt folgende Auswirkungen:

+ **Transparente Ergänzung zusätzlicher Funktionalität** – Zusätzliche Funktionalität kann transparent für einen Aufrufer hinzugefügt werden. Dies ist sogar zur Laufzeit möglich, wodurch eine Realisierung gemäß diesem Muster einer puren statischen Vererbung überlegen ist.
+ **Hintereinanderschaltung** – Mehrere unterschiedliche Dekoriererobjekte lassen sich hintereinander schalten, um komplexere Funktionalitäten zu realisieren.
+ **Flexibilität** – Die zu dekorierende Klasse ist nicht festgelegt, da lediglich gegen eine gemeinsame Schnittstelle programmiert wird. Dekoriererklassen können somit für verschiedene zu dekorierende Klassen genutzt werden und ermöglichen damit Wiederverwendung. Statische Vererbung erlaubt das nicht.
+ **Vereinfachung von Vererbungshierarchien** – Komplexe und unübersichtliche Vererbungshierarchien können durch Einsatz dieses Musters vermieden werden.
o **Gemeinsamer Basistyp benötigt** – Damit eine Ummantelung von Klassen möglich ist, müssen alle im DEKORIERER-Muster beteiligten Klassen einen gemeinsamen Basistyp besitzen, der die öffentliche Schnittstelle für Dekoriererklassen und für zu dekorierende Objekte definiert.
o **Fehlende Kontrolle** – Eine Kontrolle, wer welche Funktionalität wie und wann hinzufügt und ob dies sinnvoll ist, bleibt der Disziplin des Entwicklers überlassen. Eine mehrfache Hintereinanderschaltung von Objekten derselben Dekoriererklasse, beispielsweise mehrmals Instanzen von `BufferedInputStream` oder `ReverseComparator<T>`, ist somit möglich, aber meistens nicht sinnvoll.
- **Zugriff auf Spezialisierungen schwieriger möglich** – Die Funktionalität wird durch Dekoriererobjekte transparent hinzugefügt, wodurch ein Aufrufer nicht direkt darauf zugreifen kann, da dieser nur eine Referenz auf die allgemeine Dekoriererklasse hält. Als Lösung kann man eine Referenz auf eine konkrete Dekoriererklasse speichern, um deren Zusatzfunktionalität explizit nutzen zu können.
- **Implementierungsaufwand** – Enthält das zu ummantelnde Interface relativ viele Methoden, so müssen all diese implementiert werden.

18.2.4 Kompositum (Composite)

Beschreibung und Motivation

Das KOMPOSITUM-Muster ermöglicht es, in hierarchischen Datenstrukturen sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln. Damit kann man in den meisten Fällen den Unterschied zwischen Einzelobjekten und Kompositionen außer Acht lassen.

Struktur

Um sowohl Einzelobjekte als auch Kompositionen einheitlich behandeln zu können, definiert eine abstrakte Basisklasse (oder alternativ auch ein Interface) `Component` die gemeinsame Schnittstelle für alle Akteure. Diese Schnittstelle wird jeweils sowohl von Einzelobjekten mit dem Typ `Leaf` als auch vom Kompositum mit dem Typ `Composite` implementiert. Das Kompositum wird in der Regel noch weitere Methoden besitzen, um beispielsweise die hierarchische Struktur aufzubauen. Abbildung 18-14 stellt das Klassendiagramm für das KOMPOSITUM-Muster dar.

Abbildung 18-14 *Grundstruktur des* KOMPOSITUM-*Musters*

Besonderheiten Bei der GoF sind die speziellen Operationen, unter anderem diejenigen, die zum Aufbau der Hierarchie notwendig sind, Bestandteil der gemeinsamen Schnittstelle. Durch die Definition innerhalb der Basisklasse müssen diese Methoden in der Klasse der Einzelobjekte implementiert werden. Dies kann entweder in Form eines leeren Methodenrumpfs geschehen oder aber, indem dort eine `UnsupportedOperationException` ausgelöst wird. Beide Varianten können problematisch sein. ***Meiner Ansicht nach sollten daher Spezialoperationen nicht Bestandteil der gemeinsamen Schnittstelle sein***.

Für viele Anwendungsfälle, etwa die später gezeigten Berechnungen, wird keine Unterscheidung benötigt, sondern sie ist sogar unerwünscht. Die Aufnahme der Spezialmethoden des Kompositums in die Klasse der Einzelobjekte verletzt zwar dieses Transparenzkriterium nicht, ist aber tendenziell eher schlechtes Design, da sich diese Methoden in den Einzelobjekten nicht sinnvoll implementieren lassen.

In grafischen Oberflächen wird für einige Aktionen die Unterscheidung zwischen Einzelobjekt und Kompositum benötigt, etwa um in Kontextmenüs Aktionen auf Gruppen zu aktivieren oder Aktionen für Einzelelemente zu deaktivieren. Für diesen Anwendungsfall bietet es sich an, die gemeinsame Schnittstelle um eine Methode `isComposite()` zu erweitern. Die Implementierung für Einzelobjekte gibt immer den Wert `false` zurück, die für Komposita den Wert `true`. Nur im letzteren Fall kann man dann einen expliziten Cast auf die Kompositum-Klasse anwenden, um dafür spezifische Operationen auszuführen.

Beispiele

Im Folgenden stelle ich ein vereinfachtes Beispiel aus der Praxis vor, das mithilfe des KOMPOSITUM-Musters die Projektkosten hierarchisch organisierter Projekte berechnen soll. Ziel ist es, die Kosten unabhängig von der Hierarchieebene mit dem gleichen Aufruf ermitteln zu können. Eine Projekthierarchie besteht aus Einzelprojekten (Klasse `Project`) oder aus Projektgruppen (Klasse `ProjectGroup`), die mehrere Einzelprojekte oder untergeordnete Projektgruppen verwalten können. Diese Struktur wird mithilfe des KOMPOSITUM-Musters wie folgt modelliert (Abbildung 18-15):

- Ein Einzelprojekt entspricht einem Blatt (`Project`).
- Eine Projektgruppe entspricht dem Kompositum (`ProjectGroup`). Unterprojekte werden mit der Methode `add(ProjectComponent)` hinzugefügt.
- Die abstrakte Basisklasse `ProjectComponent` definiert die gemeinsame Schnittstelle. Dazu zählt inbesondere die abstrakte Methode `calcCosts()`.

Abbildung 18-15 *Kompositum am Beispiel einer Projektstruktur*

Einzelprojekten sind bestimmte Kosten zugeordnet (als Vereinfachung im Beispiel per Konstruktorparameter übergeben). Projektgruppen ermitteln ihre Kosten durch Addition der Kosten ihrer Unterprojekte (Einzelprojekte oder Unterprojektgruppen), die sie

transparent über die Schnittstelle der Basisklasse `ProjectComponent` verwalten. Mit dieser Modellierung können die Kosten auf jeder Hierarchieebene berechnet werden. Die Ermittlung dieser Kosten lässt sich somit einfach rekursiv wie folgt formulieren:

```
@Override
public int calcCosts()
{
    int costs = 0;
    for (final ProjectComponent current : subprojects)
    {
        costs += current.calcCosts();
    }

    return costs;
}
```

Im folgenden Sourcecode wird ein Hauptprojekt `mainProject` betrachtet, das zwei Projektgruppen `projectGroup` und `projectGroup2` als Unterprojekte enthält. Diese umfassen wiederum Einzelprojekte mit den Kosten 7, 4 und 5 Tage.

```
public static void main(final String[] args)
{
    final int SEVEN_DAYS = 7;
    final int FOUR_DAYS = 4;
    final int FIVE_DAYS = 5;
    final ProjectComponent project1 = new Project("Seven", SEVEN_DAYS);
    final ProjectComponent project2 = new Project("Four", FOUR_DAYS);
    final ProjectComponent project3 = new Project("Five", FIVE_DAYS);

    final ProjectGroup projectGroup = new ProjectGroup("Group 1: 7+4");
    projectGroup.add(project1);
    projectGroup.add(project2);
    final ProjectGroup projectGroup2 = new ProjectGroup("Group 2: 5");
    projectGroup2.add(project3);

    final ProjectGroup mainProject = new ProjectGroup("Main");
    mainProject.add(projectGroup);
    mainProject.add(projectGroup2);

    final ProjectComponent[] components = { project1, project2, project3,
                                projectGroup, projectGroup2, mainProject };
    for (final ProjectComponent current : components)
        printCosts(current);
}

private static void printCosts(final ProjectComponent projectComponent)
{
    System.out.println("Cost of '" + projectComponent.getName() + "': " +
                       projectComponent.calcCosts());
}
```

Listing 18.4 *Ausführbar als* '**COMPOSITEEXAMPLE**'

Die verschiedenen Aufrufe der Ausgaberoutine `printCosts(ProjectComponent)` zeigen sehr schön, wie durch den Einsatz des KOMPOSITUM-Musters eine Abstraktion von der Datenstruktur erfolgt und Einzelelemente und Gruppen auf allen Hierarchieebenen gleich behandelt werden können. Das Programm COMPOSITEEXAMPLE errechnet folgende Projektkosten:

```
Cost of 'Seven': 7
Cost of 'Four': 4
Cost of 'Five': 5
Cost of 'Group 1: 7+4': 11
Cost of 'Group 2: 5': 5
Cost of 'Main': 16
```

Ähnliches im JDK Die Klasse `DefaultMutableTreeNode` aus dem Package `javax.swing.tree` realisiert dieses Muster in kompakter Form: Diese Klasse kann sowohl ein Einzelobjekt als auch ein Kompositum selbst modellieren. Die Unterscheidung zwischen beiden geschieht durch die Methoden `hasChildren()` und `isLeaf()`.

Bewertung

Der Einsatz des KOMPOSITUM-Musters hat folgende Auswirkungen:

+ **Vereinfachung** – Der Zugriff auf die Funktionalität einer komplexeren Datenstruktur wird vereinfacht, weil ein Klient nicht wissen muss, welche Art von Element er anspricht.
+ **Strukturierung** – Zugriffe werden strukturiert und Implementierungsdetails lassen sich verstecken.
o **Erschwerter Zugriff auf Spezialisierungen** – Durch die Gleichbehandlung über eine gemeinsame Schnittstelle wird ein Zugriff auf Erweiterungen und Eigenschaften von Spezialisierungen von Blättern oder Kompositum verhindert. Beide können weitere Operationen anbieten, die sich jedoch nur mit Aufwand ansprechen lassen.

Achtung: Bäume, Graphen, Zyklen und das KOMPOSITUM-Muster

Sogenannte ***Bäume*** sind streng hierarchisch strukturiert und wie bei realen Bäumen wächst auch kein Ast wieder zurück in den Stamm. Ein Ende eines Asts wird als ***Blatt*** bezeichnet. Das ist ein Spezialfall eines Verästelungspunkts, den man ***Knoten*** nennt. Die verbindenden Äste heißen ***Kanten***.

Die durch ein Kompositum beschriebene Datenstruktur stellt jedoch nicht zwangsläufig einen Baum dar, sondern kann auch einen sogenannten ***Graphen*** modellieren. Graphen können sogenannte ***Zyklen*** enthalten, d. h., es kann eine Verbindung eines Unterelements auf ein beliebiges anderes Element existieren. Ist dies im Speziellen ein Oberelement, so kommt es damit zu einer Rückkopplung. Ein Baum ist also ein Spezialfall eines Graphen, nämlich einer ohne Zyklen.

Enthielte das obige Beispiel einen Zyklus, könnte die intuitive rekursive Berechnung der Methode `calcCosts()` in einer endlosen Ausführung münden.[a] Da zyklische Graphen einen seltenen Spezialfall für das KOMPOSITUM-Muster darstellen, wird hier keine Lösung gezeigt. Im Buch »Design Patterns Java Workbook« [52] von Steven J. Metsker findet man eine Behandlung dieses Themas.

[a] Tatsächlich kommt es durch die Begrenzungen der Aufrufhierarchie von Methoden zu einem `java.lang.StackOverflowError`.

18.3 Verhaltensmuster

Verhaltensmuster strukturieren oder vereinfachen komplexe Abläufe. Ein ITERATOR bietet eine einheitliche Möglichkeit, verschiedene Datenstrukturen zu durchlaufen und dabei die Details zu verstecken. Ein NULL-OBJEKT repräsentiert einen `null`-Wert als Objekt und hilft, Zustandsabfragen zu vermeiden. Eine SCHABLONENMETHODE erlaubt es, gewisse Teilschritte eines Algorithmus vorzugeben und an anderen Stellen Variationspunkte bereitzustellen. Die STRATEGIE hilft dabei, verschiedene Ausprägungen von Algorithmen zu definieren. Das COMMAND-Muster fasst Methodenaufrufe zu Befehlsaktionen zusammen. Ein PROXY agiert als Stellvertreter für ein anderes Objekt und kontrolliert den Zugriff darauf. Ein BEOBACHTER ermöglicht es, Zustandsänderungen anderer Objekte zu observieren und darauf zu reagieren. Vor allem in grafischen Applikationen nutzt man die sogenannte MODEL-VIEW-CONTROLLER-Architektur (MVC) zur Trennung von Zuständigkeiten und zur Darstellung verschiedener Sichten auf gleiche Daten.

18.3.1 Iterator

Beschreibung und Motivation

Das Entwurfsmuster ITERATOR bietet eine allgemeingültige Möglichkeit für Klienten, alle Elemente einer Datenstruktur zu besuchen, ohne dafür die internen Details für den Durchlauf kennen zu müssen. Es findet eine Abstraktion von der zu traversierenden Datenstruktur statt, wodurch keine Kenntnis über deren Aufbau nötig ist: Bäume lassen sich auf ähnliche Weise durchlaufen wie Listen oder Mengen. Dies ist ein gutes Beispiel für die Trennung von Zuständigkeiten. Die Datenstruktur verwaltet lediglich die Daten, jedoch nicht die Art, wie diese durchlaufen werden.

Struktur

Eine Iteratorklasse bietet zumindest die Möglichkeit, zu testen, ob ein nächstes Element existiert, und erlaubt es zudem, ein solches gegebenenfalls zu ermitteln. Die dazu notwendigen Methoden werden oftmals `hasNext()` und `next()` bzw. `hasMoreElements()` und `nextElement()`[6] oder ähnlich genannt.

Der in Abbildung 18-16 dargestellte `ContainerIterator` erlaubt das Durchlaufen eines `Container`-Objekts unter Verwendung dieser Methoden. Die dort gezeigte Lösung verwendet eine konkrete Realisierung eines Iterators für eine konkrete Containerklasse. Normalerweise existieren analog zur obigen Klasse `Container` weitere Datenstrukturen. Dann wäre es für deren Nutzer recht unpraktisch, dafür jeweils spezielle Iteratorklassen mit möglicherweise geringfügig unterschiedlichen Methodensignaturen kennen zu müssen. Zudem würde es die Austauschbarkeit der verwendeten Containerklassen erschweren, da der Sourcecode von Klienten auf deren spezielle Iteratortypen

[6] Dies ist die Namensgebung der Methoden im Interface `java.util.Iterator<E>` bzw. `java.util.Enumeration<E>`.

verweisen müsste. Für diese Fälle wäre es praktisch, wenn man einen Basisiteratortyp, einen sogenannten polymorphen Iterator, für alle nutzen könnte.

Abbildung 18-16 *Die Klasse* `Container` *mit zugehörigem* `ContainerIterator`

Polymorphe Iteratoren Nehmen wir dazu vereinfachend an, es gäbe ergänzend zur Klasse `Container` eine weitere selbst definierte Containerklasse namens `Set`. Beide Containerdatenstrukturen sollen selbstverständlich mithilfe von Iteratoren durchlaufen werden können. Im folgenden Beispiel implementieren die beiden Realisierungen `SetIterator` und `ContainerIterator` diese Funktionalität und verwalten die aktuelle Position während der Traversierung der Datenstrukturen. Gemäß der Technik des gemeinsamen Interface (vgl. Abschnitt 3.2.1) nutzen wir als polymorphen Iterator das Interface `CommonIterator`.

Dessen konkrete Implementierungen werden jeweils durch die Containerklassen erzeugt, indem sie ein Objekt der passenden Iteratorklasse zurückgeben. Klienten arbeiten jedoch nur gegen die allgemeine Schnittstelle `CommonIterator`. Abbildung 18-17 zeigt das entsprechende UML-Diagramm.

Abbildung 18-17 *Allgemeiner Iterator*

Beispiel

Für Java-Programmierer ist dieses Muster aus dem Collections-Framework bereits bekannt. Die definierten Container lassen sich über einen `java.util.Iterator<E>` traversieren. Dabei handelt es sich um einen polymorphen Iterator.

Bewertung

Der Einsatz des ITERATOR-Musters besitzt folgende Auswirkungen:

+ **Abstraktion und Kapselung** – Die zu durchlaufende Datenstruktur wird gekapselt und bei Bedarf kann diese im Nachhinein bei der Implementierung verändert werden, ohne dass dies Änderungen aufseiten von Klienten verursacht.
+ **Vereinfachung** – Die Realisierung eigener Iteratorklassen für komplexere Datenstrukturen kann dabei helfen, eine ansonsten schwierige Implementierung des Iterationsprozesses zu kapseln, aus der Programmlogik herauszulösen und diese zu vereinfachen.
+ **Bessere Lesbarkeit** – Die Lesbarkeit ist viel besser als bei einer `for`-Schleife mit indiziertem Zugriff. Es wird das Durchlaufen auf einer konzeptionellen Ebene statt auf einer syntaktischen Ebene abgebildet.
o **Existenz einer alternativen Schreibweise in der Sprache** – Seit Java 5 existiert mit der for-each-Schleife eine Möglichkeit, die Lesbarkeit von Iterationen zu verbessern. Es können beliebige Datenstrukturen durchlaufen werden, die das Interface `Iterable<T>` implementieren. Dies wird implizit mit einem `Iterator<E>` implementiert – versteckt aber dieses Detail.

18.3.2 Null-Objekt (Null Object)

Beschreibung und Motivation

Die Idee beim Einsatz des NULL-OBJEKT-Musters ist, einen Platzhalter für `null`-Referenzen zu verwenden und so ein nicht vorhandenes Objekt zu modellieren. Es wird ein spezielles Objekt bereitgestellt, das kein eigenes Verhalten im fachlichen Sinn definiert. Es kann überall dort eingesetzt werden, wo ansonsten `null` als semantische Aussage für »kein Objekt« dient. Damit vereinfacht sich die Behandlung für Klienten: Sie können auf `null`-Prüfungen und weitere Spezialbehandlungen im Applikationscode verzichten.

Struktur

Um als »funktionsloser« Ersatz dienen zu können, muss ein Null-Objekt die Schnittstelle der zu vertretenden Klasse erfüllen. Die Implementierung erfolgt entweder durch Vererbung oder durch Realisierung eines Interface. Abbildung 18-18 zeigt diese beiden Varianten.

Abbildung 18-18 *Implementierungsvarianten des* NULL-OBJEKT-*Musters*

Implementierungsvarianten Nutzt man Vererbung, so ist die Null-Objekt-Klasse `NullObjectAsSubClass` eine Spezialisierung der Klasse `OriginalClass`, soll aber keine Anwendungsfunktionalität zur Verfügung stellen. Demnach muss hier künstlich Funktionalität der Basisklasse wieder entfernt werden, indem Methoden überschrieben werden und eine (nahezu) leere Implementierung besitzen. Da die Null-Objekt-Klasse jedoch standardmäßig die ganze in der Basisklasse implementierte Funktionalität erbt, sind (versehentliche) Aufrufe an die Basisklasse möglich und nicht auszuschließen.

Implementiert die Klasse `OriginalClass` ein Interface, so sollte dieses bevorzugt zur Realisierung des Null-Objekts genutzt werden. Es ist sinnvoll, ein solches Interface direkt zu implementieren, d. h. ohne von der Basisklasse abzuleiten. Dadurch sind Implementierungsdetails der Basisklasse nicht mehr im Zugriff. Unwahrscheinliche, aber mögliche Fehlaufrufe sind somit ausgeschlossen. Im Beispiel ist dies durch die Klasse `NullObjectAsRealization` angedeutet.

Implementierung der Methoden Um die »Null-Eigenschaft« auszudrücken, müssen sinnvolle Implementierungen und Rückgabewerte für Methoden gefunden werden. Folgende Aufzählung nennt gute Kandidaten für Rückgabewerte:

- Alle `void`-Methoden werden leer implementiert.
- `boolean`-Methoden geben häufig `false` zurück (z. B. für `hasNext()`).
- Für `int` bieten sich oft `0` oder `-1` als Rückgabewerte an (z. B. für `getSize()`).
- Gleiches gilt für weitere Zahlentypen (`float`, `double` usw.).
- Für Strings bietet sich ein Leerstring als Rückgabe an.
- Für Objektreferenzen kann entweder rekursiv das NULL-OBJEKT-Muster angewendet oder aber der Wert `null` zurückgeliefert werden. Manchmal bietet sich auch das Auslösen einer `NoSuchElementException` an.
- Für untypisierte Container sind im Collections-Framework die NULL-OBJEKT-Konstanten `EMPTY_SET`, `EMPTY_LIST` oder `EMPTY_MAP` definiert. Für generische Typen existieren die Methoden `emptySet()`, `emptyList()` und `emptyMap()`, die typsichere, leere und unmodifizierbare Container zurückliefern.
- Arrays sollten mit `new ArrayType[0]` als leeres Array zurückgegeben werden.
- Für Methoden, deren Rückgabewert für ein Null-Objekt nicht sinnvoll gewählt werden kann, sollte eine `UnsupportedOperationException` ausgelöst werden.

Beispiel

Beim Durchlaufen einer komplexeren Baumstruktur, die über das KOMPOSITUM-Muster realisiert ist, kann die Anwendung des NULL-OBJEKT-Musters helfen, wenn man den Iterationsprozess für Blätter und Aggregationen gleichartig behandeln möchte. Jedes Element im Kompositum kann dafür einen Iterator zum Durchlaufen seiner Unterelemente anbieten: Für Container ist dies einfach durch Rückgabe der Iteratoren aus dem Collections-Framework zu realisieren. Blattelemente besitzen definitionsgemäß keine Unterelemente. Eine Rückgabe von `null` wäre denkbar, würde jedoch Sonderbehandlungen im Iterationsvorgang bedingen. Vermeiden kann man dies, wenn man eine Klasse `NullIterator<E>` definiert, die das Interface `Iterator<E>` erfüllt, aber keine Elemente zurückliefert und wie folgt realisiert ist:

```
public final class NullIterator<E> implements Iterator<E>
{
    public boolean hasNext()
    {
        return false;
    }

    public E next()
    {
        throw new NoSuchElementException("NullIterator provides no elements!");
    }

    public void remove()
    {
        throw new UnsupportedOperationException("NullIterator does not " +
                                                "implement remove()!");
    }
}
```

Durch Einsatz dieses speziellen Iterators vermeidet man, dass beim Iterieren jeweils Abfragen auf Blatt oder Container erfolgen müssen, weil Blattelemente normalerweise keinen Iterator für Unterelemente liefern. Nutzt man dagegen den `NullIterator<E>`, so werden keine Spezialbehandlungen beim Iterieren mehr benötigt.

Natürlich könnte man Kritik an der obigen Realisierung mit Exceptions üben, da Aufrufe einer der Methoden `next()` bzw. `remove()` eine Exception auslösen. Beide Methoden verhalten sich aber gemäß der Spezifikation im Interface `Iterator<E>`. Für den Aufruf von `next()` gilt im Speziellen, dass dieser niemals erfolgen sollte, ohne vorher mit `hasNext()` die Existenz eines weiteren Elements abgesichert zu haben.

Bewertung

Der Einsatz des NULL-OBJEKT-Musters bewirkt Folgendes:

+ **Bessere Lesbarkeit und keine Spezialbehandlungen** – Der Anwendungscode kann auf Spezialbehandlungen verzichten und ist dadurch klarer lesbar.
o **Konzeptionelle Probleme** – Für einige Anwendungsfälle ist es zur Repräsentation von `null`-Werten schwierig, eine sinnvolle Leerimplementierung zu finden.

Manchmal gibt es tatsächlich keine geeignete. Dies ist im obigen Beispiel bei der Realisierung der `next()`-Methode der Fall, weswegen auf das Auslösen einer Exception als Hilfsmittel zurückgegriffen wird. Auch eine sinnvolle Implementierung der Methode `remove()` lässt sich schwerlich finden – es gibt ja keine Daten, die man löschen könnte.

- **Fehlerverschleierung möglich** – In manchen Situationen ist es erforderlich, die Fälle »es gibt *ein* Objekt« und »es gibt *kein* Objekt« deutlich voneinander unterscheiden zu können. Der Einsatz dieses Musters kann in solchen Fällen zu Problemen führen und Fehler eher verschleiern. Bei der Besprechung der Technik Lazy Initialization werden wir dies in Abschnitt 22.3.1 genauer kennenlernen. Basierend auf dieser Argumentation muss abgewogen werden, ob eine Prüfung auf `null` oder der Einsatz eines Null-Objekts angebrachter ist. Seit Java 8 bietet sich für die Modellierung optionaler Werte die Klasse `Optional<T>` (vgl. Abschnitt 6.5) an.

18.3.3 Schablonenmethode (Template Method)

Beschreibung und Motivation

Das Muster SCHABLONENMETHODE definiert den grundsätzlichen Ablauf eines Algorithmus und erlaubt es, an einigen Stellen durch Subklassen spezielle Funktionalität einzubringen. Dazu wird ein Algorithmus zunächst in eine festgelegte Abfolge verschiedener Schritte aufgeteilt. Die Abfolge der Schritte wird in einer speziellen Methode der Basisklasse realisiert, die ***Schablonenmethode*** genannt wird. Sie ist in der Regel `final` definiert, um die grundsätzliche Abfolge vor Veränderungen zu schützen. Einige der Berechnungsschritte des Algorithmus sind in der Basisklasse noch undefiniert und werden durch abstrakte Methoden modelliert, die von Subklassen implementiert werden müssen. Möchte man Subklassen dagegen *optional* eine Veränderungsmöglichkeit anbieten, so kann man mit einer Leerimplementierung einer Methode in der Basisklasse arbeiten, die als ***Hook*** bezeichnet wird. Jede Subklasse kann durch Überschreiben dieser Methode bei Bedarf eigene Funktionalität ausführen.

Das beschriebene Vorgehen stellt sicher, dass die Struktur eines Algorithmus unverändert bleibt, Subklassen jedoch Möglichkeiten der Einflussnahme gegeben wird.

Struktur

Der grundsätzliche Ablauf und Algorithmus ist in der Basisklasse `BaseClass` in der Methode `templateMethod()` realisiert und umfasst die Teilschritte 1 bis 2 und einen abschließenden Hook. Diese Funktionalität wird durch die korrespondierenden Methoden `step1()`, `step2()` sowie `afterHook()` realisiert. Die Methode `templateMethod()` ist `final` und die Methoden `step1()` und `step2()` sind abstrakt. Die Methode `afterHook()` ist in der Basisklasse leer implementiert, kann aber in Subklassen mit Funktionalität gefüllt werden. Die Subklasse `SubClass` implementiert den variablen Teil des Algorithmus durch die Methoden `step1()` sowie `step2()` und kann so

beliebige Funktionalität im Ablauf des Algorithmus beisteuern. In Abbildung 18-19 ist das zugehörige Klassendiagramm gezeigt.

Abbildung 18-19 *Klassendiagramm des* SCHABLONENMETHODE-*Musters*

Beispiel

In der folgenden Klasse `BaseDrawingComponent` ist die Methode `paint(Graphics)` die Schablonenmethode. Der Vorgang des Zeichnens folgt dabei einem festgelegten Ablauf: Zunächst wird der Hintergrund mit `drawBackground(Graphics2D)` gezeichnet. Anschließend erfolgt ein Aufruf der abstrakten Methode `drawContent(Graphics2D)`, die den variablen Teil des Algorithmus definiert, der von Subklassen realisiert werden muss. Abschließend kann optional durch Implementieren der Methode `postDraw(Graphics2D)` eine beliebige Aktion nach dem Zeichnen ausgeführt werden:

```
public abstract class BaseDrawingComponent extends JComponent
{
    public final void paint(final Graphics g)
    {
        super.paint(g);

        final Graphics2D g2d = (Graphics2D) g;

        drawBackground(g2d);
        drawContent(g2d);

        postDraw(g2d);          // hook
    }

    private void drawBackground(final Graphics2D g2d)
    {
        final Dimension componentSize = getSize();
        g2d.setColor(Color.GRAY);
        g2d.fillRect(0, 0, componentSize.width, componentSize.height);
    }

    protected abstract void drawContent(final Graphics2D g2d);

    protected void postDraw(final Graphics2D g2d)
    {
    }
}
```

Von dieser Basisklasse `BaseDrawingComponent` sind die zwei Klassen `TileImageDrawingComponent` und `ImageDrawingComponent` abgeleitet, die den variablen Anteil des Algorithmus auf ganz unterschiedliche Weise implementieren, wie wir dies im Folgenden sehen werden.

Zufällige Bilder mit Kacheln: `TileImageDrawingComponent` Die Klasse `TileImageDrawingComponent` zeichnet eine Landschaft, die aus Kacheln besteht. Ein Beispiel ist in Abbildung 18-20 gezeigt.

Abbildung 18-20 *Landschaft erzeugt mit der Klasse* TILEIMAGEDRAWINGCOMPONENT

Bei jeder Größenänderung des Fensters wird per Zufall eine neue Landschaft generiert und durch die Methode `drawContent(Graphics2D)` wie folgt gezeichnet:

```
@Override
public void drawContent(final Graphics2D g2d)
{
    for (int x = 0; x < getSize().width; x += TILES_WIDTH)
    {
        for (int y = 0; y < getSize().height; y += TILES_HEIGHT)
        {
            final int tileIndex = (int) (Math.random() * tileImages.length);
            g2d.drawImage(tileImages[tileIndex], x, y, null);
        }
    }
}
```

Listing 18.5 *Ausführbar als* 'TILEIMAGEDRAWINGCOMPONENT'

Minizeichenprogramm: `ImageDrawingComponent` Ein simples Zeichenprogramm wird durch die Klasse `ImageDrawingComponent` realisiert und nutzt wieder die Basisklasse `BaseDrawingComponent`. Abbildung 18-21 zeigt die Miniapplikation.

Abbildung 18-21 *Minizeichenprogramm* IMAGEDRAWINGCOMPONENT

Die Methode `drawContent(Graphics2D)` wird so realisiert, dass dort eine Zeichenfläche mit `drawSheet(Graphics2D)` sowie ein Raster mit `drawGrid(Graphics2D)` gezeichnet wird. Die verschiedenen grafischen Figuren werden mit der Methode `drawFigures(Graphics2D)` gemalt. Als Letztes wird dort die (im Listing nicht gezeigte) Methode `drawRuler(Graphics2D)` zum Zeichnen zweier Lineale aufgerufen, wodurch gewährleistet ist, dass diese immer über allen anderen Elementen gezeichnet werden und damit in jedem Fall sichtbar sind.

```
public void drawContent(final Graphics2D g2d)
{
    drawSheet(g2d);
    drawGrid(g2d);

    drawFigures(g2d);

    drawRuler(g2d);
}

private void drawSheet(final Graphics2D g2d)
{
    g2d.setColor(Color.WHITE);
    g2d.fillRect(0, 0, getSize().width, getSize().height);
}

private void drawGrid(final Graphics2D g2d)
{
    g2d.setColor(Color.DARK_GRAY);

    for (int x = 0; x < getSize().width; x += GRID_SIZE_X)
    {
        for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)
        {
            g2d.drawLine(x, y, x, y);
        }
    }
}

private void drawFigures(final Graphics2D g2d)
{
    final Stroke oldStroke = g2d.getStroke();
    g2d.setStroke(new BasicStroke(2.0f));

    for (final AbstractGraphicsElement graphicsFigure :
                                       contentModel.getElements())
    {
        graphicsFigure.draw(g2d);
    }
    g2d.setStroke(new BasicStroke(5.0f));

    final AbstractGraphicsElement selectedElement =
                                  contentModel.getSelectedElement();
    if (selectedElement != null)
        selectedElement.draw(g2d);

    g2d.setStroke(oldStroke);
}
```

Listing 18.6 *Ausführbar als* **'IMAGEDRAWINGCOMPONENT'**

An diesem Beispiel wird der Hauptunterschied zwischen Schablonenmethode, Hook-Methode und einem Schritt im Algorithmus klar: Eine Hook-Methode ***kann*** überschrieben werden, um optionale Funktionalität zu realisieren. Eine Algorithmus-Methode ***muss*** zwingend überschrieben werden, um den Algorithmus mit Leben zu füllen. Und schließlich: Die Schablonenmethode ***darf nicht*** überschrieben werden.

Bewertung

Der Einsatz des SCHABLONENMETHODE-Musters hat folgende Auswirkungen:

+ **Definition von Erweiterungsstellen** – Es wird ein Algorithmus vorgegeben. Subklassen können an speziellen Stellen eigene Funktionalität einbringen.
o **Weniger Flexibilität** – Man ist darin eingeschränkt, den Algorithmus zu modifizieren. In diesem Fall ist diese Eigenschaft explizit gewünscht. Soll der Algorithmus jedoch variiert werden können, so ist das Muster SCHABLONENMETHODE nicht die geeignete Wahl. Alternativ kann man mit dem im Folgenden vorgestellten STRATEGIE-Muster zwar den Algorithmus variieren, dafür aber keine Teilschritte vorgeben.

18.3.4 Strategie (Strategy)

Beschreibung und Motivation

Das STRATEGIE-Muster ermöglicht es, das Verhalten eines Algorithmus an ausgesuchten Stellen anzupassen. Im Unterschied zum Muster SCHABLONENMETHODE werden die variablen Bestandteile eines Algorithmus durch eigene Klassen statt durch überschriebene Methoden realisiert. Im einfachsten Fall könnte man die Wahl der unterschiedlichen Funktionalitäten bzw. Strategien über `if`-Anweisungen regeln. Ein solches Vorgehen ist allerdings schlecht erweiterbar und wird schnell unübersichtlich. Beim STRATEGIE-Muster werden daher die variablen Teile eines Algorithmus jeweils in eigenen Klassen mit gemeinsamem Basistyp gekapselt und sind dadurch austauschbar. Das konkrete Verhalten kann bei Bedarf sogar erst zur Laufzeit durch Wahl einer beliebigen vorhandenen Realisierung festgelegt werden.

Struktur

Die abstrakte Klasse `AbstractStrategy` definiert eine Schnittstelle mit benötigten und zu variierenden Methoden. Statt einer abstrakten Klasse kann dies alternativ durch ein Interface erfolgen. Spezielle Funktionalität wird in den Subklassen `ConcreteStrategy1` und `ConcreteStrategy2` implementiert. Abbildung 18-22 stellt dies dar.

Abbildung 18-22 *Struktur des* STRATEGIE*-Musters in UML*

Beispiel

Die in den `sort()`-Methoden des Collections-Frameworks genutzten Realisierungen des Interface `Comparator<T>` sind Beispiele des STRATEGIE-Musters.

Im Folgenden stelle ich zur Demonstration des STRATEGIE-Musters einen Algorithmus zum Filtern vor: Aus einer Liste von Zahlen sollen einige Werte nach speziellen Kriterien herausgefiltert werden. Die hier gezeigte Variante beschränkt sich auf das Filtern von `int`-Werten. Weil Generics nicht für primitive Typen anwendbar sind, wird hier zur Speicherung eine Liste von `Integer`-Objekten verwendet.

Erste, intuitive Lösung ohne STRATEGIE-Muster Um eine Menge von Zahlen zu filtern, wird der erlaubte Wertebereich durch zwei `int`-Werte entweder in Form eines geschlossenen oder alternativ eines offenen Intervalls angegeben.[7] Die Art der Filterung lässt sich mithilfe einer Aufzählung `FilterType` mit den Werten `CLOSED_INTERVAL` und `OPEN_INTERVAL` bestimmen. Die eigentliche Filterung wird in einer Schleife realisiert: Dort werden die Werte geprüft und gegebenenfalls in die Ergebnismenge aufgenommen. Die folgende Implementierung setzt diese Anforderungen funktional korrekt um:

```
public static List<Integer> filterAll(final List<Integer> inputs,
                                      final FilterType filterStrategy,
                                      final int lowerBound,
                                      final int upperBound)
{
    final List<Integer> filteredList = new LinkedList<>();
```

[7] Geschlossen = inklusive, offen = exklusive der Randwerte.

```
    if (filterStrategy == FilterType.CLOSED_INTERVAL)
    {
        for (final Integer value : inputs)
        {
            if (value >= lowerBound && value <= upperBound)
                filteredList.add(value);
        }
    }
    if (filterStrategy == FilterType.OPEN_INTERVAL)
    {
        for (final Integer value : inputs)
        {
            if (value > lowerBound && value < upperBound)
                filteredList.add(value);
        }
    }

    return filteredList;
}

public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    System.out.println("Filtering values for interval 2-7");
    System.out.println("Using ClosedInterval [2,7] " + filterAll(inputs,
                                                FilterType.CLOSED_INTERVAL, 2, 7));

    System.out.println("Using OpenInterval ]2,7[ " + filterAll(inputs,
                                                FilterType.OPEN_INTERVAL, 2, 7));
}
```

Listing 18.7 *Ausführbar als* '**STRATEGYFILTERBASICEXAMPLE**'

Führen wir das Programm aus, so erhalten wir erwartungsgemäß folgende Ausgabe:

```
Filtering values for interval 2-7
Using ClosedInterval [2,7] [2, 3, 4, 5, 6, 7]
Using OpenInterval ]2,7[ [3, 4, 5, 6]
```

Funktional können wir also zufrieden sein. Was ist aber zum Design zu sagen? Eine kurze Analyse deckt folgende Probleme auf:

1. Der Sourcecode zum Filtern für ein geschlossenes bzw. offenes Intervall ist fast 1:1 dupliziert.
2. Jede zusätzlich benötigte Filterstrategie, etwa halboffene Intervalle, erfordert Erweiterungen in der `filterAll()`-Methode, wodurch die Lesbarkeit sinkt.
3. Werden weitere Auswahlkriterien gewünscht, so muss die ursprüngliche Klasse um `if`-Anweisungen zur Auswahl der entsprechenden Strategie und deren Realisierung erweitert werden – zudem werden neue Aufzählungswerte zur Unterscheidung benötigt.
4. Die realisierte Filterung setzt jeweils die Angabe von zwei Grenzwerten voraus. Wollte man eine vollständig andersartige Auswahlstrategie anwenden, so wäre dies mit dieser Implementierung extrem aufwendig, vielleicht sogar unmöglich: Eine Filterung auf eine Menge erlaubter Werte ließe sich so nicht realisieren.

Lösung mit dem STRATEGIE-Muster Es bietet sich an, das STRATEGIE-Muster einzusetzen. Die Basis bildet ein Interface `FilterStrategy` mit der dort definierten Methode `acceptValue(int)`. Zur Intervallprüfung werden zwei konkrete Strategieklassen `OpenInterval` und `ClosedInterval` definiert (vgl. Abbildung 18-23).

Abbildung 18-23 *Das Interface* `FilterStrategy` *und zwei Realisierungen*

Das grundsätzliche Vorgehen ist mit der zuvor vorgestellten Lösung identisch: Zum Filtern wird über die Eingabe iteriert und für jeden Wert überprüft, ob dieser in das Ergebnis aufgenommen werden soll. Allerdings wird diese Prüfung nicht mehr in der `filterAll()`-Methode realisiert, sondern an die dafür zuständigen Filterklassen delegiert. Exemplarisch zeige ich hier nur die Umsetzung des geschlossenen Intervalls, da die Realisierung des offenen Intervalls analog geschieht. Beide Klassen implementieren wir innerhalb einer Klasse `FilterStrategies` wie folgt:

```
public static class ClosedInterval implements FilterStrategy
{
   private final int lowerBound;
   private final int upperBound;

   public ClosedInterval(final int lowerBound, final int upperBound)
   {
      if (upperBound < lowerBound)
         throw new IllegalArgumentException("lowerBound must be <= upperBound");

      this.lowerBound = lowerBound;
      this.upperBound = upperBound;
   }

   @Override
   public boolean acceptValue(final int value)
   {
       return lowerBound <= value && value <= upperBound;
   }

   @Override
   public String toString()
   {
      return "ClosedInterval [" + lowerBound + ", " + upperBound + "]";
   }
}
```

Betrachten wir nun die konkrete Implementierung der Filterung:

```
public final class StrategyFilterExample
{
   public static List<Integer> filterAll(final List<Integer> inputs,
                                         final FilterStrategy filterStrategy)
   {
      final List<Integer> result = new LinkedList<>();
      for (final Integer value : inputs)
      {
         if (filterStrategy.acceptValue(value))
            result.add(value);
      }
      return result;
   }

   public static void main(final String[] args)
   {
      final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

      System.out.println("Filtering values for intervall 2-7");
      final FilterStrategy closedInterval = new ClosedInterval(2, 7);
      System.out.println("Using " + closedInterval + " " +
                         filterAll(inputs, closedInterval));

      final FilterStrategy openInterval = new OpenInterval(2, 7);
      System.out.println("Using " + openInterval + " " +
                         filterAll(inputs, openInterval));
   }
}
```

Listing 18.8 *Ausführbar als* **'STRATEGYFILTEREXAMPLE'**

Führt man das Programm aus, so stimmen die gelieferten Ergebnisse mit denen des ursprünglichen Programms überein. Funktional ist also kein Vorteil erzielt worden. Zudem ist der Sourcecode-Umfang und die Anzahl der eingesetzten Klassen und Interfaces gestiegen. Die Lösung mit dem STRATEGIE-Muster ist also länger, umfangreicher und komplexer. Wieso ist diese Lösung aber trotzdem in der Regel besser? Die Antwort ist einfach: Diese Lösung ist modularer und leichter erweiterbar. Der Vorgang der Filterung und der tatsächlichen Auswahl sind voneinander unabhängig und lediglich über das Interface `FilterStrategy` verbunden. Eine Einschränkung auf eine spezielle Art der Filterung ist lediglich durch die `acceptValue(int)`-Methode gegeben. Zudem ist die Lösung deutlich objektorientierter, da nun eigenständige Klassen die Auswahl realisieren und weitere Funktionalität, beispielsweise die Aufbereitung einer aussagekräftigen Stringrepräsentation mithilfe der Methode `toString()`, zur jeweiligen Filterstrategie anbieten können. Ohne Strategieobjekte wären für eine derartige textuelle Ausgabe der gewählten Filterstrategie jedes Mal umfangreiche Prüfungen mit diversen `if`-Anweisungen durchzuführen, die ähnlich der Auswahl des Filterkriteriums selbst sind. Dies ist fehlerträchtig. Durch Strategieobjekte wird diese unnötige Komplexität vermieden. Jedes einzelne Objekt kann die `toString()`-Methode überschreiben, um eine informative Ausgabe zu liefern.

Falls Sie noch nicht überzeugt sein sollten, kommt das Beste nun zum Schluss: Durch Einsatz des STRATEGIE-Musters können komplett neue Filterstrategien implementiert werden, die insbesondere auch nicht auf die ursprüngliche Schnittstelle mit den zwei `int`-Werten zur Bereichsprüfung eingeschränkt sind. Ein erstes Beispiel dafür ist etwa eine Filterung auf alle geraden Zahlen, deren Klasse `EvenFilter` wir auch wieder in der Klasse `FilterStrategies` implementieren:

```
public static class EvenFilter implements FilterStrategy
{
    @Override
    public boolean acceptValue(final int value)
    {
        return value % 2 == 0;
    }

    @Override
    public String toString()
    {
        return "EvenFilter";
    }
}
```

Algorithmisch etwas anspruchsvoller ist eine Filterung von Primzahlen:

```
public final class PrimeFilter implements FilterStrategy
{
    @Override
    public boolean acceptValue(final int value)
    {
        return isPrime(value);
    }

    private boolean isPrime(final int value)
    {
        if (value < 2)
            return false;

        if (value == 2)
            return true;

        for (int i = 2; i <= Math.sqrt(value); i++)
        {
            // Test auf Teilbarkeit
            if ((value % i) == 0)
                return false;
        }
        return true;
    }

    @Override
    public String toString()
    {
        return "PrimeFilter";
    }
}
```

Wären diese Erweiterungen in der ursprünglichen Klasse zu realisieren gewesen, dann wäre dort bereits ein ziemliches Chaos entstanden.

Noch stärker wäre der Effekt, wenn Erweiterungswünsche aufkommen: Stellen Sie sich vor, dass die inverse Auswahloperation realisiert werden soll. In der ursprünglichen Lösung würde diese Forderung spätestens jetzt zu sehr unübersichtlichem Sourcecode führen: Jede weitere Änderung ist fehlerträchtig und bestenfalls noch durch eine große Anzahl an Unit Tests zu beherrschen. Auch hierbei hilft der Einsatz des STRATEGIE-Musters, um den Implementierungs- und Testaufwand erheblich zu reduzieren. Man nutzt zudem das aus Abschnitt 18.2.3 bekannte DEKORIERER-Muster und implementiert eine allgemeingültige inverse Filterstrategie `InverseFilter`. Damit ergibt sich das in Abbildung 18-24 gezeigte Klassendiagramm.

Abbildung 18-24 *Das Interface* `FilterStrategy` *und das* DEKORIERER-*Muster*

Die Implementierung eines inversen Filters durch die Klasse `InverseFilter` geschieht wie folgt:

```
public final class InverseFilter implements FilterStrategy
{
    private final FilterStrategy originalFilterStrategy;

    public InverseFilter(final FilterStrategy originalFilterStrategy)
    {
        this.originalFilterStrategy =
                      Objects.requireNonNull(originalFilterStrategy,
                      "parameter 'originalFilterStrategy' must not be null!");
    }

    @Override
    public boolean acceptValue(final int value)
    {
        return !originalFilterStrategy.acceptValue(value);
    }

    @Override
    public String toString()
    {
        return "InverseFilter " + originalFilterStrategy;
    }
}
```

Betrachten wir nun, mit wie wenig Aufwand die inversen Filter »ungerade Zahl« und »keine Primzahl« aus den ursprünglichen Filtern `EvenFilter` und `PrimeFilter` rea-

lisiert werden können, wenn man die gerade entwickelte Dekoriererklasse `InverseFilter` einsetzt:

```
public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    // EvenFilter und OddFilter (InverseEvenFilter)
    final FilterStrategy evenFilter = new EvenFilter();
    System.out.println(evenFilter + ": " + filterAll(inputs, evenFilter));

    final FilterStrategy oddFilter = new InverseFilter(evenFilter);
    System.out.println(oddFilter + ": " + filterAll(inputs, oddFilter));

    // PrimeFilter und InversePrimeFilter
    final FilterStrategy primeFilter = new PrimeFilter();
    System.out.println(primeFilter + ": " + filterAll(inputs, primeFilter));

    final FilterStrategy inversePrimeFilter = new InverseFilter(primeFilter);
    System.out.println(inversePrimeFilter + ": " + filterAll(inputs,
          inversePrimeFilter));
}
```

Listing 18.9 *Ausführbar als* **'STRATEGYFILTEREXAMPLE2'**

Startet man das Programm STRATEGYFILTEREXAMPLE2, so kommt es zu folgender Ausgabe:

```
EvenFilter: [2, 4, 6, 8]
InverseFilter EvenFilter: [1, 3, 5, 7, 9]
PrimeFilter: [2, 3, 5, 7]
InverseFilter PrimeFilter: [1, 4, 6, 8, 9]
```

Bewertung

An diesem Beispiel sieht man deutlich, welche Vorteile der bewusste und gezielte Einsatz des für ein zu lösendes Problem passenden Entwurfsmusters bringen kann. Der Einsatz des STRATEGIE-Musters bewirkt Folgendes:

+ **Erhöhte Flexibilität** – Die Auswahl aus verschiedenen Implementierungen kann zur Laufzeit erfolgen. Dadurch erhöht sich die Flexibilität und die Wiederverwendbarkeit.
+ **Bessere Erweiterbarkeit** – Die Alternativen besitzen keine Abhängigkeiten untereinander, sodass sich verschiedene Anforderungen (Bereichsgrenzen, Primzahlen usw.) unabhängig realisieren lassen. Zudem kann man zusätzliche Anforderungen, wie für die inverse Abbildung gezeigt, leicht und gezielt integrieren.
o **Kopplung an konkrete Strategien** – Klienten müssen die konkreten Strategieklassen kennen und instanziieren, um die gewünschte Funktionalität auszuführen. Eine Kombination mit einer FABRIKMETHODE (vgl. Abschnitt 18.1.2) bietet sich zur Kapselung und loseren Kopplung an.
- **Erhöhter Umfang** – Die Anzahl der Klassen erhöht sich.

Vereinfachungen mit JDK 8

Das bisher Erreichte ist wirklich eine schöne Demonstration, wie Entwurfsmuster das Entwicklerleben vereinfachen können. Praktischerweise sind bereits einige Ideen auch in die Neuerungen von Java 8 eingeflossen. Viele Aufgabenstellungen, vor allem das Filtern, lassen sich mit dem Filter-Map-Reduce-Framework (vgl. Abschnitt 7.2) sowie dem Functional Interface `Predicate<T>` (vgl. Abschnitt 4.5.1) elegant in Java umsetzen. Nachfolgend wollen wir uns anschauen, wie wir die Beispiele schrittweise umformen und von den Neuerungen aus Java 8 profitieren können. Die Ideen des STRATEGIE- und DEKORIERER-Musters bleiben weiter gültig, aber lassen sich nun anders nutzen.

Ausgangsbasis 1: Die Klasse STRATEGYFILTEREXAMPLE Schauen wir uns zunächst zur Erinnerung noch mal die `main()`-Methode an, die Zahlen auf ein geschlossenes und offenes Intervall filtert:

```
public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    System.out.println("Filtering values for interval 2-7");
    final FilterStrategy closedInterval = new ClosedInterval(2, 7);
    System.out.println("Using " + closedInterval + " " +
                       filterAll(inputs, closedInterval));

    final FilterStrategy openInterval = new OpenInterval(2, 7);
    System.out.println("Using " + openInterval + " " +
                       filterAll(inputs, openInterval));
}
```

Schritt 1: Stream-API und `Predicate<T>` nutzen In einem ersten Schritt modifizieren wir das Programm STRATEGYFILTEREXAMPLE so, dass das Stream-API und das Functional Interface `Predicate<T>` in der Methode `filterAll()` zum Einsatz kommen. Zunächst bleibt die Signatur gleich und wir wandeln mit einem kleinen Trick übergebene Instanzen vom selbst erstellten Interface `FilterStrategy` in ein `Predicate<T>` um:

```
public static List<Integer> filterAll(final List<Integer> inputs,
                                      final FilterStrategy filterStrategy)
{
    // Konvertierung FilterStrategy -> Predicate
    final Predicate<Integer> predicate =
                             value -> filterStrategy.acceptValue(value);

    return inputs.stream().
                  filter(predicate).
                  collect(Collectors.toList());
}
```

Schritt 2: Vollständig auf `Predicate<T>` umstellen Als Nächstes wollen wir durchgängig das `Predicate<T>` nutzen. Weil dies die Signatur der Methode ändert, nutzen wir das Refactoring EXTRACT METHOD, um eine auf `Predicate<T>` spezialisierte Variante von `filterAll()` herauszulösen. Zum Aufräumen nutzen wir noch das Refactoring INLINE, wodurch das Ganze noch kompakter wird:

```
public static List<Integer> filterAll(final List<Integer> inputs,
                                      final FilterStrategy filterStrategy)
{
    return filterAll(inputs, value -> filterStrategy.acceptValue(value));
}

private static List<Integer> filterAll(final List<Integer> inputs,
                                       final Predicate<Integer> predicate)
{
    return inputs.stream().
                  filter(predicate).
                  collect(Collectors.toList());
}
```

Die Methoden zum Filtern haben wir umgestellt. Als Nächstes schauen wir uns die Aufrufstelle in der `main()`-Methode an. Nun können wir (zumindest in IntelliJ IDEA) das Basis-Refactoring INLINE nutzen oder wir erzeugen uns eine künstliche Hilfsvariable `closedIntervalPred` wie folgt:

```
final FilterStrategy closedInterval = new ClosedInterval(2, 7);
final Predicate<Integer> closedIntervalPred =
                         value -> closedInterval.acceptValue(value);
System.out.println("Using " + closedInterval + " " +
                   filterAll(inputs, closedIntervalPred));
```

An der Aufrufstelle können wir nun nur noch `Predicate<T>` verwenden und müssen die Variable `closedInterval` entfernen – bevor wir dazu das Basis-Refactoring INLINE nutzen können, muss die Konsolenausgabe leicht angepasst werden:

```
final Predicate<Integer> closedIntervalPred =
                         value -> new ClosedInterval(2, 7).acceptValue(value);
System.out.println("Using closedInterval" +
                   filterAll(inputs, closedIntervalPred));
```

Schritt 3: Eigene Lambdas definieren Schließlich wollen wir die Klassen `ClosedInterval` und `OpenInterval` vollständig durch Lambdas ersetzen: Spontan ist man versucht, für das geschlossene Intervall etwa Folgendes zu schreiben:

```
final Predicate<Integer> closedInterval =
                         value -> lowerBound <= value && value <= upperBound;
```

Doch das schlägt fehl, weil die Variablen `lowerBound` und `upperBound` (natürlich) nicht definiert sind. Das zeigt uns, dass man in Lambdas (logischerweise) nur mit Parametern und gegebenenfalls umgebenden »effectively final«-Variablen arbeiten kann. Es gibt aber eine trickreiche Abhilfe, die so einfach wie elegant ist, nämlich die Definition von Hilfsmethoden:

```
static Predicate<Integer> closedInterval(final int lowerBound,
                                         final int upperBound)
{
    if (upperBound < lowerBound)
        throw new IllegalArgumentException("lowerBound must be <= upperBound");

    return value -> lowerBound <= value && value <= upperBound;
}

static Predicate<Integer> openInterval(final int lowerBound,
                                       final int upperBound)
{
    if (upperBound < lowerBound)
        throw new IllegalArgumentException("lowerBound must be <= upperBound");

    return value -> lowerBound < value && value < upperBound;
}
```

Unter deren Verwendung vereinfachen wir die `main()`-Methode wie folgt:

```
public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
    System.out.println("Filtering values for intervall 2-7");

    final Predicate<Integer> closedInterval = closedInterval(2, 7);
    System.out.println("Using " + closedInterval + " " +
                       filterAll(inputs, closedInterval));

    final Predicate<Integer> openInterval = openInterval(2, 7);
    System.out.println("Using " + openInterval + " " +
                       filterAll(inputs, openInterval));
}
```

Listing 18.10 *Ausführbar als* **'STRATEGYFILTEREXAMPLEJDK8_STEP3'**

Ausgangsbasis 2: STRATEGYFILTEREXAMPLE2 Schauen wir uns zunächst zur Erinnerung noch mal die `main()`-Methode an, die Zahlen auf gerade und ungerade Zahlen sowie auf Primzahlen und Nicht-Primzahlen filtert:

```
public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    final FilterStrategy evenFilter = new EvenFilter();
    System.out.println(evenFilter + ": " + filterAll(inputs, evenFilter));

    final FilterStrategy oddFilter = new InverseFilter(evenFilter);
    System.out.println(oddFilter + ": " + filterAll(inputs, oddFilter));

    final FilterStrategy primeFilter = new PrimeFilter();
    System.out.println(primeFilter + ": " + filterAll(inputs, primeFilter));

    final FilterStrategy inversePrimeFilter = new InverseFilter(primeFilter);
    System.out.println(inversePrimeFilter + ": " +
                       filterAll(inputs, inversePrimeFilter));
}
```

Diverse Vereinfachungen vornehmen Nun wollen wir die bisher gemachten Erfahrungen nutzen, um auch das zweite Beispiel in Form des Programms STRATEGYFILTEREXAMPLE2 mit Java-8-Bordmitteln zu vereinfachen. Dabei wenden wir folgende Transformationsideen an:

- `EvenFilter` – Lässt sich durch den Lambda `value -> value % 2 == 0` ersetzen, weil hier nur auf dem übergebenen Parameter gearbeitet wird und keine weiteren Kontextinformationen benötigt werden.
- `InverseFilter` – Zum Invertieren eines `Predicate<T>` benötigt man keine eigene Klasse, sondern ruft die Methode `negate()` auf (vgl. Abschnitt 4.5.1).
- `PrimeFilter` – Hier wenden wir den bereits bekannten Trick an, die ursprüngliche Funktionalität über einen Lambda anzusprechen – bei Bedarf kann man das dann auch noch auslagern.

Mit diesen Anmerkungen im Hinterkopf modifizieren wir das Programm leicht (vor allem durch direkte Angabe der Namen der Filter in den Konsolenausgaben):

```
public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    final Predicate<Integer> evenFilter = value -> value % 2 == 0;
    System.out.println("EvenFilter: " + filterAll(inputs, evenFilter));

    final Predicate<Integer> oddFilter = evenFilter.negate();
    System.out.println("OddFilter: " + filterAll(inputs, oddFilter));

    final Predicate<Integer> primeFilter =
                             value -> new PrimeFilter().acceptValue(value);
    System.out.println("PrimeFilter: " + filterAll(inputs, primeFilter));

    final Predicate<Integer> inversePrimeFilter = primeFilter.negate();
    System.out.println("InversePrimeFilter: " +
                       filterAll(inputs, inversePrimeFilter));
}
```

Listing 18.11 *Ausführbar als* **'STRATEGYFILTEREXAMPLE2_JDK8'**

Fazit

Anhand zweier Beispiele konnten wir gut nachvollziehen, dass man durch Einsatz der adäquaten Java-8-Bordmittel ehemals aufwendigen Sourcecode deutlich vereinfachen kann. Auch muss man die Entwurfsmuster nicht mehr in dem Umfang selbst ausprogrammieren – aber es ist weiterhin hilfreich, sie verstehen und anwenden zu können.

Ganz nebenbei haben wir noch einmal ein paar Ideen zu Refactorings und kleinteiligen Schritten rekapituliert und einige Besonderheiten und Tricks beim Einsatz von Lambdas kennengelernt, etwa wie man Parametrierungen vornehmen oder alte Implementierungen geschickt wiederverwenden kann.

18.3.5 Befehl (Command)

Beschreibung und Motivation

Das Muster BEFEHL vereinfacht die Ausführung komplexerer Aufgaben. Da dieses Muster nicht so leicht zu verstehen ist, möchte ich es anhand eines Beispiels aus der Realität, einer Bestellung in einem Restaurant, verdeutlichen. Ein Gast gibt eine Bestellung auf, etwa: »Ein Steak medium mit Pommes und ein Bier«. Der Kellner notiert diese Bestellung und übergibt sie der Küche und der Theke als Arbeitsanweisungen zur Ausführung. Die ausführenden Einheiten können anhand der Bestellung verschiedene konkrete kleinere Handlungen durchführen (Braten, Frittieren, Bier zapfen usw.), die für den Gast aber uninteressant sind (etwa die Temperatureinstellung am Herd).

Auf die Software übertragen erkennen wir folgende Akteure bei diesem Muster: Ein Klient (Gast) gibt eine Bestellung (einen oder mehrere (abstrakte) Befehle) auf, die von einer Komponente (Kellner) registriert und von anderen Komponenten (Küche und Theke) ausgeführt wird. Ein klassisches Beispiel sind Aktionen eines Editors, die durch Menüs, Kontextmenüs oder Toolbars ausgelöst werden und beispielsweise zum Speichern eines Dokuments oder zum Erzeugen einer grafischen Figur führen.

Struktur

Dieses Muster existiert in diversen, unterschiedlich komplexen Ausprägungen, jedoch findet immer eine Kapselung von Methodenaufrufen in Befehlsobjekten statt. Dadurch erreicht man eine Trennung von einem Befehlswunsch und dessen Ausführung. Die Basis bildet normalerweise ein Interface `ICommand`, das standardmäßig eine Methode `execute()` definiert. Jedes konkrete Befehlsobjekt implementiert das Interface `ICommand` und bietet somit eine einfache und für alle Befehlsklassen einheitliche Schnittstelle zur Ausführung an. Jede konkrete Realisierung eines Befehls, im nachfolgenden Beispiel `PrintTextCommand` bzw. `WaitCommand`, implementiert die Methode `execute()` entsprechend der gekapselten Aufgabe, indem dort die Verarbeitung durch feingranulare Methodenaufrufe stattfindet. Dies zeigt Abbildung 18-25.

Abbildung 18-25 *Struktur des Musters* BEFEHL *in UML*

Die Komplexität der im Befehlsobjekt realisierten konkreten Aufgabe ist für Klienten zum Anstoßen einer Befehlsausführung irrelevant. Ein Klient benötigt darüber hinaus auch kein Wissen über das konkrete API der im Befehlsobjekt verwendeten Programmteile, da diese dort gekapselt werden. Daher kann man dieses Muster gut zur Entkopplung von Vorgängen (sprich des befehlsaufrufenden und des befehlsempfangenden Programmteils) einsetzen.

Beispiel

Ich stelle das Muster BEFEHL zunächst in seiner einfachsten Form dar, um es schrittweise zu erweitern und auf mögliche Unzulänglichkeiten und Verbesserungen aufmerksam zu machen.

Einfachste Form Der folgende Sourcecode zeigt einen simplen Ablauf. Es werden zunächst drei Befehlsobjekte erzeugt und anschließend ausgeführt:

```
public static void main(final String[] args)
{
    // Erzeugen der Command-Objekte
    final ICommand command1 = new PrintTextCommand("Dies ist ein Text");
    final ICommand command2 = new WaitCommand();
    final ICommand command3 = new PrintTextCommand("Der Test ist beendet!");

    // Ausführen der Command-Objekte
    command1.execute();
    command2.execute();
    command3.execute();
}
```

Listing 18.12 *Ausführbar als* **'SIMPLECOMMANDEXAMPLE'**

Betrachten wir dieses einführende Beispiel, so sehen wir, dass die Funktionalität in den jeweiligen Befehlsklassen gekapselt wird. Dies ist bereits gut, könnte allerdings auch durch die Extraktion einer Methode erreicht werden. In diesem Fall wird zudem der Befehl direkt vom Klienten ausgeführt. Dies kann in einigen wenigen Situationen sinnvoll sein. In der Regel rufen jedoch Klienten die `execute()`-Methode nicht direkt auf. Vielmehr übergeben Klienten Befehle zur Ausführung an eine separate Verarbeitungseinheit, die dann `execute()` aufruft.

Erweiterung um eine Ausführungskomponente Wir werden nun dieses einfache Modell dahingehend erweitern, dass eine ausführende Komponente hinzugefügt wird. Klienten führen hier die Befehle nicht mehr selbst aus, sondern übergeben diese an die Ausführungskomponente. Diese speichert die auszuführenden Befehle in einer listenähnlichen Struktur. Die im folgenden Listing gezeigte Klasse `CommandExecutor` nutzt dazu eine `LinkedBlockingDeque<E>` zur Thread-sicheren Bearbeitung. Die Implementierung der Klasse `CommandExecutor` als `Runnable` ermöglicht es, diese in einem Thread abzuarbeiten. Parallel dazu findet die Programmausführung der Klienten statt. Diese stellen bei Bedarf Befehle zur Ausführung ein.

```
public final class CommandExecutor implements Runnable
{
   private final Deque<ICommand> commands = new LinkedBlockingDeque<>();

   private static final ICommand NULL_COMMAND = new ICommand()
   {
       @Override
       public void execute()
       {
       }

       @Override
       public String toString()
       {
           return "NULL_COMMAND";
       }
   };
```

Im vorherigen Listing wird zudem ein statisches Attribut `NULL_COMMAND` gemäß dem NULL-OBJEKT-Muster (vgl. Abschnitt 18.3.2) erzeugt, das wir im Folgenden nutzen werden, um bei der Bearbeitung der Befehlsliste `null`-Abfragen zu vermeiden.

Das folgende Listing setzt das Beispiel fort und zeigt, wie Befehlsobjekte von Klienten mithilfe der Methode `registerCommand(ICommand)` als Bearbeitungswunsch eingetragen werden können. Parallel dazu werden Befehlsobjekte in der Methode `run()` sequenziell abgearbeitet. Dort wird das nächste auszuführende Befehlsobjekt mithilfe der Methode `getAndRemoveNextCommand()` ermittelt und per Aufruf von `execute()` ausgeführt. Ist kein solches Befehlsobjekt verfügbar, erspart der Einsatz des `NULL_COMMAND` eine Spezialbehandlung in der Abarbeitung:

```
public void registerCommand(final ICommand commandToExecute)
{
    Objects.requireNonNull(commandToExecute, "Passed command must not be null");

    commands.offer(commandToExecute);
}

public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        final ICommand commandToExecute = getAndRemoveNextCommand();
        commandToExecute.execute();

        SleepUtils.safeSleep(50); // Vermeide CPU-Belastung
    }
}

private ICommand getAndRemoveNextCommand()
{
    final ICommand commandToExecute = commands.poll();
    if (commandToExecute == null)
        return NULL_COMMAND;

    return commandToExecute;
}
```

Im folgenden Programmausschnitt wird der `CommandExecutor` genutzt, um die aus dem vorherigen Beispiel bekannten drei Befehle abzuarbeiten. Im Unterschied zur simplen Variante erkennt man, dass hier kein Aufruf von `execute()` durch den Klienten erfolgt, sondern lediglich ein Bearbeitungswunsch durch `registerCommand(ICommand)` ausgedrückt wird:

```
public static void main(final String[] args)
{
    // Die Abarbeitung der Kommandos erfolgt nebenläufig
    // zu den folgenden Aktionen
    final CommandExecutor executor = new CommandExecutor();
    new Thread(executor).start();

    // Client erzeugt Kommandos
    final ICommand command1 = new PrintTextCommand("Dies ist ein Text");
    final ICommand command2 = new WaitCommand();
    final ICommand command3 = new PrintTextCommand("Der Test ist beendet!");

    // Client übergibt Kommandos an Executor
    executor.registerCommand(command1);
    executor.registerCommand(command2);
    executor.registerCommand(command3);

    // wait and quit
    Thread.sleep(7_000);
    executorThread.interrupt();
}
```

Listing 18.13 *Ausführbar als* '**COMMANDEXECUTOR**'

Nach der Erweiterung um den `CommandExecutor` sieht das Klassendiagramm wie in Abbildung 18-26 aus.

Abbildung 18-26 *Command und Executor in UML*

Ausführungskontext

Bislang sind die vorgestellten `execute()`-Methoden funktional sehr einfach. In der Praxis benötigen Befehle meistens einen Kontext oder Eingabeparameter zu ihrer Ausführung bzw. Daten, um auf diesen zu operieren. Selbst für das einfache `PrintTextCommand` ist ein Parameter zur Übergabe des auszugebenden Texts erforderlich.

Sind zur Ausführung benötigte Informationen bereits zum Erzeugungszeitpunkt der Befehlsobjekte bekannt, sollten diese als Parameter an deren Konstruktor übergeben werden. Auf diese Weise kann man verschiedene Rahmenbedingungen und Eigenschaften festlegen. Bezogen auf das Beispiel einer Bestellung im Restaurant sind dies beispielsweise die Namen des Gerichts und des Getränks.

Allerdings sind mitunter einige Informationen, etwa die konkrete Umgebung der Ausführung oder die ausführende Einheit, eventuell noch unbekannt. Dies sind Detailinformationen, die zur tatsächlichen Abarbeitung eines Befehls, also von dessen `execute()`-Methode, benötigt werden. Bezogen auf das Restaurantbeispiel kennt der Gast wesentliche Details der Zubereitung nicht: Welches Stück Fleisch gebraten wird und welcher Koch die Zubereitung übernimmt, wird erst in der Küche entschieden.

Aus diesen beiden Randbedingungen folgt, dass es in der Regel zwei Arten von Parametrierungen eines Befehls gibt:

1. **Spezifikationsdaten** – Einige Parameter werden genutzt, um eine konkrete Spezifikation des Befehls und damit eine Präzisierung des Auftrags vorzunehmen: Es wird das »Was« beschrieben, etwa: »Steak medium mit Pommes«. Diese Daten werden von Klienten festgelegt.
2. **Ausführungsdaten** – Um den gewünschten Auftrag auszuführen und aus der zuvor angegebenen Spezifikation eine konkrete Abarbeitung des Befehls zu ermöglichen, werden in der Regel weitere Daten und Akteure benötigt, die an den Befehl übergeben werden müssen. Es wird das »Wie« und »Wer« beschrieben, etwa: »Welcher Koch die Arbeit übernimmt, welches Steak zubereitet wird, welcher Grill verwendet wird.« Auf diese Daten hat ein Klient in der Regel keinen Einfluss.

Die Spezifikationsdaten (Daten zur Präzisierung des Auftrags) sollten von Klienten möglichst zum Konstruktionszeitpunkt eines Befehlsobjekts übergeben und in dessen Attributen gespeichert werden.

Wie bereits angedeutet, wird ein Befehl zur Ausführung seiner `execute()`-Methode Zugriff auf diverse andere Objekte und Informationen benötigen. Zur Vereinfachung ist es sinnvoll, die erforderlichen Daten als Laufzeitkontext in Form einer Value-Object-Klasse (vgl. Abschnitt 3.4.5) `ExecutionContext` zu bündeln. Zur Aufbereitung dieser Informationen ist ein Zugriff auf alle beteiligten Klassen nötig, um deren Referenzen im Value Object speichern zu können. Wann und wo können diese Informationen dem Befehlsobjekt übergeben werden? In einigen Applikationen kann man diese den Befehlsobjekten bereits zu ihrem Konstruktionszeitpunkt übergeben. Häufig stehen die benötigten Informationen den Erzeugern der Befehlsobjekte (z. B. Menüeinträgen) jedoch nicht ohne Weiteres zur Verfügung. Dann müssten die

Daten für die Erzeuger nur deshalb zugänglich gemacht werden, um die Befehlsobjekte zum Konstruktionszeitpunkt mit dem notwendigen Kontext versorgen zu können. Dadurch würde die Anzahl der Parameter und Abhängigkeiten zunehmen. Das stört die Kapselung und Wiederverwendbarkeit. Wenn ein Befehlsobjekt von verschiedenen Programmteilen erzeugt werden kann (z. B. über ein Menü, ein Kontextmenü, einen Button oder Shortcut), muss jedes dieser Programmteile all diese – ansonsten nicht benötigten – Informationen bereitstellen können.

Einfacher und klarer ist es deshalb meistens, diese Aufgabe der Informationsbereitstellung durch ein Objekt vom Typ `CommandExecutor` erledigen zu lassen. Es bietet sich folgender Trick an, der die Ideen der sogenannten ***Dependency Injection*** aufgreift: Der Methode `execute()` werden die Spezifikationsdaten bzw. Kontextdaten als Klasse `ExecutionContext` übergeben. Diese stellt dann alle zur Ausführung benötigten Informationen zur Verfügung. Die Konstruktion von Befehlsobjekten kann nun unabhängig von der Bereitstellung der Ausführungsdaten erfolgen.

Tipp: Bereitstellung von Ausführungsdaten

Werden Applikationen komplexer oder sollen Befehle in mehreren Kontexten ausgeführt werden, so bietet sich die Übergabe eines `ExecutionContext`s an die Methode `execute()` an. Nur so ist eine Unabhängigkeit von Befehlserzeugung und deren Ausführung möglich. Das gilt insbesondere dann, wenn Befehle in einer JVM erzeugt und in einer anderen JVM ausgeführt werden sollen.

Abbildung 18-27 *Das Muster* BEFEHL *in einem grafischen Editor*

Erweiterungen

Auch eine Undo-Funktionalität ist mithilfe des Musters BEFEHL leichter zu realisieren als ohne. Im einfachsten Fall speichert man bei jedem Ausführen eines Befehls zuvor den aktuellen Zustand des Gesamtsystems ab und kann diesen bei einem Undo wiederherstellen. Das ist aber in der Regel zu aufwendig. Daher wird bevorzugt mit relevanten, von dieser Aktion betroffenen Ausschnitten des Zustands gearbeitet. Undo ist daher in realen Systemen immer mit einigem Aufwand verbunden.

Leichter als eine Undo-Funktionalität lassen sich Befehle als Makros (Kommandofolgen) zusammenfassen und als Einheit ausführen. Durch eine Speicherung einer Abfolge von Befehlen kann man eine Art Befehlsrekorder bauen, der ein späteres Abspielen erlaubt. Die Makros können wir mithilfe des KOMPOSITUM-Musters (vgl. Abschnitt 18.2.4) und durch eine Klasse realisieren.

Bewertung

Der Einsatz des Musters BEFEHL bewirkt Folgendes:

+ **Lose Kopplung** – Durch die Kapselung von Aktionen in Form von Objekten können diese von beliebigen Programmteilen erzeugt und von anderen Programmteilen ausgeführt werden. Funktionalität lässt sich auf verschiedene Wege bereitstellen, etwa als Reaktion auf Menüs, Buttons oder andere Eingaben, ohne dass es zu einer Kopplung an konkrete Elemente der Benutzerschnittstelle kommt: Aufrufer und ausführende Einheit sind nicht miteinander verbunden, wie dies bei einem Methodenaufruf der Fall wäre.
+ **Bessere Abstraktion** – In einem Befehl werden verschiedene auszuführende Methodenaufrufe an einer Stelle gebündelt. Dies ist ähnlich zu der Extraktion einer Methode, geht allerdings einen Schritt weiter, da ein neues, in anderen Kontexten wiederverwendbares Befehlsobjekt entsteht.
+ **Wiederverwendbarkeit** – Ein Befehl stellt eine Verhaltensbeschreibung, eine Art semantische Klammer über verschiedene Methoden, dar und ist vielseitig einsetzbar. Ein Befehl kann von unterschiedlichen Auftraggebern (Menü, Kontextmenü, Toolbar usw.) an verschiedene ausführende Einheiten weitergereicht und dort in diversen Kontexten (Einzelselektion, Mehrfachselektion usw.) ausgeführt werden.
+ **Fernsteuerung** – Eine Speicherung und spätere Ausführung von Befehlen ist möglich: Befehlsobjekte können wie alle anderen Objekte auch verarbeitet oder in Methodenaufrufen verwendet werden. Im Speziellen lassen sich diese serialisieren und zu späteren Zeitpunkten erneut ausführen. Daher kann dieses Muster gut zur zeitgesteuerten Fernsteuerung eingesetzt werden. Tatsächlich sind auch andere Programme – die sogar in eigenständigen JVMs laufen können – in der Lage, die Befehle zu verarbeiten.

+ **Undo-/Redo-Fähigkeit** – Eine Undo-/Redo-Fähigkeit ist leichter mit dem Muster BEFEHL zu realisieren als ohne. Zu definierten Zeitpunkten können Zwischenstände des zugrunde liegenden Modells gesichert werden, die bei Bedarf wiederhergestellt werden können.
- **Erhöhter Umfang** – Die Anzahl der Klassen erhöht sich.

18.3.6 Proxy

Beschreibung und Motivation

Das PROXY-Muster dient zum Verlagern der Kontrolle über ein Objekt auf ein Stellvertreterobjekt, den sogenannten ***Proxy***. Ein Klient spricht den Stellvertreter an und kennt das eigentliche Objekt nicht. Dadurch kann der Proxy sowohl die Erzeugung als auch den Zugriff auf das eigentliche Objekt kontrollieren und somit eine Zugangskontrolle realisieren oder Remote Calls verstecken. Außerdem lässt sich ein Proxy zur Vereinfachung von Lazy Initialization einsetzen.

Struktur

Die Basis dieses Musters ist eine gemeinsame Schnittstelle `IService`. Diese stellt sicher, dass sowohl der Proxy in Form der Klasse `ServiceProxy` als auch das eigentliche Objekt vom Typ `Service` nach außen gleich behandelt werden können. Zur Durchführung der eigentlichen Aufgaben verwaltet der Proxy eine Referenz auf das Objekt. Falls angebracht, kann man sogar die Erzeugung des Objekts an den Proxy delegieren.

Abbildung 18-28 *Struktur des* PROXY-*Musters in UML*

Varianten

Das PROXY-Muster existiert unter anderem in folgenden konkreten Ausprägungen, die verschiedene Intentionen verfolgen:

- **Access Control Proxy** – Kann den Zugriff auf gewisse Daten steuern. Beispielsweise lässt sich vor jeder Methodendelegation eine Rechteprüfung ausführen, die bei Bedarf die Eingabe eines Passworts verlangt.
- **Decorator/Interceptor Proxy** – Dieser Typ von Proxy fügt transparent weitere Funktionalität gemäß dem DEKORIERER-Muster hinzu. Häufig spricht man in diesem Kontext auch von INTERCEPTOR.
- **Lazy Init Proxy** oder **Virtual Proxy** – Vermittelt den Eindruck, ein Objekt stehe schon zur Verfügung, bevor es tatsächlich erzeugt wurde. Objektbestandteile werden erst in dem Moment konstruiert, in dem diese auch tatsächlich benutzt werden. Ein virtueller Proxy fungiert als Platzhalter, etwa beim Zugriff auf Objekte, die aufwendig zu konstruieren oder zu laden sind. Man kennt es von Bildern im Internet. Für diese werden im Browser teilweise zunächst Platzhalter dargestellt, bis diese nach erfolgtem Ladevorgang durch das entsprechende Bild ersetzt werden. Wenn man optionale Funktionalität oder besonders aufwendig zu konstruierende Daten mit einem solchen Proxy kapselt, kann dies zu signifikanten Performance-Verbesserungen führen. Dies wird in Abschnitt 22.3 detailliert beschrieben.
- **Remote Proxy** – Der Remote Proxy agiert als Stellvertreter für Objekte, die über ein Netzwerk (remote) angesprochen werden sollen. Für mögliche Nutzer bleibt diese Tatsache (weitgehend) transparent. Im Speziellen müssen sich diese nicht darum kümmern, wo welche Objekte erzeugt werden und wie die Methodenaufrufe über das Netzwerk geregelt werden. Ein Remote Proxy kann den Eindruck erwecken, ein entferntes Objekt wäre ein lokales. Dies ist zum Verbergen des Einsatzes von RMI (Remote Method Invocation) oder Webservices praktisch.

Beispiel

Als Beispiel realisieren wir einen Access Control Proxy, der den Zugriff auf eine Map steuern soll. Dazu implementiert die folgende Proxy-Klasse `RestrictedAccessMap` das Interface `Map<K,V>` und erweitert die ursprüngliche Realisierung um eine Prüfung auf Zugriffsrechte. Das geschieht in einer Methode `ensureAccessGranted()`. Diese Methode wird vor jeder Methodendelegation an die zugrunde liegende Map aufgerufen, damit sichergestellt, dass der gewünschte Zugriff erlaubt ist. Hat ein Benutzer keine Zugriffsrechte, so erfolgt kein Zugriff auf die Map und es wird eine selbst definierte `InvalidAccessRightsException` ausgelöst. Diese muss vom Typ `RuntimeException` sein, da in den Signaturen der Methoden des Interface `Map<K,V>` keine Checked Exceptions definiert sind.

```
public class RestrictedAccessMap<K, V> implements Map<K, V>
{
    private final Map<K, V> underlyingMap = new HashMap<>();

    public void ensureAccessGranted() throws InvalidAccessRightsException
    {
        // ...
    }

    @Override
    public V put(final K key, final V value)
    {
        ensureAccessGranted();
        return underlyingMap.put(key, value);
    }

    ...
}
```

Übrigens könnte man das Ganze auch als Dekorator lösen und die zu schützende Map als Parameter bereitstellen. Wie fast immer, gibt es viele Wege zum Ziel.

Bewertung

Der Einsatz des PROXY-Musters bewirkt Folgendes:

+ **Steuerung von Funktionalität** – Ähnlich zum DEKORIERER-Muster lässt sich die eigentliche Anwendungsfunktionalität um weitere Funktionen ergänzen. Bei diesem Muster steht jedoch der steuernde Charakter im Vordergrund.
o **Aufwand durch Delegation** – Besitzt ein Originalobjekt viele Methoden, so ist die Realisierung des Proxy-Objekts durch die vielen notwendigen Delegationen aufwendig. Hierbei kann die in Abschnitt 12.5 beschriebene Funktionalität der Dynamic Proxies hilfreich sein, die ins JDK integriert ist. Dadurch lässt sich der durch Delegation verursachte Aufwand mitunter deutlich reduzieren.

18.3.7 Beobachter (Observer)

Beschreibung und Motivation

Der Einsatz des BEOBACHTER-Musters ermöglicht es, Änderungen am Zustand eines speziellen Objekts durch andere interessierte Objekte beobachten zu können, ohne dass sich ein Objekt und seine Beobachter direkt kennen müssen. Beobachter melden sich dazu als Interessenten bei einem Objekt an und »abonnieren« damit Informationen über Veränderungen, die ihnen das Objekt in der Folge mitteilt. Sind Interessenten nicht mehr an Änderungen interessiert, so können sie sich abmelden.

Man kann die diesem Muster zugrunde liegende Idee mit dem Abonnement einer Zeitschrift vergleichen. Hat man sich als Abonnent registriert, so erhält man regelmäßig die neuesten Ausgaben, ohne ständig am Kiosk nachschauen zu müssen. Dieses Muster ist auch unter dem Namen ***Publisher/Subscriber*** oder ***Listener*** bekannt.

Ereignisbehandlungen in grafischen Oberflächen basieren in der Regel auf diesem Prinzip – häufig durch Einsatz einer Model-View-Controller-Architektur, die in Abschnitt 18.3.8 beschrieben wird.

Struktur

Um über Änderungen an einem beobachteten Subjekt `ObservedSubject` informiert zu werden, können sich andere, das Interface `ChangeListener` erfüllende Objekte als Änderungsinteressenten bei dem `ObservedSubject` anmelden. Ein zugehöriges Klassendiagramm ist in Abbildung 18-29 gezeigt.

Abbildung 18-29 BEOBACHTER-*Muster mit zwei Beobachtertypen in UML*

Sobald sich der relevante Zustand des `ObservedSubject`s verändert, löst dieses über die Methode `notifyChangeListeners()` eine Benachrichtigungsmitteilung an jeden angemeldeten Beobachter vom Typ `ChangeListener` aus. Dazu wird die Änderung in Form eines Methodenaufrufs propagiert: In den folgenden einführenden Beispielen übernehme ich hier zunächst vereinfachend den in der Literatur gebräuchlichen Namen `update()` für diese Methode. Im Verlauf der Beschreibung dieses Musters werde ich begründen, warum ein solch allgemeiner Name nicht optimal ist, und dann Alternativen mit besser lesbaren und informativeren Methodennamen vorschlagen. Bei einer Änderungsnachricht variiert deren Inhalt in der Art und Anzahl der übertragenen Daten. Man unterscheidet zwischen den im folgenden Abschnitt beschriebenen Varianten ***Pull*** und ***Push***.

Anmerkungen

Nachdem wir den groben Ablauf kennengelernt haben, sollten wir das BEOBACHTER-Muster genauer betrachten, da es einige Fallstricke bereithält, die man unbedingt kennen sollte.

Beobachtung zur Aktualisierung von Darstellungen In der Model-View-Controller-Architektur (vgl. Abschnitt 18.3.8) sorgt das beschriebene Vorgehen bei einer Datenänderung dafür, dass als Beobachter angemeldete Views automatisch aktualisiert werden können. Ein möglicher Ablauf ist in Abbildung 18-30 in Form eines Sequenzdiagramms für das Modell `ObservedSubject` und zwei registrierte Views `ConcreteListener1` bzw. `ConcreteListener2` visualisiert.

Abbildung 18-30 *Ablauf im* BEOBACHTER-*Muster*

Varianten der Information über Zustandsänderungen Um Beobachter über Zustandsänderungen zu informieren, wird eine `update()`-Methode aufgerufen. Dabei gibt es zwei mögliche Varianten zur Übermittlung benötigter Zustandsdaten:

- **Pull** – Jeder einzelne Beobachter holt sich zur Reaktion auf die Änderung benötigte Zustandsdaten beim beobachteten Subjekt selbst ab.
- **Push** – Alle Zustandsdaten des beobachteten Subjekts werden bei Aufruf der Methode `update()` mitgeschickt.

Bei der Pull-Variante dient die `update()`-Methode lediglich als Hinweis, dass eine Änderung stattgefunden hat. Zur Ermittlung und zum Abgleich von Zustandsinformationen werden durch die jeweiligen Beobachter in der Regel diverse `get()`-Methoden zur Zustandsabfrage des `ObservedSubjects` aufgerufen. ***Dies bedingt allerdings, dass jeder Beobachter eine Rückreferenz auf das beobachtete Subjekt besitzt.*** Im Idealfall ist dies lediglich ein Interface, das aus einem Übergabeparameter der `update()`-Methode stammt und nicht im Beobachter gespeichert wird (vgl. Diskussion im Absatz »Mythos lose Kopplung«). Ein Vorteil des Pull-Verfahrens ist, dass der Zustand nur bei Bedarf durch den Beobachter auch tatsächlich nachgefragt wird. Für gewisse Darstellungen kann es bereits ausreichend sein, zu wissen, dass überhaupt eine Änderung stattgefunden hat, um etwa ein '*' für eine veränderte Datei anzuzeigen.

Bei der Push-Variante werden der `update()`-Methode alle benötigten (relevanten) Zustandsinformationen als Parameter mitgegeben. Dadurch werden keine Rückrufaktionen durch die Beobachter erforderlich, wodurch die Kopplung gelöst wird: Beobachter müssen sich lediglich bei »ihrem« Subjekt als Interessent anmelden, dessen sonstige Schnittstelle jedoch nicht kennen. Nachteil dieser Lösung ist allerdings, dass man alle möglicherweise benötigten Informationen ermitteln und versenden muss. ***Da keine Annahmen über Beobachter und von diesen benötigte Anteile des Subjektzustands im Voraus bekannt sind, muss immer der komplette Zustand mitgeteilt werden. Das zu übertragende Datenvolumen kann unter Umständen recht umfangreich sein.***

Verzögerung durch Bearbeitung der Listener Bei der Benachrichtigung der Beobachter besteht das Problem, dass es sich beim Aufruf der Methode `update()` um synchrone Aufrufe an die Beobachter handelt (vgl. Abbildung 18-30). Das bedeutet auch, dass eine weitere Verarbeitung durch Aufruf der `notifyChangeListeners()`-Methode so lange blockiert wird, bis alle Beobachter informiert wurden. Dies kann unter Umständen einige Zeit dauern, wenn einer oder mehrere Beobachter lang andauernde `update()`-Methoden realisieren. Dafür gibt es zwei Möglichkeiten zur Lösung: Man kann die gesamte Benachrichtigung asynchron in einem eigenen Thread ablaufen lassen oder jeden einzelnen `update()`-Aufruf asynchron ausführen. Beides führt aber schnell zu weiteren Synchronisationsproblemen, weil es zu konkurrierenden Lesezugriffen kommt, die dann eventuell auf bereits von anderen Threads aktualisierte Daten zugreifen.

Benachrichtigungsmechanismus bei Multithreading Im Fall einer Zustandsänderung sollen alle angemeldeten Beobachter informiert werden. Dies klingt viel einfacher, als es tatsächlich ist. Betrachten wir dazu folgendes Szenario: Zu beliebigen Zeitpunkten können sich Beobachter nebenläufig an- oder abmelden. Dies könnte theoretisch auch während einer laufenden Benachrichtigung geschehen. Sofort fragt man sich: Wie sorgt man für eine konsistente Beobachterliste bei Multithreading-Zugriffen? Im Prinzip muss man alle Zugriffe auf die Beobachterliste synchronisieren, um Probleme zu verhindern. Das bedeutet allerdings auch, dass Beobachter sich während eines

Benachrichtigungsvorgangs weder an- noch abmelden können, wenn die Realisierung synchronisiert wie in der folgenden Methode erfolgt:

```
private synchronized void notifyChangeListeners()
{
    final Iterator<ChangeListener> it = listeners.iterator();
    while (it.hasNext())
    {
        final ChangeListener listener = it.next();

        listener.update(this);
    }
}
```

In Kapitel 10 haben wir ähnliche Problemstellungen detailliert analysiert. Bei Bedarf nach mehr Parallelität bietet sich der Einsatz von `CopyOnWriteArrayList<ChangeListener>` zur Speicherung der Beobachter an. Eine explizite Synchronisierung wird dadurch überflüssig.

Tipp: Weitere Probleme von Benachrichtigungen

Bei der Realisierung von Benachrichtigungsmethoden gibt es weitere, bisher nicht genannte Probleme:

1. Soll ein während eines Benachrichtigungsvorgangs neu hinzugefügter Beobachter über eine Änderung informiert werden?
2. Müssen sich abmeldende Beobachter, die bereits benachrichtigt wurden, Zustandsänderungen rückgängig machen?

Für beide Fragen gibt es keine allgemeingültige Antwort. Das Verhalten ist applikationsspezifisch. Diese Probleme werden nur extrem selten auftreten, weil An- und Abmeldungen normalerweise nicht während der Beobachtung ausgeführt werden. Die obigen Probleme werden dann häufig akzeptiert, da eine Behandlung nur zu viel mehr Komplexität für relativ wenig Nutzen führen würde. Bezogen auf das Zeitschriftenabonnement hieße dies: Wenn man sich am Tag vor der Versendung von Zeitschriften als Abonnent anmeldet, erhält man in der Regel die aktuelle Ausgabe nicht mehr. Ähnliches gilt für eine kurzfristige Abmeldung: Man wird höchstwahrscheinlich die aktuelle Ausgabe noch zugestellt bekommen.

Mythos lose Kopplung Bei Realisierungen analog zum eingangs gezeigten Beispiel wird oftmals keine lose Kopplung erreicht. Schauen wir auf die Gründe.

Zum einen gibt es eine Abhängigkeit durch die Speicherung von Referenzen der einzelnen `ChangeListener` im beobachteten Subjekt. Da dies jedoch in der Regel in Form eines Interface, also gekapselt, erfolgt, ist das meistens nicht störend.

Zum anderen kommt es zu einer Kopplung von Beobachtern in Richtung Subjekt. Diese ist in der Regel unangenehmer, da Beobachter häufig eine Referenz auf den konkreten Typ des Subjekts nutzen, um nach Änderungsmitteilungen auf den Zustand des Subjekts zuzugreifen. Es kommt deshalb zu einer starken, unidirektionalen Kopplung

zwischen Beobachter und Subjekt. Ein Beobachter kann dadurch sämtliche Methoden der Schnittstelle seines Subjekts aufrufen. ***Eine Verbesserung erreicht man durch Einsatz eines Interface `IDataProvider`, das vom Subjekt zu erfüllen ist und lediglich Zugriff auf die für Beobachter relevanten Daten erlaubt: Nutzen die Beobachter nur noch Referenzen auf dieses Interface, so löst man die Kopplung***. Abbildung 18-31 zeigt eine mögliche Umsetzung.

Abbildung 18-31 BEOBACHTER-*Muster mit* `IDataProvider`-*Interface*

Beobachtung mehrerer Subjekte Es ist möglich, dass sich ein Beobachter bei mehreren Subjekten als Interessent registriert. Zur Unterscheidung der Quelle der Änderungen ist es daher wichtig, dass eine Referenz auf den Urheber der Änderung in der `update()`-Methode mitgesendet wird. Arbeitet man jedoch nur mit einer simplen `update()`-Methode ohne einen solchen Quellenparameter, so weiß der Beobachter nicht, bei welchem Subjekt er den aktuellen Zustand nachfragen soll.

Warum ist Java-Built-In-Observer ungünstig?

Die Entwickler von Sun haben in das JDK mehrere BEOBACHTER-Muster integriert. Die Realisierungen durch die Klassen `ActionListener` und `EventListener` sind gelungen. Leider kann man das nicht für die Realisierung in Form der Klasse `Observable` und des Interface `Observer` sagen. Diese Realisierung verstößt gegen verschiedene OO-Gedanken. Beobachtbar zu sein ist eine Rolle und keine Erweiterung einer Basisklasse: Merkwürdigerweise erfordert die JDK-Realisierung aber hier eine technisch bedingte Ableitung von der Basisklasse `Observable` (Implementierungsvererbung). ***Interessanterweise widerspricht zudem die gewählte Namensgebung allen Konventionen im JDK:*** Die Endung `-able` wird ansonsten immer für Namen von Interfaces verwendet, hier jedoch zur Benennung einer Klasse. Diese etwas krude Klassenhierarchie ist in Abbildung 18-32 dargestellt.

Abbildung 18-32 *Nutzung der Observer-Funktionalität aus dem JDK*

Erschwerend kommt hinzu, dass durch den Zwang, von einer konkreten Klasse abzuleiten, eine Nutzung dieser vorgefertigten Realisierung in einer eigenen Klassenhierarchie schwierig wird: Besitzt eine eigene Klasse bereits eine Basisklasse, so kann sie nicht mit dem Java-Built-In-Observer genutzt werden.[8] Daher muss man in der Regel die gesamte Funktionalität selbst realisieren.

Weiterhin ist die Methode `update()` zu unspezifisch. Erstens besitzt ihr Name wenig Aussagekraft. Zweitens ist unklar, welche Teile des geänderten Zustands als Übergabeparameter verwendet werden sollten. Für eine solche generische Lösung ist damit der Datentyp der mitzuteilenden Daten unbekannt. Daher muss in diesem Fall ein Parameter vom Typ `Object` übergeben werden. Um sinnvoll mit den Daten arbeiten zu können, muss jeder Beobachter daraus die benötigten Informationen zurückgewinnen. Dies erfordert eine explizite Typprüfung per `instanceof`, um den ansonsten nicht typsicheren Cast auf den erwarteten Typ der Zustandsinformation abzusichern.

Anhand dieser kurzen Diskussion erkennt man leicht, dass eine universelle, generische `update()`-Methode, wie sie im Built-In-Observer von Java realisiert ist, wenig Sinn macht. Aufgrund der geschilderten Nachteile sind die Klasse `Observable` und das Interface `Observer` seit JDK 9 als deprecated markiert.

[8] Eine derartige Umsetzung des BEOBACHTER-Musters wäre nur dann sinnvoll gewesen, wenn Java Mehrfachvererbung unterstützen würde. Aber selbst in diesem Fall wäre die Einbindung über Vererbung kein guter Stil.

Verbesserungen

Statt einer allgemeinen `update()`-Methode ist es oft sinnvoller, mehrere Methoden anzubieten, mit denen jeweils spezifische, mögliche Zustandsänderungen propagiert werden können. Neben sprechenderen Methodennamen lassen sich Änderungen wesentlich besser nachvollziehen als bei einer `update()`-Methode. Beispielhaft verdeutlicht dies das folgende Interface `IModelListener`:

```
public interface IModelListener
{
   public void imageElementsChanged(final List<AbstractGraphicsElement> images);
   public void pdfElementsChanged(final List<AbstractGraphicsElement> pdfs);

   public void nameChanged(final String newName);
}
```

Als Erweiterung kann man statt eines allgemeinen Beobachters mehrere spezialisierte Beobachter für ausgewählte Teilaspekte des Objektzustands definieren. Eine Änderung der Selektion könnte etwa durch einen speziellen `ISelectionListener` und dessen `selectionChanged()`-Methode behandelt werden:

```
public interface ISelectionListener
{
   void selectionChanged(final List newSelection);
}
```

Abbildung 18-33 zeigt ein mögliches Klassendiagramm.

Abbildung 18-33 *Beispiel für mehrere Beobachter*

Für die beiden dargestellten Beobachter werden aufgrund der Push-Variante keine Referenzen und Zugriffe auf das Datenmodell `ContentModel` benötigt. Durch die Trennung der Interfaces für verschiedene Beobachter ist es sogar möglich, spezifische Informationen in den Benachrichtigungsmethoden zu kommunizieren.

Bewertung

Der Einsatz des BEOBACHTER-Musters hat folgende Auswirkungen:

+ **Lose Kopplung** – Im Idealfall kennen sich Subjekt und registrierte Beobachter nicht direkt, wenn eine Kapselung über Interfaces erfolgt. Durch die Trennung von Subjekt und Beobachtern sind Erweiterungen in beiden Klassen problemlos möglich, solange die Schnittstelle unverändert bleibt.
+ **Flexibilität** – Die Anzahl der Interessenten kann zur Laufzeit verändert werden. Hinzukommende oder später wieder entfernte Beobachter beeinflussen im Idealfall andere Beobachter nicht.
o **Fehlende Reaktion** – Falls einen Beobachter gewisse Meldungen nicht interessieren, kann er diese gegebenenfalls ignorieren: Es bleibt dem Beobachter überlassen, ob und wie er mit Nachrichten umgeht. Im Extremfall kann ein Beobachter die Änderungsmitteilungen einfach nicht beachten.
o **Probleme der Statusänderung** – Das Subjekt weiß nicht, welche Aktionen die Beobachter bei einer Änderungsmitteilung ausführen. Komplexe Aktionen können die Abarbeitung der Benachrichtigung verzögern oder sogar blockieren.
- **Komplexität** – Es stellt eine große Herausforderung dar, den Benachrichtigungsvorgang konsistent und Thread-sicher durchzuführen. In der Regel muss man mit gewissen Kompromissen leben, etwa dem, dass Beobachter, die sich während einer Benachrichtigung neu anmelden, erst bei zukünftigen Änderungen informiert werden.
- **Nachvollziehbarkeit** – Beim Einsatz dieses Musters werden oft ziemlich viele, mitunter auch zu feingranulare Benachrichtigungen erzeugt, was die Übersicht erschwert. Die Ursachen solch unerwünschter Updates sind dadurch schwierig zu finden.
- **Gefahr von Zyklen** – Werden Beobachter wieder von anderen Klassen beobachtet, so kommt man schnell in ein Benachrichtigungschaos oder endet sogar in einem Zyklus.

Tipp: Einfluss von Business-Methoden auf das BEOBACHTER-Muster

Das Einführen von Business-Methoden hilft, ein mögliches Benachrichtigungschaos beim Einsatz des BEOBACHTER-Musters zu vermeiden. Wenn lediglich Business-Methoden Zustandsänderungen propagieren, sorgt dies für mehr Klarheit und Transparenz.

Achtung: BEOBACHTER als Anti-Pattern

Das BEOBACHTER-Muster kann schnell zum Anti-Pattern werden, wenn Beobachter ihrerseits selbst ein beobachtetes Subjekt sind. Sind Abhängigkeitsbeziehungen nicht klar definiert, kommt es zu schwer nachvollziehbaren Updates. Eine scheinbar harmlose Zustandsänderung auf einem Subjekt kann eine Kaskade von Updates an Beobachter und deren abhängiger Beobachter zur Folge haben. In einem solchen System können kleinste Änderungen dazu führen, dass das System nicht mehr konvergiert. Nur minimale Variationen der Anfangsparameter führen möglicherweise zu einem komplett anderen Systemverhalten.

Es kann im Extremfall schnell eine Lawine von nicht mehr beherrschbaren Ereignissen ausgelöst werden, die im schlimmsten Fall zu Endlosschleifen führen, zumindest jedoch eine häufige Ursache für eine schlechte Performance sind.

18.3.8 MVC-Architektur

Die sogenannte Model-View-Controller-Architektur teilt ein zu modellierendes System in die drei Bestandteile Daten (Model), Darstellung (View) und Business- bzw. Kontrolllogik (Controller). Jeder Teil wird möglichst unabhängig von den anderen realisiert und oftmals über Schnittstellen gekapselt.

Die Idee, Daten und ihre Repräsentation zu trennen, ist insofern sinnvoll, als dass sich mehrere Darstellungsformen anbieten lassen. Weiterhin erreicht man durch diese Trennung eine klarere Struktur. Die Trennung bedingt allerdings, dass Änderungen im Modell an die Views kommuniziert werden müssen. Abschnitt 18.3.7 beschreibt das BEOBACHTER-Muster, das dafür eingesetzt werden kann.

Abbildung 18-34 *Model-View-Controller*

Anmerkungen

Wenn man es auf sehr feingranularer Ebene betrachtet, verstößt die MVC-Architektur gegen die Idee der Objektorientierung, die besagt, dass ein Objekt all seine Aspekte selbst behandeln sollte. Eine interessante Darstellung dazu finden Sie im Buch »Holub on Patterns« von Allen Holub [31]. Auf der anderen Seite kann man es auch als Fortführung des Gedankens der Kapselung sehen. Ein Objekt wird dabei semantisch in mehrere Objekte aufgespalten. Jeweils eins ist für den Zustand (Model) und das Verhalten (View und Controller) zuständig.

Bewertung

Der Einsatz der MVC-Architektur besitzt folgende Auswirkungen:

+ **Lose Kopplung** – Kommunizieren die einzelnen Komponenten ausschließlich über Interfaces, so sind alle Komponenten nur lose miteinander verbunden.
+ **Trennung von Zuständigkeiten** – Jede Komponente repräsentiert einen speziellen Aspekt des Gesamtsystems. Es findet eine Trennung von Zuständigkeiten statt.
+ **Flexibilität** – Mehrere Darstellungen oder Ansichten (z. B. Balkendiagramm und Tortendiagramm) auf dieselben Daten lassen sich leicht realisieren. Die einzelnen Komponenten können als eine Klasse oder als mehrere Klassen oder sogar in Form eines eigenständigen Programms realisiert sein.
+ **Konsistente Darstellung** – Durch die zentrale Datenhaltung in einem Modell und die Änderungsbenachrichtigungen mithilfe des BEOBACHTER-Musters können alle Darstellungen (Views) konsistente Daten anzeigen (sofern die Views korrekt auf Änderungsmitteilungen reagieren).
o **Overengineering** – In einer Auslegung der Objektorientierung sollte ein Objekt alle seine Belange regeln. Nicht immer ist eine Aufteilung wie bei der Model-View-Controller-Architektur sinnvoll und notwendig. Wie bei allen Techniken sollte man auch hier darauf achten, dass man nicht mit Kanonen auf Spatzen schießt.
- **Erhöhte Komplexität** – Die Anzahl der benötigten Klassen führt zu etwas höherer Komplexität.
- **Gefahr vieler Änderungsmitteilungen** – Werden viele feingranulare Änderungen im Modell vorgenommen und sofort an die Ansichten kommuniziert, so kommt es zu diversen Aktualisierungen der Ansichten. Das kann unerwünscht sein und sich negativ auf die Performance auswirken.
- **Gefahr enger Kopplung** – Werden die Komponenten nicht durch Schnittstellen voneinander abgekoppelt, so führt man ungewollt direkte Abhängigkeiten ein und die Wiederverwendbarkeit der Einzelkomponenten sinkt.

18.4 Weiterführende Literatur

Zum Thema Entwurfsmuster ist unzählige Literatur erschienen. In der folgenden Aufzählung empfehle ich einige Bücher, die jeweils unterschiedliche Aspekte und Herangehensweisen bei der Vorstellung von Entwurfsmustern verfolgen.

- »**Design Patterns – Elements of Reusable Object Oriented Software**« bzw. »**Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software**« von der sogenannten Gang of Four (GoF): Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides [22] bzw. [23]
  Das Standardwerk der GoF habe ich 1998 kennen- und viele der Muster schätzen gelernt. Manche Beschreibungen sind etwas trocken und formal. Die dahinter steckenden Ideen sind jedoch beim Entwurf guter Software sehr hilfreich.
- »**Design Patterns Explained**« von Alan Shalloway und James R. Trott [66]
  Dieses exzellente Buch erklärt die von der GoF beschriebenen Entwurfsmuster auf eine verständlichere Weise anhand von Praxisbeispielen. Es wird zudem ein sehr guter, fundierter Einstieg in den OO-Entwurf gegeben.
- »**Head First Design Patterns**« von Eric Freeman, Elisabeth Freeman, Kathy Sierra und Bert Bates [21]
  Dieses unterhaltsame Buch beschreibt Entwurfsmuster weniger formal und ermöglicht einen guten und abwechslungsreichen Einstieg. Die Muster werden sehr verständlich und anschaulich in Form von Text und Bildern dargestellt.
- »**Holub on Patterns**« von Allan Holub [31]
  Dieses Buch betrachtet Entwurfsmuster aus dem Blickwinkel der Praxis. Anhand zweier Beispielapplikationen werden alle GoF-Muster vorgestellt und verdeutlicht. Dabei wird großer Wert auf sauberes objektorientiertes Design gelegt.
- »**Design Patterns Java Workbook**« von Steven J. Metsker [52]
  Dieses Buch beschreibt alle von der GoF vorgestellten Muster und ist so konzipiert, dass jedes Muster durch Übungsaufgaben erlernt und das jeweilige Wissen vertieft werden kann.

VI Qualitätssicherungs-maßnahmen

19 Programmierstil und Coding Conventions

In meiner Tätigkeit als Softwareentwickler hat es sich immer wieder als Vorteil erwiesen, gewisse Konventionen beim Programmieren einzuhalten. In diesem Kapitel beginne ich mit allgemeinen Hinweisen zum Programmierstil in Abschnitt 19.1. Auch das Layout des Sourcecodes kann großen Einfluss auf dessen Verständlichkeit haben. Das wird in Abschnitt 19.2 diskutiert. Im Anschluss daran stellt der nach Themen gegliederte Katalog der Codierungsregeln (Coding Conventions) in Abschnitt 19.3 Leitsätze, Regeln und Tipps vor, die dabei helfen, Fehler provozierenden Sourcecode zu vermeiden und übersichtlichere und verständlichere Programme zu schreiben. Man könnte diese Techniken auch die »Good Smells« nennen – im Gegensatz zu den »Bad Smells« aus Kapitel 16. Werden die »Good Smells« zur Gewohnheit, so vermeidet dies Probleme und hilft dabei, Fehler leichter zu finden, ohne dafür viel Mühe zu investieren. Man erspart sich dadurch einigen Aufwand, der in unübersichtlichen Programmen betrieben werden muss, um Softwaredefekte im Nachhinein aufzuspüren und zu beheben.

Abschnitt 19.4 geht auf das Thema Sourcecode-Prüfung ein und stellt Bewertungskriterien, sogenannte Metriken, zur Beurteilung der Güte von Sourcecode vor. Anschließend werden verschiedene Tools betrachtet, um die Einhaltung der vorgestellten Programmierrichtlinien zu überprüfen.

19.1 Grundregeln eines guten Programmierstils

Wenn sich die Anforderungen an Programme über die Zeit wandeln, ist es für eine Wartung von großem Vorteil, wenn der Sourcecode übersichtlich und gut lesbar ist. Dies erleichtert das Verständnis. Für den Computer ist das Kriterium der Lesbarkeit allerdings vollkommen unbedeutend, solange das Programm syntaktisch korrekt ist. Diese syntaktische Korrektheit sagt aber noch gar nichts aus. Selbst wenn das Programm auch noch semantisch korrekt ist, kann es immer noch fürchterlich strukturiert sein und damit jegliche Wartung zur Qual werden lassen. Martin Fowler schreibt dazu Folgendes: »Any fool can write code that a computer can understand. Good programmers write code that humans can understand.« Übersetzt: ***»Jeder Dummkopf kann Code schreiben, der für Computer verständlich ist. Gute Programmierer schreiben Code, den Menschen verstehen können.«*** [20].

19.1.1 Keep It Human-Readable

Guter Sourcecode liest sich (zumindest auf der Ebene der öffentlichen Methoden) fast wie ein Roman. Es ist ein klarer Handlungsstrang (Kontrollfluss) erkennbar und jeder Akteur (Klasse, Objekt oder Methode) macht nur das, was seiner Aufgabe entspricht. Die gute Lesbarkeit hat verschiedene Aspekte. Als Basis dienen aussagekräftige und sprechende Namen für Klassen, Methoden und Variablen. Gezielt eingesetzte erklärende Kommentare erleichtern das Verständnis und geben Kontextinformationen. Wichtig ist aber auch das Layout des Sourcecodes. Sind Einrückungen und Klammerungen nicht zu erkennen, so erschwert dies die Lesbarkeit ungemein.

19.1.2 Keep It Simple And Short (KISS)

Halten Sie Ihr Design und den Sourcecode einfach. Denken Sie an Albert Einsteins Spruch: »Everything should be made as simple as possible, but no simpler.« Frei übersetzt: »Alles sollte so einfach wie möglich gemacht werden, aber nicht einfacher.« Man spricht auch vom KISS-Prinzip (Keep It Simple And Short). Versuchen Sie also nicht, andere Entwickler durch komplizierte Konstrukte zu beeindrucken – denn erfahrungsgemäß sind gerade diese scheinbar besonders cleveren Lösungen oftmals schlecht lesbar, wenig verständlich und führen deswegen später häufig zu einem Wartungsalbtraum.

Je weniger Komplexität in Methoden und Klassen steckt, desto übersichtlicher, test- und wartbarer werden diese. Im besten Fall lassen sie sich in anderen Zusammenhängen wiederverwenden. Wird jedoch zu viel Funktionalität in eine Klasse oder Methode eingebracht, so stört dies die ***Orthogonalität***. Methoden und Klassen lassen sich dann nicht mehr so gut zu neuen Einheiten kombinieren. Dies adressiert auch das Single Responsibility Principle (SRP), das besagt, dass Klassen und Methoden nur eine Zuständigkeit besitzen und somit auch möglichst einfach und kurz sein sollten (vgl. Abschnitt 3.5.3).

19.1.3 Keep It Natural

Bevor Sie mit dem Entwurf Ihrer Software beginnen, sollten Sie die Anforderungen daran recht gut verstanden haben, da ansonsten die Gefahr groß ist, etwas zu entwerfen, was die Bedürfnisse Ihrer Kunden nicht (korrekt) erfüllt. Am besten schreiben Sie wichtige Dinge nieder, weil sich dadurch Ihr Verständnis für das zu lösende Problem verbessert. Das hilft dabei, ein besseres Bild zu gewinnen und eine angemessene und natürliche Lösung zu finden.

Denken Sie beim Design in klaren Strukturen und Zuständigkeiten und diskutieren Sie Ihre Entwürfe mit anderen, wobei UML-Diagramme hilfreich sein können. Verwenden Sie zur Strukturierung immer dann Entwurfsmuster, wenn es von Nutzen ist. Verfallen Sie aber nicht jedem neuen Hype und versuchen Sie nicht, etwas krampfhaft einsetzen zu wollen: Als Mitte der 90er-Jahre Entwurfsmuster bekannt wurden, habe ich Kollegen erlebt, die diese ohne Hinterfragen selbst für einfache Problemstellungen genutzt haben. Ähnliches gilt für Optimierungen: Befassen Sie sich nicht zu früh damit, denn diese führen teilweise zu merkwürdigen Umsetzungen.

19.1.4 Keep It Clean

Halten Sie unbedingt immer Ordnung in Ihrem Sourcecode. Es gilt das Gesetz der Entropie: Wo etwas Unordnung herrscht, kommt (automatisch) neue hinzu. Man kennt dies auch als »Broken-Windows-Theorie«.

```
public void updateXMLState(final InputStream inStream) throws IfsException,
                                                              IfsException
```

Sie halten dies für ein konstruiertes Beispiel? Nein, weit gefehlt. Das ist tatsächlich Sourcecode aus einem realen Programm. Man fragt sich, wie das passieren konnte. Nachträglich kann man sich das schwer erklären. Durch ein wenig Disziplin, Ordnung und Achtsamkeit können solche Unschönheiten meist schnell aufgedeckt werden.

Der Ratschlag »Keep It Clean« folgt dem DRY-Prinzip (»Don't Repeat Yourself«), das besagt, dass man Duplikation in Sourcecode möglichst vermeiden sollte (vgl. [34]).

19.2 Die Psychologie beim Sourcecode-Layout

Jeder kennt es: Es müssen Änderungen im Sourcecode durchgeführt werden, aber bereits das Verständnis des Sourcecodes bereitet Probleme, da auf den ersten Blick keine klare Struktur erkennbar ist. Vielmehr sieht das Programm nach einer wild zusammengewürfelten Methodensammlung aus. In der Wahrnehmungspsychologie gibt es für das Erkennen der Zusammengehörigkeit von Elementen verschiedene Gestaltgesetze. Zwei für die Sourcecode-Formatierung wichtige stelle ich im Folgenden vor.

19.2.1 Gesetz der Ähnlichkeit

Das ***Gesetz der Ähnlichkeit*** beschreibt, wie wir Dinge miteinander in Verbindung bringen, wenn sich diese ähneln, z. B. in ihrer Farbe oder Form. Verschiedene Merkmale führen zu unterschiedlicher Gruppierung (vgl. Abbildung 19-1).

Abbildung 19-1 *Gesetz der Ähnlichkeit*

Auf Sourcecode-Ebene übertragen, kann man den Grad an Ähnlichkeit anhand von Abstraktionsebenen ausdrücken. Es bietet sich an, etwa folgende drei zu unterscheiden:

1. Auf *hoher* Abstraktionsebene werden lediglich Business-Methoden aufgerufen.
2. Auf *mittlerer* Abstraktionsebene finden normale Methodenaufrufe statt.
3. Auf *niedriger* Abstraktionsebene werden einzelne Anweisungen kombiniert.

Was bedeutet das für die Praxis, wenn man in einer Methode verschiedene Abstraktionsgrade findet? Zunächst scheint das nur kosmetischer Natur zu sein – es steht aber ein größeres Problem dahinter: Die unterschiedlichen Abstraktionsebenen erschweren das Verständnis sowie das Erkennen von Ähnlichkeiten und Zusammenhängen.

Beispiel

Schauen wir uns die Problematik konkret am Beispiel der folgenden Methode `paint(Graphics)` an:

```
public void paint(final Graphics graphics)
{
    if (showGrid)
    {
        graphics.setColor(Color.DARK_GRAY);

        // Raster zeichnen
        for (int x = 0; x < getSize().width; x += GRID_SIZE_X)
        {
            for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)
            {
                graphics.drawLine(x, y, x, y);
            }
        }
    }

    paintFigures(graphics);
}
```

Diese Methode ist schon recht kurz und übersichtlich. Man kann das Ganze noch verbessern, indem man (private) Hilfsmethoden – in diesem Fall eine Methode zum Zeichnen des Rasters `paintGrid(Graphics)` – einführt. Damit erreicht man eine bessere Strukturierung des Sourcecodes. Außerdem kann die Implementierung der jeweiligen Methode nun auf einheitlichen Abstraktionsebenen erfolgen, wie nachfolgend gezeigt:

```
public void paint(final Graphics graphics)
{
    if (showGrid)
    {
        paintGrid(graphics);
    }

    paintFigures(graphics);
}

private void paintGrid(final Graphics graphics)
{
    graphics.setColor(Color.DARK_GRAY);

    for (int x = 0; x < getSize().width; x += GRID_SIZE_X)
    {
        for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)
        {
            graphics.drawLine(x, y, x, y);
        }
    }
}
```

Die gezeigte Strukturierung erleichtert das Nachvollziehen. Für öffentliche Methoden (`paint(Graphics)`) kann man in Konzepten denken. Für private Methoden (`paint-Grid(Graphics)`) bleibt man immer auf der Implementierungsebene. Der Vorteil einer einheitlichen Abstraktionsebene innerhalb der Implementierung einer Methode ist, dass nicht ständig gedanklich zwischen den Ebenen gewechselt werden muss. Das kommt dem Lesefluss und der Verständlichkeit zugute.

19.2.2 Gesetz der Nähe

Das ***Gesetz der Nähe*** beschreibt, dass eng benachbarte Elemente als Teil eines größeren Ganzen wahrgenommen werden, wie dies Abbildung 19-2 zeigt, wo die acht Linien zu vier Linienpaaren kombiniert werden. Dagegen kann sich eine schlechte Sourcecode-Formatierung negativ auf das Verständnis auswirken.

Abbildung 19-2 *Gesetz der Nähe*

Beispiel

Wenn wir semantisch zusammengehörende Elemente gruppieren, so wird der Sourcecode besser lesbar und dessen logische Struktur ist leichter zu erkennen. Als Negativbeispiel dient ein über mehrere Zeilen verstreuter, wirklich so in Produktivcode gefundener Kommentar, der gemäß dem Gesetz der Nähe nur mit viel Mühe als ein zusammenhängender Kommentar erkennbar ist:

```
private boolean showContent = false; // Don't

// show
// content
// first

private final Object lockObject = new Object();
```

Die verschiedenen Kommentarzeilen zum Attribut `showContent` sind unübersichtlich und werden kaum als miteinander verbunden erkannt. Das Gesetz der Nähe lässt uns darin spontan zwei Kommentare sehen und Teile davon fälschlicherweise dem Attribut `lockObject` zuordnen. Eine Verbesserung erreicht man durch den Einsatz eines einzeiligen Kommentars oder bevorzugt durch eine Javadoc-Kommentierung. Dadurch wird eine klare visuelle Trennung erzielt und auch eine Zuordnung der Kommentare zum Attribut `lockObject` vermieden. Folgendes Listing zeigt dies:

```
/** Don't show content first */
private boolean showContent = false;  // Don't show content first

private final Object lockObject = new Object();
```

19.3 Coding Conventions

Hinweise zur Gestaltung des Sourcecodes werden durch ***Coding Conventions*** beschrieben. Als Ausgangsbasis für ein eigenes Regelwerk können die Regeln[1] von Scott Ambler [1] dienen. In diesem Abschnitt werden diese Basisregeln um weitere »Best Practices« ergänzt, die dabei helfen sollen, gewisse bekannte Fallstricke zu vermeiden.

Vor allem im Team profitiert man von »gelebten« Coding Conventions: Ein nahezu einheitlicher Programmierstil ermöglicht, dass sich jeder Entwickler sofort auch in »fremdem« Sourcecode zurechtfindet, da alles ähnlich und vertraut ist. Persönliche Vorlieben und Eigenarten werden auf ein Minimum reduziert, wodurch auch die Durchführung von Codereviews und Pair Programming erleichtert wird.

Definition: Codereviews und Pair Programming

Unter ***Codereviews*** versteht man Meetings mehrerer Entwickler (in der Regel der Sourcecode-Produzent sowie ein oder mehrere Begutachter), die einige Sourcecode-Abschnitte kritisch betrachten und auf potenzielle Schwachstellen oder Fehler analysieren. Weitere Informationen liefert Kapitel 21.

Mit ***Pair Programming*** bezeichnet man das Schreiben von Programmen durch zwei Entwickler, die gemeinsam vor einem Rechner sitzen. Der eine ist aktiv und programmiert, der andere hat die Rolle des aktiven Beobachters und Reviewers. Er denkt mit, stellt Fragen und gibt Kommentare ab oder bringt Verbesserungsvorschläge ein, wodurch sich sukzessive ein gemeinsames Verständnis entwickelt. Nach einer gewissen Zeit wechselt dann die jeweilige Rolle.

Eine große Akzeptanz von Coding Conventions erreicht man, indem man vor deren Einführung möglichst alle Entwickler an Entwürfen des neuen Regelwerks mitwirken lässt. Nichtsdestotrotz erfordert die Einhaltung der Regeln anfangs immer etwas Disziplin. Es sollte zudem eine dynamische Entwicklung des Regelwerks vollzogen werden, um dessen Akzeptanz weiter zu erhöhen. Bisher nicht adressierte Programmierprobleme können als Regel neu aufgenommen werden. Eher hinderliche oder überkritische Regeln können nach gründlicher Prüfung und sinnvoller Begründung – nicht nur aufgrund des persönlichen Missfallens einzelner Entwickler – aus dem Regelwerk entfernt werden. Bevor man allerdings vorschnell eine Regel entfernt, kann man besser in einigen Ausnahmesituationen gegen einzelne Regeln verstoßen: ***Nicht jede Regel ist gleichermaßen gut auf jede Situation anzuwenden, sodass es in Einzelfällen durchaus sinnvoll ist, gegen diese zu verstoßen. Allerdings sollte dies gut begründet und dokumentiert werden***.

[1] `http://www.ambysoft.com/essays/javaCodingStandards.html`

Meiner Erfahrung nach ist es sehr hilfreich, die Coding Conventions durch Tools unterstützt zu überprüfen. Falls die Einhaltung der Regeln nur über Codereviews geprüft wird, so sind meistens diejenigen die »Buhmänner« und Kritiker, die das Regelwerk am besten kennen. Weiterhin kann es zum »Schwarzen Peter«-Syndrom kommen: Der Reviewte fühlt sich persönlich angegriffen und die Reviewer mögen kaum noch weitere Kritik äußern. Als Folge werden dann aber nicht mehr alle Softwaredefekte und Regelverstöße angemerkt. Somit werden Sinn und Nutzen des durchgeführten Codereviews fraglich. Diesem Dilemma kann man durch den Einsatz von Tools zur Sourcecode-Prüfung entgegenwirken. Diese übernehmen die Rolle des Kritikers, jedoch ohne emotionale und persönliche Probleme zu verursachen. Diese sozialen Aspekte sollte man im Hinterkopf behalten, wenn man Coding Conventions einführen und ergänzend dazu Codereviews durchführen möchte. Das Buch »Soft Skills für Softwareentwickler« von Uwe Vigenschow und Björn Schneider [72] enthält weitere Informationen zu psychologischen Aspekten der Arbeit in Teams.

19.3.1 Grundlegende Namens- und Formatierungsregeln

Es ist sinnvoll, die bereits erwähnten Coding Conventions bezüglich Formatierung und Namensgebung als Basis für eigene Regeln zu verwenden. Als wesentliche Punkte für eine gute Übersicht und Lesbarkeit möchte ich folgende zwei Punkte besonders hervorheben, die in diesem Buch genutzt werden:

1. Klammern und Anweisungen sollten in jeweils eigenen Zeilen stehen, pro Zeile also möglichst nur eine Anweisung bzw. eine Variablendeklaration.
2. Zeilen sollten eine gewisse Länge nicht überschreiten. Ein Wert zwischen 100 und 150 Zeichen hat sich als praktikabel erwiesen.

Das folgende Beispiel zeigt sowohl die Anwendung der Namenskonventionen als auch der Formatierungsregeln:

```
public final class FormatingExample
{
    private static final Logger log = Logger.getLogger("FormatingExample");

    public static String asHex(final byte[] tele)
    {
        log.info("asHex(" + Arrays.toString(tele) + ")");

        final StringBuffer sb = new StringBuffer("0x");

        for (int i = 0; i < tele.length; i++)
        {
            final String hex = Integer.toHexString(tele[i]);
            sb.append(hex);
        }

        return sb.toString();
    }
    // ...
}
```

Ausnahmen für die Formatierung

Nur in wenigen Fällen sind Abweichungen von der Standardformatierung sinnvoll. Dies gilt insbesondere für die Forderung »ein Statement pro Zeile«. Gegen diese Regel kann man beispielsweise dann verstoßen, wenn nur eine Menge einfacher `get()`-Methoden angeboten wird. Wenn die Methodenrümpfe – wie nachfolgend gezeigt – hinter die Methodennamen geschrieben werden, lässt sich die Lesbarkeit erhöhen.

```
public final String getName()          { return name; }
public final String getCity()          { return city; }
public final int    getAge()           { return age; }
```

Wie man leicht sieht, ist diese Schreibweise kurz und elegant und fokussiert die Aufmerksamkeit auf die Schnittstelle und nicht die Implementierung, die sowieso trivial ist. Die Lesbarkeit des Sourcecodes wird positiv beeinflusst.

Namensregeln

Eine konsistente Namensgebung von Variablen, Methoden und Klassen hilft, auf einen Blick diese Elemente erkennen und voneinander unterscheiden zu können. Für Namen gilt in der Regel die sogenannte ***CamelCase-Schreibweise***, d. h., jedes neue Wort wird wieder mit einem Großbuchstaben begonnen. Tabelle 19-1 liefert Anregungen.

Tabelle 19-1 *Namenskonventionen*

Typ	Präfix	Postfix	Beispiel
Packages	-	-	`mypackage.subpackage`
Interfaces	`I` / -	`IF` / -	`IFileInfo`, `Filterable`
Abstrakte Klassen	`Base`, `Abstract`	-	`AbstractProcessStep`
Klassen	-	-	`JobAssistant`
Testklassen	-	`Test`	`ProcessStepTest`
Methoden	-	-	`getFileName()`
Konstanten	-	-	`MAX_VALUE`
Lokale Variablen	-	-	`imageWidth`
Membervariablen	-, `m_`, `this.`	-	`name`, `m_name`, `this.name`

Über den Einsatz für Interfaces und Membervariablen gibt es kontroverse Meinungen mit guten Argumenten sowohl dafür als auch dagegen – ansatzweise wird dies im folgenden Hinweiskasten beleuchtet.

Testklassen sollten mit dem Postfix `Test` enden und nicht damit starten, weil so die Testklasse leichter zu einer bestehenden Klasse gefunden werden kann. Wie bereits in Abschnitt 2.4.1 angedeutet, würden durch Einsatz von `Test` als Präfix alle Testklassen damit beginnen und man fände den »Wald vor lauter Bäumen nicht«.

Hinweis: Stilfragen

Position der geschweiften Klammern Die Position der öffnenden, geschweiften Klammer löst häufig kontroverse Diskussionen aus. Mir helfen Klammern in einer eigenen Zeile, die Struktur besser zu erkennen.

Präfixe für Attribute Die Diskussion um Präfixe für Attribute ist wahrscheinlich genauso endlos zu führen wie die um die richtige Position der öffnenden Klammer. Allerdings können Präfixe manchmal helfen, Fehler zu vermeiden und die Übersicht zu erhöhen. Dies gilt vor allem bei der Verwendung von Übergabeparametern, die wie Attribute heißen. Folgende Methode `setCounter(int)` zeigt diesen Fehler:

```
public void setCounter(int count)
{
    count = count;   // Achtung: Sinnlose Selbstzuweisung!
}
```

Hier findet nicht die gewünschte Änderung im Objektzustand statt, da die Variable an sich selbst zugewiesen wird. Als Abhilfe existieren folgende Möglichkeiten:

1. Aktivierung entsprechender Style-Checks in der IDE, sodass solche Zuweisungen als fehlerhaft angemerkt werden.
2. Den Übergabeparameter `final` deklarieren, wodurch ein Kompilierfehler bei einer solchen versehentlichen Zuweisung erzeugt wird.
3. Die Angabe von `this.` verhindert die Zuweisung und man erreicht eine bessere visuelle Trennung von Attributen, lokalen Variablen und Parametern.

Alle Abhilfen lassen sich kombinieren. Nachfolgend sind Abhilfen 2 und 3 eingesetzt:

```
public void setCounter(final int count)
{
    this.count = count;
}
```

Typpräfixe für Attribute Das Präfix `m_` für Attribute führt zu einer starken visuellen Trennung – geht aber mit einer etwas schlechteren Lesbarkeit einher. Manchmal findet man zusätzliche Präfixe, etwa `str` für Strings oder `is` für boolesche Variablen. Auch hier scheiden sich wieder die Geister. *Auf jeden Fall sollte dies mit Bedacht genutzt werden.* In der Regel sollte nicht der Typ einer Variablen im Namen stecken, sondern bevorzugt ihre semantische Bedeutung.

Kennzeichnung von Interfaces Auch über die Verwendung von Präfixen bzw. Postfixen für Interfaces kann man diskutieren. Zum Teil mag ich diese explizite Kennzeichnung, aber oftmals lassen sich passendere Namen finden.

Teilweiser Verstoß gegen die Oracle-Regeln Meine Empfehlungen widersprechen zwar einigen Regeln von Oracle. Ein Blick in den Sourcecode des JDKs zeigt aber, dass die dort genutzten recht kurzen Zeilenlängen und die Formatierung nicht immer für gute Lesbarkeit sorgen. Schlussendlich ist die Sourcecode-Formatierung aber ein kontrovers diskutiertes Thema.

19.3.2 Namensgebung

Namensregeln lassen sich nur bedingt durch Tools prüfen. Zwar können diese gerade noch eine regelkonforme Schreibweise testen, aber für ein Tool ist der Name `map` oder `list` genauso gut oder schlecht wie die verständlichen Namen `nameToPersonMap` oder `deviceList`. Daher ist es sinnvoll, etwas Selbstdisziplin walten zu lassen. Auch Pair Programming und Codereviews können beim Finden von sinnvollen Namen helfen.

Vermeide Namenskürzel

Aussagekräftige Namen machen ein Programm besser lesbar. Abkürzungen und Namenskürzel können das Verständnis erschweren, wenn es sich nicht gerade um die gebräuchlichen Abkürzungen `sb` für `StringBuffer` oder `it` für Objekte vom Typ `Iterator` handelt. Im Extremfall verkümmert ein Variablenname zu einer kryptischen Zusammenstellung aus den Anfangsbuchstaben der Teilworte, etwa `nds` für `NotificationDeliveryService`.[2] Manchmal und vor allem für gängige Abkürzungen (HTML, DB, XML) kann auch dies vernünftig sein. Oft leidet aber die Verständlichkeit ungemein. Als »Leckerbissen« hat mir ein Kollege die Variable `AAA` zugespielt ... nein, es hat nichts mit Batterien zu tun! Dies ist eine misslungene Abkürzung für eine booleschen Variable mit der Bedeutung »**A**lle **A**nschlüsse **A**nzeigen«.

Schauen wir auf ein weiteres Negativbeispiel, das Kommentare nutzt, um den ansonsten unleserlichen Sourcecode zu beschreiben:

```
// enthält den Maximalwert
int val = -1;

// Durchlaufe alle vorhandenen Tabellenzeilen
for (int i = 0; i < 50; i++)
{
    val = Math.max(val, values[i].getValue());
}
```

Die beiden Kommentare sind trivial, kaum hilfreich und sogar ein wenig irreführend. Natürlich vermutet man, dass es sich bei `i` um die Zeile handelt und dass `val` den Maximalwert ermitteln soll, aber warum schreibt man es dann nicht gleich wie folgt?

```
final int PERSON_TABLE_ROW_COUNT = 50;

int maxAge = -1;
for (int rowIndex = 0; rowIndex < PERSON_TABLE_ROW_COUNT; rowIndex++)
{
    maxAge = Math.max(maxAge, persons[rowIndex].getAge());
}
```

Durch eine sinnvolle Benennung der Variablen, hier z. B. `maxAge` statt `val` und `persons` statt `values`, wird nicht nur der Algorithmus (Berechnung des Maximums), sondern auch der dahinter liegende Sinn (maximales Alter der aufgelisteten Personen

[2] Abweichend zu den vorherigen Auflagen empfinde ich mittlerweile `ioe` oder `sdf` für `IOException` bzw. `SimpleDateFormat` im `catch` in Ordnung.

ermitteln) offensichtlich. Gut gewählte Namen sorgen also nicht nur für Verständlichkeit, sondern erlauben es auch, auf eine »Pseudokommentierung« zu verzichten.

Vermeide Variablennamen, die nur den Typ wiederholen

Manchmal findet man Sourcecode, in dem Variablennamen aus deren Typen hergeleitet werden. Derartige Variablennamen sind oftmals wenig verständlich, da sie keinen Hinweis auf den Einsatzzweck enthalten. Betrachten wir folgendes Beispiel:

```
final Vector<File> vector = new Vector<>();

final List<File> list = new ArrayList<>();
final List<File> list1 = new ArrayList<>();
final List<File> list2 = new ArrayList<>();
```

Für einzelne temporäre Variablen kann es zum Teil schwierig sein, einen sinnvollen Namen zu finden – eine (reine) Nutzung des Typs ist dann durchaus tolerierbar. Es gibt aber Ausnahmen: Problematisch wird dies etwa, wenn mehrere Objekte gleichen Typs verwendet werden: Zur Unterscheidung muss man die Objekte irgendwie kennzeichnen. Meistens werden diese dann einfach durchnummeriert. Dabei stellt sich die Frage nach der Namenskonsistenz: Wird das erste Objekt mit einer »`0`« oder einer »`1`« gekennzeichnet oder erhält der Name, wie im obigen Beispiel, keinen Zusatz?

Verwende sinnvolle, konsistente Namen

Wie bereits angemerkt, helfen sinnvoll gewählte Namen, ein Programm nachvollziehbar zu machen, lassen sich aber leider nicht durch Tools prüfen. Für die Lesbarkeit und die Semantik ist man als Entwickler demnach selbst verantwortlich. Helfen Sie sich und Ihren Kollegen durch passend gewählte Namen.

Greifen wir das obige Beispiel wieder auf. Nehmen wir an, die drei Listen würden zur Modellierung von Veränderungen des Inhalts eines Verzeichnisses genutzt. Folgende Namensgebung macht diesen Sachverhalt intuitiv klar:

```
final List<File> newFiles = new ArrayList<>();
final List<File> changedFiles = new ArrayList<>();
final List<File> removedFiles = new ArrayList<>();
```

Eine Verwechselung von `list1` und `list2` aus dem vorherigen Beispiel ist leicht möglich. Bei der Namensgebung `changedFiles` und `removedFiles` ist eine Verwechselung nahezu ausgeschlossen.

Nicht immer ist ein Problem auf den ersten Blick zu erkennen, wenn man nicht direkt die Definition der Variablen und des zugehörigen Typs sieht:

```
final PersonService personDAO = ServiceFactory.getPersonService();
```

Betrachtet man den verständlichen Variablennamen `personDAO`, so könnte man meinen, ein Data Access Object (DAO) zu referenzieren. Erst ein zweiter, genauerer Blick auf die Definitionsstelle offenbart die Inkonsistenz zwischen Name und Inhalt (Refe-

renz auf einen Service). Wir erkennen daran, dass die konsequente Benennung von Variablen ein weiterer wichtiger Schritt in Richtung Nachvollziehbarkeit ist. Dazu gehört auch, einen Zähler namens `counter` nicht an anderer Stelle `cnt` zu nennen.

Benenne Klassen nach der bereitgestellten Funktionalität

Klassen sollten nach ihrer Aufgabe benannt werden und nicht nach dem Package, in dem sie gespeichert sind. Es gibt Projekte, in denen einige Entwickler dem Klassennamen den Package-Namen voranstellen. Das führt zu seltsamen Klassennamen:

```
JC_JOB
TEDI_Figure
PIDMessageHandler
```

Wird eine Klasse mit Package-Präfix in ein anderes Package verschoben, so müsste als Folge konsequenterweise der Klassenname angepasst werden – unterbleibt dies, so kommt es zu Inkonsistenzen. Wird eine Klasse von vielen weiteren Klassen referenziert, so müssen überall dort die Namensänderungen nachgezogen werden. Zum Glück geschieht dies meistens automatisch durch die IDE. Das führt wiederum zu vielen geänderten Dateien und erhöht die Gefahr für Konflikte beim Einchecken ins Repository.

Verwende für Containerklassen einen Pluralnamen

Nutzt man Pluralnamen für Collections oder Arrays, so erhöht dies die Lesbarkeit und erleichtert die Unterscheidung von normalen Referenzen:

```
// Einzelelemente
final GraphicObject graphicObject = new GraphicObject();
final String predefinedName = "Name";

// Collections
final List<GraphicObject> graphicObjects = new ArrayList<>();
final String[] predefinedNames = new String[] { "Name 1", ..., "Name n" };
```

Bei Containern kann der Typ der Containerklasse im Namen wiederholt werden und als Kontextinformation über die verwendete Datenstruktur (z. B. `List<E>` oder `Map<K,V>`) dienen. Dadurch verringert sich jedoch teilweise die Lesbarkeit:

```
final List<GraphicObject> listOfGraphicObjects = new ArrayList<>();
final List<GraphicObject> graphicObjectsList = new ArrayList<>();

final String[] predefinedNamesArray = new String[] { "Name 1", ..., "Name n" };
```

Hilfsvariablen können die Lesbarkeit erhöhen

Manchmal lässt sich durch die Definition einer Hilfsvariablen mit sprechendem Namen der Sourcecode besser verständlich gestalten. Betrachten wir folgendes Beispiel:

```
private final TreeSet<String> timeStampSet = new TreeSet<>(Collections.
                                                          reverseOrder());
```

Wenn wir diese Zeile anschauen, erkennen wir, dass eine umgekehrte Sortierreihenfolge realisiert wird. Das ist recht klar, aber was bedeutet das konkret? Versuchen wir den Sinn durch eine Hilfsvariable zu verdeutlichen und sehen wir uns an, wie gut der folgende Sourcecode dies kommuniziert:

```
private final Comparator<String> mostCurrentFirst = Collections.reverseOrder();
private final TreeSet<String> timeStampSet = new TreeSet<>(mostCurrentFirst);
```

Hinweis: Vorteil sprechender Namen

Achten Sie darauf, möglichst gut lesbaren Sourcecode zu schreiben. Der Grund ist einfach folgender: Sie werden Sourcecode z. B. im Rahmen von Erweiterungen oder Bugfixes viel häufiger lesen und verstehen müssen, als Dinge neu zu schreiben.

19.3.3 Dokumentation

Verwende kurze, aussagekräftige Kommentare

Ist ein Abschnitt des Sourcecodes trotz guter Formatierung und sinnvoller Namensgebung von Variablen und Methoden noch nicht ausreichend verständlich, so sollten kurze Kommentare eingefügt werden.

Vermeide Kommentare, die nur den Ablauf beschreiben

Manchmal wird die vorherige Regel falsch ausgelegt und es werden selbst *einfache* Programmstellen kommentiert: Man findet dann häufig Beschreibungen zum Ablauf bzw. zur Ausführung des Sourcecodes, nicht aber zu dessen Ziel. Das haben wir schon im Beispiel der Namensgebung gesehen und wird nun nochmals aufgegriffen:

```
// SCHLECHTER KOMMENTAR: 365 Tage durchlaufen
for (int tag = 0; tag < 365; tag++)
{
    gesamtUmsatz += tagesUmsatz[tag];
}
```

Es ist es viel sinnvoller, zu beschreiben, warum hier eine zwar durch die Variablennamen naheliegende, aber zunächst willkürliche Grenze von 365 Durchläufen verwendet wird, statt nur die offensichtliche Abbruchbedingung der Schleife zu kommentieren. Eine Verbesserung des Kommentars und der Implementierung, die auch Schaltjahre korrekt behandeln kann, könnte so aussehen – der Kommentar ist jetzt nahezu überflüssig:

```
// KORREKTUR: Gesamtumsatz aus den Tagesumsätzen eines Jahres berechnen
final int tageImJahr = tagesUmsatz.length;
for (int tag = 0; tag < tageImJahr; tag++)
{
    gesamtUmsatz += tagesUmsatz[tag];
}
```

Dokumentiere alle Methoden im öffentlichen Interface der Klasse

Im Sinne der Wiederverwendbarkeit ist es vorteilhaft, öffentliche Klassen, Methoden sowie vor allem Interfaces möglichst gut zu dokumentieren. Der Einsatz des Javadoc-Kommentarstils für alle im öffentlichen Interface von Klassen sichtbaren Methoden (`public`, `protected`) bietet sich an. Für interne Methoden kann in vielen Fällen auf eine Kommentierung verzichtet werden, wenn die Methoden entsprechend kurz sind, einen aussagekräftigen Namen haben und nur wenige Übergabeparameter besitzen.

Ein Kommentar sollte zumindest den Zweck einer Methode, idealerweise auch die Übergabeparameter sowie den Rückgabewert beschreiben. ***Die Rückgabe von `null` sollte ausdrücklich dokumentiert werden, wenn dies ein gewollter und möglicher Rückgabewert ist***. Dadurch schützt man Aufrufer vor einer sonst nicht erwarteten `NullPointerException` beim Zugriff auf den Rückgabewert. Ebenso sollte man auf Seiteneffekte hinweisen, falls sich diese nicht anderweitig vermeiden lassen.

```
/**
 * copies all contents from the passed input stream into the given output
 * stream using buffered copy.
 *
 * @param is    source input stream
 * @param os    destination output stream
 *
 * @return the amount of bytes copied
 *
 * @throws java.lang.NullPointerException if parameters are <code>null</code>
 * @throws java.io.IOException if access to streams fails
 *
 * @see java.io.InputStream, java.io.OutputStream,
 *      java.lang.NullPointerException, java.io.IOException
 */
public static int copyBuffered(final InputStream is,
                               final OutputStream os) throws IOException
```

Tipp: Dokumentation mit Javadoc

Die Erfahrung zeigt, dass eine vom Sourcecode unabhängige Dokumentation sehr schnell veraltet und außerdem häufig nicht die für Entwickler wichtigen Informationen enthält. Die Idee hinter Javadoc, dem Dokumentationswerkzeug des JDKs, ist es, aus Kommentaren im Java-Sourcecode automatisch HTML-Dokumentationen erstellen zu können. Aufgrund der Nähe zur Entwicklung besteht außerdem die Hoffnung, mit Javadoc eine aktuelle und konsistente Dokumentation für Klassen, Methoden usw. erzeugen zu können.

Der Sourcecode wird bei der Aufbereitung der Dokumentation durch das Javadoc-Tool nach Kommentaren im Javadoc-Stil (startend mit `/**` und abgeschlossen mit `*/`) durchsucht. Alle Informationen der dort enthaltenen Javadoc-Tags (beginnend mit `@`) werden für die Ausgabe in HTML ausgewertet. Eine Übersicht gebräuchlicher Javadoc-Tags liefert Tabelle 19-2. Weitere Informationen finden Sie unter `http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`.

Mit Eclipse ist es möglich, über das Menü PROJECT –> GENERATE JAVADOC eine Javadoc-Generierung durchzuführen. Noch einfacher ist es, diesen Schritt durch einen Aufruf im Build-Prozess zu automatisieren (vgl. Abschnitt 2.7.2).

Tabelle 19-2 *Gebräuchliche Javadoc-Tags*

Javadoc-Tag & Wert	Zweck
`@author <name>`	Beschreibt den Autor.
`@version <versionInfo>`	Erzeugt einen Versionseintrag.
`@param <name> <description>`	Parameterbeschreibung einer Methode.
`@return <description>`	Beschreibung des Rückgabewerts einer Methode.
`@throws <exception> <description>`	Beschreibung einer Exception, die von dieser Methode geworfen werden kann.
`@deprecated <description>`	Markiert eine veraltete Methode, die nicht mehr eingesetzt werden sollte. Zusätzlich sollte immer die `@Deprecated`-Annotation verwendet werden.
`@inheritDoc`	Kopiert die Beschreibung aus der überschriebenen Methode. Erspart damit das früher notwendige Copy-Paste für alle Kommentare aus Interfaces.
`@see reference`	Erzeugt einen Verweis auf ein anderes Element der Dokumentation.
`@since <version>`	Gibt an, ab welcher Version eine bestimmte Funktion enthalten ist. Das ist insbesondere bei Bibliotheken hilfreich.

19.3.4 Programmdesign

Halte den Sourcecode sauber

Unbenutzte Programmteile und auskommentierte Blöcke blähen den Sourcecode nur unnütz auf und machen ein Programm schlechter lesbar. Diese »Vorratsdatenspeicherung« von ehemals sinnvollen Zeilen ist durch den Einsatz einer Versionsverwaltung überflüssig. Allerspätestens nach ein paar Tagen sollten solche Programmteile gesäubert werden. Wird dieser »Frühjahrsputz« immer wieder verschoben, können Sie im Nachhinein über Kommentare wie den folgenden schmunzeln: »`Quickfix. Muss nach der Auslieferung unbedingt korrigiert werden. 7.7.2001`«. Heutzutage würde man solche Stellen mit einem Kommentar `TODO` oder `FIXME` versehen, wodurch automatisch eine Markierung und eine Aufnahme in eine Task-Liste in der IDE erfolgt.

Vermeide Seiteneffekte

Seiteneffekte sind oft unerwartet und können einem als Programmierer die Laune verderben. Ich habe einige Methoden kennenlernen müssen, die als `get()`-Methode »getarnt« Schreibzugriffe auf andere Variablen durchgeführt haben. Ein absolut extremes Beispiel ist mir in Form von Datenbank-Updates in einer `get()`-Methode begegnet. So etwas sollte man unbedingt vermeiden, damit das Programm dem »***Prinzip des geringsten Erstaunens***« (»***Principle of Least Astonishment***« (POLA)) folgt.

Vermeide übermäßige Debug- und Log-Ausgaben

Logging kann dabei helfen, Fehler zu ermitteln, wenn keine Möglichkeit des Debuggings besteht oder ein Fehler im Nachhinein analysiert werden muss. Zum Teil sieht man jedoch Sourcecode, der nur aus der Aufbereitung von Debug- und Log-Ausgaben zu bestehen scheint. Die eigentliche Programmlogik tritt in den Hintergrund. Das ist häufig ein Zeichen dafür, dass das zu lösende Problem nicht verstanden wurde. Solcher Sourcecode sollte überarbeitet werden, um diesen Ballast loszuwerden.

Vermeide Ausgaben per `System.out`

Ausgaben mit `System.out` können für einfache Anwendungen und zum schnellen Testen durchaus eingesetzt werden. Für professionelle Programme ist dies aber wenig hilfreich, da keine Konfiguration oder Persistierung der Ausgabe erfolgen kann. Man benutzt daher besser ein Logging-Framework (vgl. Abschnitt 9.4).

Verwende `final`, wenn sinnvoll

Über das Schlüsselwort `final` lässt sich steuern, ob der Wert einer Variablen konstant oder veränderlich ist. Vor allem für Parameter und Attribute kann es eine gute Hilfe sein, diese mittels `final` davor zu schützen, dass sie für andere Dinge zweckentfremdet werden. Je weniger Zustandsinformationen in einem Objekt veränderlich sind, desto übersichtlicher und wartbarer wird der Sourcecode.

Diese Hinweise gelten vor allem, wenn Sie noch nicht so viel Erfahrung gesammelt haben. Mit zunehmenden Fertigkeiten und einem in Fleisch und Blut übergegangenen sauberen Programmierstil profitiert man weniger von `final`. Trotzdem sollte man dies als Hilfe insbesondere auf Parameter anwenden.

Bevorzuge Konstanten vor Magic Numbers und Strings

Die Verwendung fest codierter Zahlenwerte, Texte und boolescher Werte im Sourcecode erschwert sowohl dessen Lesbarkeit als auch Refactorings. Die Bedeutung eines Zahlenwerts lässt sich meistens sinnvoller durch eine benannte Konstante ausdrücken. Das erhöht gleichzeitig die Verständlichkeit und die Lesbarkeit. Außerdem wird eine spätere Wartung erleichtert, falls ein solcher Wert einmal geändert werden muss. Für Strings und boolesche Werte gilt Ähnliches.

Bevorzuge wenige `return`-Anweisungen

Umfangreiche Methoden mit vielen `return`-Anweisungen sind oftmals fehlerträchtig und schwierig nachzuvollziehen. Eine Methode mit nur einem definierten Ausgangspunkt (oder wenigen) hilft, den Ablauf besser verfolgen zu können. Diese Empfehlung gilt allerdings nur dann, wenn die Methode komplex oder länger als etwa eine Bildschirmseite ist, weil dann mehrere `return`-Anweisungen ein Problem darstellen können. Denn je umfangreicher eine Methode wird, desto mehr erschweren viele `return`-Anweisungen das Herausfaktorieren von Methoden und das Nachvollziehen des Programmablaufs. Beachten Sie bitte folgende Ausnahmen: Insbesondere Shortcut-Returns am Methodenanfang für mögliche Fehlerabfragen sind fast immer eine Vereinfachung. Auch gilt: Sind Methoden kurz und übersichtlich, so ist es durchaus sinnvoll, mehrere `return`-Anweisungen zu verwenden, da eine Methode dann häufig ohne zusätzliche Hilfsvariablen und Verschachtelungen auskommt. Für diese Fälle ist es sogar schlimmer, krampfhaft zu versuchen, mehrere `return`-Anweisungen zu vermeiden, weil dann künstliche Hilfsvariablen eingeführt werden (müssen).

Vermeide `return` in `catch`/`finally`-Blöcken

Verwendet man ein `return` in `catch`-Blöcken, so besteht die Gefahr, dass beim Hinzufügen eines `finally`-Blocks der zurückgelieferte Wert verändert wird, wie es im folgenden konstruierten Beispiel gezeigt ist:

```
public static String test()
{
    try
    {
        throw new IllegalStateException("Exception simulieren!");
    }
    catch (final IllegalStateException ex)
    {
        return "Im Fehlerfall";
    }
    finally
    {
        return "Tatsächlich";
    }
}

public static void main(final String[] argv)
{
    System.out.println(test());
}
```

Listing 19.1 *Ausführbar als* **'TESTRETURNINCATCH'**

Der `finally`-Block wird immer *nach allen* anderen Aktionen, also als letzte Anweisungsfolge abgearbeitet. Es kommt hier zur Ausgabe von "`Tatsächlich`", und das obwohl der `catch`-Block und das dortige `return` ausgeführt werden. Das `return` im `finally` übersteuert aber alle vorherigen `return`-Anweisungen. Daher sollte man ein `return` nur mit viel Bedacht in `catch`- und `finally`-Blöcken nutzen.

Prüfe Eingabeparameter öffentlicher Schnittstellen auf Gültigkeit

Vor allem an Komponenten- oder Modulgrenzen sollten die Parameter von öffentlichen Methoden auf Gültigkeit geprüft werden, um fehlerhafte Werte zu erkennen und zurückzuweisen.[3] Interne Methoden können sich somit auf einen gültigen Objektzustand verlassen und müssen nicht zwingend weitere Parametertests durchführen. Eine Parameterprüfung soll also verhindern, dass ungültige Parameterwerte an andere Methoden weitergegeben werden und dort Schaden anrichten. Dies folgt den Gedanken von »***Design by Contract***« (vgl. gleichnamigen Praxistipp in Abschnitt 3.1.5). Dadurch werden falsch initialisierte Objekte vermieden.

Verwende Assertions oder Exceptions zur Absicherung für Pre- und Post-Conditions

Teilweise müssen gewisse Annahmen über den Zustand des Programms bzw. die Wertebelegung von Variablen getroffen werden. Zur Auswertung kann man das Schlüsselwort `assert` nutzen, manuell die gewünschte Bedingung als booleschen Ausdruck prüfen und gegebenenfalls bei Verstoß eine Exception auslösen oder aber die Prüfung mithilfe von Google Guavas `Preconditions` (vgl. Abschnitt 9.2.4) erledigen. Bei Problemen im Programmablauf und Inkonsistenzen im Zustand wird ein etwaiger Denkfehler sofort sichtbar. Bedenken Sie aber, dass die Auswertung von Assertions mit `assert` standardmäßig ausgeschaltet ist und explizit aktiviert werden muss (vgl. Abschnitt 5.6.5), weshalb Exceptions zur Absicherung zu bevorzugen sind.

Vermeide die sorglose Verwendung und Rückgabe von `null`

Man sollte `null` möglichst sparsam nutzen und stattdessen bevorzugt gültige Referenzen. Seit der Einführung der Klasse `Optional<T>` in Java 8 zur Modellierung optionaler Werte gilt diese Regel ganz besonders: Nicht verfügbare Werte etwa bei Suchen lassen sich besser über `Optional<T>` (vgl. Abschnitt 6.5) anstatt durch `null` modellieren.

Ähnliches gilt für Rückgabe vom Wert `null`. Das sollte von Aufrufern unbedingt beachtet werden, um Sonderbehandlungen durchzuführen. Wird dies vergessen, so kommt es zu `NullPointerException`s. Neben dem saloppen Umgang mit `null` ist auch der pedantische nicht angemessen. Wieso? Dadurch findet sich im Sourcecode eine Vielzahl an `null`-Prüfungen, ob sinnvoll oder auch nicht. Fakt ist jedoch, dass sich der Sourcecode dadurch oftmals nur aufbläht und unleserlich wird. Zudem kann diese übervorsichtige Prüfung auch Fehler verschleiern, die besser während der Implementierungsphase des Programms aufgetreten wären.

Um Klarheit für Nutzer zu erzielen, sollte eine mögliche Rückgabe von `null` ausdrücklich in der Methodendokumentation erwähnt werden. Oftmals sind das NULL-OBJEKT-Muster (vgl. Abschnitt 18.3.2) oder die Klasse `Optional<T>` erwägenswerte

[3] Man kann sich dies in etwa wie Grenzkontrollen vorstellen. Innerhalb des Landes wird die Identität natürlich nicht ständig geprüft, sondern nur einmalig bei einer Ein- oder Ausreise.

Alternativen – anstatt vorschnell oder aus (schlechter) Gewohnheit den Wert `null` zurückzuliefern.

Für den (hoffentlich) seltenen Fall, dass eine Referenz doch einmal einen `null`-Wert enthalten kann, sollte man vor einem Zugriff darauf prüfen und besser fehlertolerant arbeiten. Zudem gibt man aber für den unerwarteten `null`-Fall eine Warnmeldung in eine Log-Datei aus.

Prüfe Rückgabewerte und behandle Fehlersituationen

Rückgabewerte von Methoden sollten vom Methodenaufrufer ausgewertet werden, damit angemessen auf eine mögliche Fehlersituation reagiert werden kann. Geschieht dies nicht, so werden Fehler verschleiert oder sie machen sich entweder gar nicht oder nur durch merkwürdiges Programmverhalten bemerkbar. Im UI wird beispielsweise eine Erfolgsmeldung ausgegeben, obwohl eine Aktion nicht vollständig durchgeführt werden konnte, etwa aufgrund eines Problems bei einem Datenbank- oder Remote-Zugriff.

Tipp: Sorglose Reaktion auf Fehler

Fehler durch den sorglosen Umgang mit Rückgabewerten haben mich einiges an Arbeit gekostet. Teilweise ist es viel schwieriger, solche Fehler zu lokalisieren, als diese zu beheben. Oftmals hätte zumindest ein Hinweis in einer Log-Ausgabe eine Fehlersuche (enorm) verkürzen können.

Wenn man – aus welchen Gründen auch immer – momentan nicht in der Lage ist, eine angemessene Behandlung für eine Fehlersituation oder einen unerwarteten Rückgabewert (etwa `null`) in den Sourcecode einzubauen, muss wenigstens im Log-Level `warn` ein Hinweis in eine Log-Datei geschrieben werden. Ansonsten kommt es zu dem sogenannten »***Silent Fail***« oder »verschluckten Fehlern«, die extrem schwierig aufzufinden sind: Im Normalfall läuft das Programm einwandfrei, aber im Fehlerfall erhält man keinen Hinweis auf ein Problem.

Behandle auftretende Exceptions wenn möglich

Ebenso wie es problematisch ist, Rückgabewerte zu ignorieren, gilt dies analog für Exceptions: Wünschenswert ist eine sinnvolle Fehlerbehandlung. Falls diese lokal nicht möglich ist, sollte eine Exception weiter propagiert werden, um anderen Programmteilen zu ermöglichen, eine Fehlersituation adäquat zu behandeln, beispielsweise diese in einer Log-Datei zu protokollieren. Werden zusätzliche Kontextinformationen, etwa der Stacktrace, die Belegung lokaler Variablen und von Attributen, ausgegeben, so erleichtert dies häufig eine spätere Fehlersuche.

Vermeide `catch (Exception ex)` und `catch (Throwable th)`

Exceptions möglichst spezifisch abzufangen, hilft dabei, diese sinnvoll behandeln zu können. Anhand des Typs der Exception sollte dann eine passende Fehlerbehandlung erfolgen. Werden dagegen Exceptions unspezifisch mit `catch (Exception ex)` oder `catch (Throwable th)` abgefangen, führt dies dazu, dass alle möglichen Fehlersituationen mit diesem `catch`-Block bearbeitet werden. Treten unerwartet weitere Exceptions auf, werden diese dann so behandelt wie erwartete Exceptions. Unerwartete Exceptions sollten allerdings besser an andere Programmteile weiter propagiert werden, um sie dort in spezifischen `catch`-Blöcken angemessen behandeln zu können.

Im Extremfall werden auch `RuntimeException`s und `Error`s abgefangen. Diese stellen jedoch schwerwiegende Fehler dar, die man – außer in Frameworks – in der Regel nicht in einem `catch`-Block behandeln sollte.

19.3.5 Klassendesign

Bevorzuge lokale Variablen gegenüber Attributen

Wenn möglich sollte man lokale Variablen anstelle von Attributen verwenden: Lokale Variablen sind immer Thread-sicher, da sie niemals von mehreren Threads gleichzeitig geschrieben bzw. gelesen werden können. Außerdem verringern sie die Anzahl möglicher Objektzustände und machen ein Programm dadurch oftmals verständlicher.

Vermeide statische Attribute

Statische Attribute haben die unangenehme Eigenschaft, dass sie vom Garbage Collector nicht freigegeben werden können, da erst nach Programmende die letzte Referenz darauf erlischt.[4] Diese »Dauerreferenzierung« ist in den meisten Fällen unerwünscht. Nützlich sind statische Variablen jedoch für Metadaten von Klassen, etwa zur Bereitstellung von Loggern.

Schlimmer als statische Attribute sind statische Collections. Diese können ungehindert während der Programmlaufzeit wachsen, wenn sie nicht in ihrer Größe beschränkt sind. Programmierfehler können so leicht Out-of-Memory-Situationen provozieren.

Greife auf Attribute bevorzugt über Methoden zu

Die Forderung, auf Attribute nicht direkt, sondern über Methoden zuzugreifen, folgt dem OO-Gedanken der Kapselung. Innerhalb von Klassen kann man beim Zugriff auf eigene Attribute auf den Einsatz von Zugriffsmethoden verzichten. Dies gilt vor allem, wenn man dort auf private Attribute zugreift. Auf jeden Fall sollte man bei Attributen entweder konsequent auf `get()`-Methoden verzichten oder diese konsequent einsetzen. Mischt man den Zugriff, fragt man sich als Betrachter der Klasse immer, worin der Unterschied zwischen einem direkten Zugriff und einem `get()`-Aufruf liegt.

[4] Sofern nicht zuvor eine Zuweisung mit `null` erfolgt.

Vermeide viele öffentliche `set()`-Methoden

Klassen mit vielen öffentlichen `set()`-Methoden erlauben eine feingranulare Änderung des internen Zustands von außen und widersprechen damit dem objektorientierten Gedanken, dass ein Objekt seinen Zustand kapseln und nach außen möglichst nur fachliche Operationen (meistens in Form von Business-Methoden) anbieten sollte.

Minimiere Zustandsänderungen

Der Einsatz von Read-only-Interfaces (vgl. Abschnitt 3.4.1) kann vor ungewollten Veränderungen von außen schützen. Manchmal ist es zusätzlich möglich, den Objektzustand mithilfe des IMMUTABLE-Musters (vgl. Abschnitt 3.4.2) vollständig gegen Modifikationen abzusichern. Auf jeden Fall sollten all diejenigen Attribute `final` definiert werden, die nach ihrer Initialisierung unveränderlich sein sollen. Ungewollte und inkonsistente Änderungen am Objektzustand werden dadurch vermieden.

Vermeide es, Referenzen auf interne Datenstrukturen zurückzugeben

Wie schon gerade eben angedeutet, sollte ein Objekt seinen Zustand kontrollieren. Dies ist nahezu unmöglich, wenn Referenzen auf interne Datenstrukturen herausgegeben werden. Stattdessen sollte man entweder Kopien davon zurückgeben oder die `unmodifiable`-Wrapper des Collections-Frameworks nutzen. Selbst wenn nur Daten lesende Methoden angeboten werden, lauern noch versteckte Probleme durch die Referenzsemantik von Java: Es können Änderungen am Zustand der in Containerklassen gespeicherten Objekte erfolgen.

Programmiere gegen Interfaces oder abstrakte Klassen

Sofern Abstraktion, Austauschbarkeit und Wiederverwendbarkeit wichtige Kriterien beim Design sind, empfiehlt sich der Einsatz von Interfaces und abstrakten Klassen, um konkrete Implementierungen zu verstecken.

Ein gängiges Beispiel ist bei der Deklaration einer Variablen die Nutzung des Interface `List<E>` statt der konkreten Klasse `ArrayList<E>`. Damit wird neben einer loseren Kopplung zudem sichergestellt, dass nur die Methoden des angebotenen Interface aufgerufen werden (können). Wenn lose Kopplung und Abstraktion so gut sind, könnte man auf die Idee kommen, nahezu immer ein noch allgemeineres Interface, im Beispiel also `Collection<E>`, zu verwenden. ***Eine Generalisierung sollte aber nur so weit fortgesetzt werden, wie es keinen Aussagekraftverlust für den gewünschten Einsatzzweck gibt***. Betrachten wir zur Verdeutlichung eine Verwaltung von Listenern, wobei jeder von diesen nur einmal angemeldet werden soll, um doppelte (bzw. potenziell mehrfache) Benachrichtigungen zu unterbinden. Als ungünstige Variante ist einführend eine Definition als `Collection<Subscriber>` gezeigt:

```
// kommuniziert Mengeneigenschaft nicht
private final Collection<Subscriber> subscribers = new HashSet<>();
```

Durch diese starke Abstraktion verliert man jedoch die wichtige Information über die Mengeneigenschaft. Daher verwendet man passenderweise ein `Set<Subscriber>`. Die folgende Variante bietet eine gute Abstraktion und liefert ein für den Anwendungsfall zugeschnittenes Interface:

```
private final Set<Subscriber> subscribers = new HashSet<>();
```

Halte das Interface übersichtlich

Ein Interface sollte möglichst nur Methoden für genau einen klar abgegrenzten Aufgabenbereich definieren. Steigt die Anzahl der in einem Interface deklarierten Methoden über einen gewissen Wert (etwa fünf bis zehn Methoden), so deutet dies darauf hin, dass das Interface zu viel Funktionalität anbietet und vermutlich auch das Single Responsibility Principle (SRP) verletzt ist und das Interface besser in mehrere separate, voneinander unabhängige Interfaces aufgespalten werden sollte. Diesen letzten Punkt adressiert auch das Interface Segregation Principle (ISP) – das ebenso wie das SRP eines der SOLID-Prinzipien für guten OO-Entwurf (vgl. Abschnitt 3.5.3) darstellt.

Sorge für Lesbarkeit und die richtige Abstraktionsebene

Innerhalb von öffentlichen Methoden sollten nicht zu viele Implementierungsdetails gezeigt werden. Schauen wir uns folgende beiden Methodenaufrufe an, wobei RBL eine branchenübliche Abkürzung für ein Rechner-basiertes Leitsystem ist:

1. `systemStateService.isSystemAlive(SystemType.RBL);`
2. `isRBLAlive();`

Der erste Methodenaufruf erfolgt auf Realisierungsebene und zeigt viele Implementierungsdetails. Er macht die beteiligte Klasse sowie den Methodenaufruf sichtbar. Die zweite Variante ist von ihrer Abstraktionsebene viel höher angesiedelt. Die Realisierung auf dieser fachlichen Ebene zeigt keine Details und versteckt die dahinterliegende Komplexität. Öffentliche Methoden sollten bevorzugt auf einer solch hohen Abstraktionsebene als verhaltensdefinierende Business-Methoden formuliert werden. Je geringer die Sichtbarkeit, desto mehr nimmt die Abstraktion von technischen Details gewöhnlich ab: Private Methoden dürfen demnach beliebige Implementierungsdetails zeigen.

Verstecke Implementierungsdetails

Benutze bevorzugt eine möglichst kleine Sichtbarkeit (und das ist nicht `PUBLIC` ;-)). Attribute sollten `private` oder Package-private definiert werden. Für Hilfsmethoden gilt dies ebenfalls. Business-Methoden, die nach außen sichtbare Funktionalität anbieten, sind dagegen meistens `public`. Nur wenn Klassen so entworfen sind, dass sie eine Erweiterbarkeit durch Überschreiben von Methoden erlauben, sollten auch Methoden mit der Sichtbarkeit `protected` existieren.

Definiere abstrakte Methoden möglichst `protected`

Zur Spezialisierung von Verhalten in Subklassen dienen abstrakte Methoden in Basisklassen. Werden diese Methoden `protected` definiert, können abgeleitete Klassen auf sie zugreifen – Zugriffe aus demselben Package sind auch möglich.

Beachte die maximale Methodenlänge von ca. 30 – 50 Zeilen

Eine Methode sollte genau eine Teilaufgabe realisieren. Leider sieht man teilweise eine Verquickung von Informationsbeschaffung und -verarbeitung. Das stört die Orthogonalität und Wiederverwendbarkeit. Methoden können dann nicht wie kleine Bausteine zu neuen, größeren Methoden kombiniert werden.

Tipp: Methodenlänge

Man findet leider immer wieder Methoden, deren Länge die 100 Zeilen mehrfach übersteigt. Wiederholt sind mir Methoden mit über 500 Zeilen aufgefallen. Derartige Monstren sind nahezu unwartbar. Man sollte versuchen, sie durch das Herausfaktorieren von Hilfsmethoden beherrschbar zu machen. Das Überarbeiten ist jedoch kein Selbstzweck, sondern sollte immer einen (Business-)Nutzen haben, etwa um Erweiterungen einfacher integrieren zu können oder aber ein Codereview zu erleichtern. Je nach Intention kann die Verbesserung nach ein oder zwei Extraktionen beendet werden. Manchmal ist es aber auch notwendig, den Vorgang so lange zu wiederholen, bis die ursprüngliche Methode eine vernünftige Struktur und Länge besitzt.

Auf was möchte ich hinaus? Weniger ist manchmal mehr: Je mehr man sich vornimmt und je mehr man ändert, desto größer ist auch die Gefahr, sich zu verzetteln oder ein paar Flüchtigkeitsfehler zu machen. Bleiben Sie also fokussiert und arbeiten sie iterativ.

Beachte die maximale Klassenlänge von ca. 500 – 1000 Zeilen

Eine Klasse sollte nur für einen Aufgabenbereich zuständig sein. Wird eine Klasse immer länger, so deutet dies darauf hin, dass sie für zu viele verschiedene Dinge verantwortlich ist. Wie bereits für die Methodenlänge angedeutet, stört dies die Orthogonalität und Wiederverwendbarkeit – in diesem Fall der Klasse.

Vermeide zu viele Referenzen auf andere Klassen

Werden von einer Klasse viele andere Klassen referenziert (mehr als ca. sieben), kommt es schnell zu »Objekt-Spaghetti«. Eine solche Situation entsteht vielfach dadurch, dass die Aufteilung von Funktionalität auf Klassen nicht sinnvoll durchgeführt wurde.

19.3.6 Parameterlisten

Vermeide überflüssige Parameter

Zum Teil sieht man Methoden, die Parameter in Form eines Objekts übergeben bekommen und zusätzlich separat noch einen Parameter, der aus dem Objekt stammt. Das folgende Listing verdeutlicht dies anhand der Methode `handleMsg(byte[], Device, String)`, die auch Informationen aus dem Objekt `device` erhält:

```
getTelegramHandler().handleMsg(tmpBytes, device, device.getAddress());
```

Der über `device.getAddress()` ermittelte Wert könnte als Parameter entfallen und in der Methode selbst aus dem übergebenen `device` ermittelt werden.

Vermeide mehrere gleiche Typen aufeinander folgend in der Parameterliste

Mehrere gleiche Typen aufeinander folgend in einer Parameterliste können problematisch sein, da die Gefahr der Verwechselung von Positionen besteht. Je mehr Parameter gleichen Typs verwendet werden, desto stärker macht sich das Problem bemerkbar. Nachfolgende Signaturen provozieren geradezu Fehler:

```
public void fillRect(int x1, int y1, int x2, int y2, long colorRGB)
public void fillRect(int red, int green, int blue,
                     int x, int y, int width, int height)
```

Mithilfe von PARAMETER VALUE OBJECTS (vgl. Abschnitt 3.4.5) kann man Parameter zusammenfassen und erreicht so kürzere, typsichere und verständlichere Signaturen. Damit kann man Anwendungsfehler nahezu ausschließen:

```
public void fillRect(Point point1, Point point2, long colorRGB)
public void fillRect(Color fillColor, Point point, Dimension size)
```

Halte die Parameterliste kurz

Aus der Psychologie ist bekannt, dass ein Mensch sich etwa 7 ± 2 Dinge im Kurzzeitgedächtnis merken kann. Daher sollte man maximal sieben Parameter verwenden. Übersichtlicher sind natürlich weniger Parameter, am besten nur ein bis vier. Das haben wir gerade schon gesehen, ebenso wie den Einsatz von PARAMETER VALUE OBJECTS.

Halte die Reihenfolge von Parametern bei Methodenaufrufen konsistent

Besitzen verschiedene Methoden gleiche Parameter und rufen sich auf, so sollte die Reihenfolge möglichst einheitlich gehalten werden. Folgendes Listing zeigt ein Negativbeispiel: Die Parameter `a` und `b` sind für die Methoden `doThis(TypeA, TypeB)` und `doThat(TypeB, TypeA)` in ihrer Reihenfolge vertauscht:

```
public int doThis(TypeA a, TypeB b)
{
    return doThat(b,a);
}

public int doThat(TypeB b, TypeA a)
{
    // ...
}
```

Warum ist das ungünstig? Die Begründung ist einfach: Ändert sich die Reihenfolge der Parameter ständig, so wird das Verwenden der Funktionen mühselig, da man sich jedes Mal wieder Gedanken über die Aufrufreihenfolge machen muss. Zudem kann durch Vertauschen der Position eines Parameters diesem eine andere semantische Bedeutung gegeben werden. Solche Fehler sind schwierig zu finden.

19.3.7 Logik und Kontrollfluss

Vermeide explizite `true`-/`false`-Literale in `if`-/`while`-Anweisungen

Verwendet man in Abfragen von booleschen Variablen explizit die Schlüsselwörter `true` oder `false`, so wird dies sperrig und unleserlich. Hier ein Beispiel:

```
if (attributeValue.isNullValue() == false)
    if (isEditable == true)
```

Besser lesbar werden die Abfragen durch Weglassen der booleschen Konstanten und die Kombination zu einer einzeiligen Bedingung:

```
if (!attributeValue.isNullValue() && isEditable)
```

Verwende möglichst wenige `else if`-Anweisungen

Komplexe Bedingungen mit mehreren `else if`-Varianten machen Programme recht schnell unübersichtlich. Mit zunehmender Anzahl wird es immer schwieriger, die Bedingungen so zu formulieren, dass sich diese tatsächlich gegenseitig ausschließen. Folgende Zeilen zeigen ein Negativbeispiel:

```
if (x >= 3 && y < 11 && z == 300 && command.equals("one"))
{
    System.out.println("1");
}
else if (x < 3 && (y > 10 || y < 10))
{
    System.out.println("2");
}
// für x == 3 und z == 300 Überschneidung mit Fall 1
else if (x <= 3 && z == 300)
{
    System.out.println("3");
}
```

Einfache Bedingungen mit mehreren `else if`-Anweisungen sind akzeptabel:

```
if (command.equalsIgnoreCase("cd"))
{
    executeCommand(new CdCommand());
}
else if (command.equalsIgnoreCase("list"))
{
    executeCommand(new ListCommand());
}
```

Allerdings lässt sich auch hier eine elegantere Lösung finden – etwa mit einem gemeinsamen Interface und Polymorphie.

Verwende nicht mehr als drei bis vier Logikauswertungen in einem `if`

Werden sehr viele, mitunter auch komplexe Bedingungen in einer `if`-Anweisung geprüft, so leidet die Übersichtlichkeit, wie dies folgendes Beispiel zeigt:

```
if (hasSpecialTexts() // Sondertext
    || !deviceIsActive // Anlage deaktiviert
    || !displayIsActive // Anzeiger deaktiviert
    || !myDisplay.isEnabled() // Anzeiger gesperrt
    || noDeps // keine Abfahrten
    || ((getFallbackCode() == FailureConstants.NO_RADIOCONNECT) && !
          showWithinFallback)
    || ((getFallbackCode() == FailureConstants.NO_AVMCONNECT) && !ready)
    || (getFallbackCode() == FailureConstants.NO_DISPCONNECT)
    || (getFallbackCode() == FailureConstants.NO_PIDCONNECT))
```

Das Einführen von einigen Prüfmethoden kann die Lesbarkeit ungemein erhöhen:

```
if (   hasSpecialTexts()
    || deviceOrDisplaysInactive()
```

Verwende den Conditional-Operator mit Bedacht

Einfache `if`-Bedingungen mit einigen Zeilen Sourcecode lassen sich manchmal elegant mit dem Conditional-Operator formulieren. Folgende Zeilen dienen als Basis:

```
final String strType;
if (isFolder)
    strType = "directory";
else
    strType = "file";
```

Der Einsatz des Conditional-Operators vereinfacht den Sourcecode wie folgt:

```
final String strType = (isFolder ? "directory" : "file");
```

Folgender Conditional-Operator ist durch die Methodenaufrufe in den Auswertungen (schon fast) zu kompliziert, um leicht verständlich zu sein und eingesetzt zu werden:

```
return oldState != null ? oldState.compareTo(newState) == 0 :
                         newState.equals(null);
```

Wirklich verwirrend und zu komplex ist aber folgendes Konstrukt:

```
final Double value = value1 == null ? value2 : value2 == null
                                    ? value1 : new Double(value1 + value2);
```

Tatsächlich ist das – wie nahezu alle Beispiele – Produktivcode, den ich so vorgefunden habe. Bitte beachten Sie die Hinweise aus dem folgenden Kasten.

Vorsicht: Änderungen zur Lesbarkeit mit Folgen

Beim Anblick des obigen Konstrukts hatte ich zunächst Verständnisschwierigkeiten und eine erste Idee bestand darin, den scheinbar überflüssigen Konstruktoraufruf `new Double()` wegzulassen. Gesagt, getan: Neben der Vereinfachung des Konstrukts hat sich dadurch aber ein Applikationsfehler eingeschlichen: Nun konnte eine `NullPointerException` auftreten. Das liegt an Spezialitäten bei der Ausführung von Auto-Boxing und Auto-Unboxing zur Typkonvertierung im Zusammenhang mit dem Conditional-Operator.

Bei einem so verwirrenden Konstrukt bietet es sich zur Vereinfachung an, genau das als Sourcecode zu schreiben, was man logisch ausdrücken möchte, etwa wie folgt:

```
public static Double nullsafeAdd(final Double value1, final Double value2)
{
    if (value1 == null && value2 == null)
    {
        return null;
    }
    if (value1 == null)
    {
        return value2;
    }
    if (value2 == null)
    {
        return value1;
    }

    return value1 + value2;
}
```

Das Ganze ist zwar damit deutlich länger und besitzt sogar vier `return`s, ist aber trotzdem absolut klar und verständlich.

19.4 Sourcecode-Prüfung mit Tools

In den vorherigen Abschnitten wurden diverse Regeln aufgestellt, deren Einhaltung die Qualität des Sourcecodes verbessern kann. Eine manuelle Prüfung umfangreicher Projekte ist durch Sourcecode-Inspektion und Codereviews jedoch mühevoll und extrem zeitaufwendig. Das macht die Unterstützung durch Tools wünschenswert.

In den folgenden Abschnitten gehe ich zunächst auf die Grundlagen der Auswertung durch Tools ein. Anschließend werden einige Kennzahlen zur Bewertung der Güte von Programmen vorgestellt. Nachfolgend wird dann die automatisierte Sourcecode-Prüfung durch Tools thematisiert. Dort werden Checkstyle, PMD und SonarQube kurz beschrieben und gezeigt, wie man diese in den automatischen Build-Lauf integrieren kann.

Sourcecode-Checker und statische Analyse

Die Auswertungen von Sourcecode-Checkern basieren auf einer sogenannten ***statischen Analyse***, die den Sourcecode bzw. den vom Java-Compiler generierten Bytecode untersucht. Es ist demnach keine Ausführung des Programms notwendig, wie dies bei der sogenannten ***dynamischen Analyse***, beispielsweise der eines Profilers, der Fall ist. Durch eine statische Analyse können somit keine Aussagen zum Laufzeitverhalten gemacht werden. Es lassen sich jedoch unter anderem folgende Schwachstellen erkennen:

- Duplikation von Sourcecode
- Lange Klassen und Methoden
- Umfangreiche Parameterlisten
- Unbenutzte Methoden und Variablen

Viele solcher Probleme auf Sourcecode-Ebene lassen sich mithilfe von Codereviews und einer Sammlung von Coding Conventions aufwendig, aber kaum vollumfänglich feststellen, wie das automatisiert durch Tools möglich ist. Architektur- und Designprobleme sind dagegen meistens schwieriger zu erkennen und können zum Teil nur über die Auswertung der in Abschnitt 19.4.1 vorgestellten Kennzahlen, sogenannten ***Metriken***, ermittelt werden. Anhand dieser lassen sich beispielsweise folgende Probleme aufdecken:

- Viele Abhängigkeiten zwischen Packages und Klassen
- Schwache Kohäsion innerhalb von Klassen
- Starke Kopplung zwischen Klassen
- Tiefe oder breite Vererbungshierarchien

Nicht alle von den Tools aufgedeckten Schwachstellen sind tatsächlich auch problematisch. Es handelt sich dabei eher um Sourcecode-Abschnitte, die möglicherweise Probleme enthalten und daher genauer untersucht werden sollten. Erfolgt eine Integration des Prüftools in die IDE, so kann man die Stellen direkt anspringen. Eine Auswertung

und Bearbeitung ist damit ohne Verlassen der IDE möglich. Das erleichtert eine Fehlersuche bereits während der Programmierung.

Mit Tools lassen sich festgestellte Mängel als Warnung oder Kompilierfehler klassifizieren. Dies kann man auch zur Einhaltung einiger wichtiger Regeln nutzen, wie es der folgende Praxistipp beschreibt.

Tipp: Einhaltung von Codierungsregeln

Will man die Einhaltung einer speziellen Regel forcieren, so wertet man einen ***Verstoß dagegen als Fehler*** und verhindert somit ein erfolgreiches Kompilieren. Dieser Trick hat sich als hilfreich erwiesen, ***um gewissen Regeln die gewünschte Aufmerksamkeit zu verleihen***. Allerdings sollte dieses Vorgehen nicht überstrapaziert werden, also lediglich für einige als wichtig angesehene Regeln angewendet werden.

Finden Entwickler nach der Aktivierung von neuen Regeln zu viele Fehler vor, so sind diese schnell entmutigt. Wichtig ist es daher, dass die Regeln allen Entwicklern gut bekannt und auch von allen akzeptiert werden – dazu tragen Schulungen und Workshops bei. Zudem hat sich eine schrittweise Einführung und Verschärfung der Regeln als vorteilhaft erwiesen. Die Grundlage bilden mehrere unterschiedlich strenge Regelsätze. Dabei kann die strengste Auslegung nahezu jeden Regelverstoß als Fehler ahnden. Die niedrigste Stufe dient als Einstieg in die Sourcecode-Prüfung und erleichtert die Akzeptanz solcher Maßnahmen bei Skeptikern.

19.4.1 Metriken

Wünschenswert ist es, die Qualität eines Programms anhand einiger Kennzahlen ablesen zu können. Dazu wird der Sourcecode analysiert und unter verschiedenen Aspekten betrachtet, etwa der Anzahl der Zeilen oder der Anzahl der Klassen bzw. Interfaces. Die ermittelten Werte kann man jeweils auf eine sogenannte ***Metrik*** abbilden. Diese liefert eine bewertbare Kennzahl für eine spezielle Eigenschaft eines Softwaresystems. Definiert man erlaubte Wertebereiche für die jeweiligen Metriken, so kann man durch fortlaufende Prüfungen des Sourcecodes eines Projekts die Güte der Einhaltung und Trends ablesen. Abweichungen vom gewünschten Ziel lassen sich erkennen und durch entsprechende Maßnahmen korrigieren. Anhand der ermittelten Kennzahlen können dann Auswirkungen von Änderungen bewertet werden. Zudem lassen sich durch Metriken eventuelle Schwachstellen leichter lokalisieren. ***Allerdings ist eine abschließende Bewertung, ob tatsächlich ein Problem vorliegt, häufig nur durch einen erfahrenen Entwickler möglich, und nicht allein aufgrund einer Kennzahl***.

Folgende Aufzählung nennt einige Bewertungskriterien und Metriken zur Beurteilung der Güte des Sourcecodes. Weitere Informationen finden Sie im Buch »Object-Oriented Metrics: Measures of Complexity« von Brian Henderson-Sellers [27].

- **Lines Of Code (LOC)**
  Die Anzahl der Zeilen im Sourcecode ist wohl die bekannteste Messgröße und kann ein Maß für die Komplexität eines Softwaresystems sein. Eine Weiterentwicklung dieser Metrik besteht darin, lediglich Zeilen mit Anweisungen, also weder Zeilen mit Kommentar noch Leerzeilen, zu zählen. Diese Metrik wird als NCSS (Non Commented Sourcecode Statements) oder auch MLOC (Method Lines of Code) bezeichnet und ermittelt normalerweise eine aussagekräftigere Kennzahl als LOC.

  Bewertung: In der Regel gilt ein einfacher Zusammenhang: Je mehr Zeilen ein Programm besitzt, desto komplizierter, fehleranfälliger und unübersichtlicher ist es. Die Zeilenmetriken geben aber nur einen groben Überblick über den Umfang eines Programms, jedoch nicht über dessen Qualität. Somit besitzen diese Metriken lediglich eine beschränkte Aussagekraft.

- **Cyclomatic Complexity / McCabe-Metrik**
  Die Cyclomatic Complexity oder McCabe-Metrik misst die Anzahl alternativer Wege durch ein Programm. Komplexe `if-else`-Konstrukte und hohe Schachtelungstiefen erhöhen diesen Wert. Vielfach steigt mit zunehmender Komplexität, z. B. durch eine schlecht gewählte Strukturierung, der Aufwand, den Programmablauf nachzuvollziehen. Als Faustregel sollte die Cyclomatic Complexity möglichst kleiner als 15 sein. Bei nicht zu tief verschachtelten `if`s können aber auch höhere Werte unproblematisch sein. Ab einer gewissen Grenze, etwa dem Wert 50, kann man den Sourcecode als immer schwieriger wartbar klassifizieren.

  Bewertung: Diese Metrik ist mit Vorsicht zu genießen, da auch für übersichtliche, aber mehrfach aufeinander folgende `if`- oder `case`-Anweisungen hohe Kennzahlen berechnet und diese damit negativ bewertet werden. Nichtsdestotrotz ist mit dieser Metrik in der Regel eine bessere Aussage über die Qualität des Sourcecodes zu erzielen als mit den Zeilenmetriken wie LOC etc.

- **Dokumentation**
  Aus der Anzahl und der Güte der Kommentare innerhalb eines Programms kann man Rückschlüsse auf dessen Wartbarkeit ziehen. Das ist dadurch begründet, dass Kommentare normalerweise zur Strukturierung und Lesbarkeit eines Programms beitragen und damit ein späteres Nachvollziehen (auch von anderen Entwicklern) erleichtern. Eine fehlende Dokumentation[5] von Klassen und öffentlichen Methoden zeugt meistens von einem unausgereiften Design oder von Zeitdruck. ***Wenn man nicht in der Lage ist, die Aufgaben von (öffentlichen) Methoden und Klassen in Worte zu fassen, fehlt häufig das Verständnis für das zu lösende Problem***. Dementsprechend unstrukturiert ist dann oftmals auch die Implementierung. Es gibt ebenso den Fall der übermäßigen oder überflüssigen Dokumentation. Nicht jedes Stück Sourcecode muss kommentiert werden. Private Methoden können vielfach bereits ohne Kommentar allein durch aussagekräftig gewählte Methoden- und Variablennamen sowie durch eine kurze Methodenlänge verständlich sein. Guter Sourcecode ist oft (nahezu) selbsterklärend. Dann sollte auf eine eher »künstliche«

[5] Hierunter fallen auch die von IDEs automatisch generierten Javadoc-Vorlagen ohne Inhalt.

Kommentierung von Methoden verzichtet werden, sofern sie nicht Teil der öffentlichen Schnittstelle sind.

Bewertung: Die Diskussion zeigt, dass diese Metrik kritisch betrachtet werden sollte: Ein Tool kann sinnvolle Kommentare nicht von nichtssagenden oder automatisch generierten Kommentaren unterscheiden. Weiterhin besteht das Problem der Konsistenz zwischen Dokumentation und Sourcecode. Selbst wenn die Voraussetzung für ein einfaches Verständnis sowohl durch einen aussagekräftigen Kommentar als auch durch gut lesbaren Sourcecode vorliegt, kann es Verwirrung stiften, wenn das Programm etwas anderes macht, als es der Kommentar ausdrückt.

- **OO-Metriken nach Chidamber und Kemerer**
  OO-Metriken messen hauptsächlich Ursachen für Probleme im Design, die ich bereits in Kapitel 3 im Zusammenhang mit Kohäsion, Kapselung und Vererbung vorgestellt habe. Je stärker z. B. die Abhängigkeiten zwischen verschiedenen Klassen sind, desto mehr kann man von »Objekt-Spaghetti« sprechen. Durch eine fehlende Kapselung pflanzen sich Änderungen meistens unangenehm fort. Dies wird unter anderem durch die bereits 1994 von Shyan R. Chidamber und Chris F. Kemerer [9] in einem Artikel vorgestellten Metriken erfasst. Das sind etwa folgende:

  - **Depth in Inheritance Tree (DIT)**
    Die Metrik DIT beschreibt die Tiefe des Ableitungsbaums, also die Anzahl der Oberklassen einer betrachteten Klasse. Einerseits wird dadurch die Wiederverwendung gemessen, andererseits kann ein hoher Wert aber auch ein Hinweis auf falsche Abstraktionen sein. In Java ist der minimale Wert für DIT der Wert zwei, da jede Klasse von der Basisklasse `Object` erbt. Sinnvolle Werte für DIT liegen im Bereich von zwei bis vier. Für Frameworks und einige Swing-Komponenten kommt durch Vererbung von Basiskomponenten sogar für DIT ein Wert von fünf bis acht zustande.
  - **Number Of Children (NOC)**
    Die Metrik NOC gibt die Anzahl der direkten Subklassen einer Klasse an und misst demnach die Breite des Ableitungsbaums. Diese kann als ein Indiz für die Wichtigkeit und Wiederverwendung einer Klasse angesehen werden: Bei einem hohen Wert erfordern Änderungen an Basisklassen mit einiger Wahrscheinlichkeit auch Folgeänderungen in Subklassen. Generell dürfen Basisklassen zwar mehrere Subklassen besitzen und damit einen höheren Wert für NOC: Je tiefer man sich in der Ableitungshierarchie befindet, desto unwahrscheinlicher ist es aber, dass von den Spezialisierungen wiederum viele weitere Spezialisierungen existieren. Ähnlich wie bei DIT liegen sinnvolle Werte für NOC im Bereich von fünf bis acht, häufiger eher im Bereich von zwei bis vier. Für viele Klassen, etwa solche, die `final` sind, ist der Wert für NOC 0.
  - **Coupling Between Objects (CBO)**
    Die Metrik CBO bestimmt die Anzahl der Klassen, mit denen die betrachtete Klasse gekoppelt ist, d. h. auf deren Methoden oder Attribute sie zugreift. Zudem gehen auch alle Zugriffe und Aufrufe von anderen Klassen in die be-

trachtete Klasse mit in die Kennzahl ein. Ein hoher Wert zeigt eine starke Kopplung an, was wahrscheinlich zu einem erhöhten Aufwand bei Änderungen und nachfolgenden Tests führen wird. Der ermittelte Wert sollte möglichst kleiner als fünf bis zehn sein. Selten sind Werte bis 20 in Ordnung.

- **Number Of Methods (NOM)**
  Die Metrik NOM beschreibt die Anzahl in einer Klasse definierter Methoden. Ein hoher Wert deutet darauf hin, dass die Klasse für zu viele Dinge verantwortlich ist. Hier gelten die gleichen Schlussfolgerungen wie bei der Metrik CBO. Bevorzugt sollte der Wert im Bereich von 10 bis 30 liegen. Bei komplexeren Klassen können ausnahmsweise auch höhere Werte akzeptiert werden.

Durch Überprüfen der obigen Kennzahlen kann man zwar einige potenzielle Fehlerquellen identifizieren, eine korrekte Realisierung der gewünschten Funktionalität lässt sich allein basierend auf Metriken allerdings nicht sicherstellen. Zudem existieren weitere für die Qualität entscheidende Bewertungskriterien, die sich jedoch kaum durch Tools prüfen lassen. Dies sind unter anderem die folgenden:

- **Lesbarkeit / Übersichtlichkeit**
  Normalerweise geht eine gute Lesbarkeit mit verbesserten Werten anderer Bewertungskriterien einher und stellt daher eine der wichtigsten Messgrößen dar. Die Lesbarkeit kann man als erfahrener Entwickler für wenige Klassen relativ gut durch stichprobenartige Blicke in den Sourcecode beurteilen. Bei umfangreicheren Projekten ist eine derartige Prüfung weder vollumfänglich noch sinnvoll möglich. Problematisch ist zudem, dass sich die Lesbarkeit nur schwierig formalisieren lässt und dadurch auch nicht durch Tools geprüft werden kann. Am einfachsten lässt sich dieses Kriterium erfüllen, wenn vor Projektbeginn ein Regelwerk ähnlich zu den vorgestellten Coding Conventions vereinbart und beim Programmieren auf dessen Einhaltung geachtet wird.
- **Abstraktionsgrad**
  Die Wahl geeigneter Abstraktionsgrade in Programmkomponenten ist eine durch Menschen intuitiv erfassbare Messgröße, die wiederum lediglich stichprobenartig überprüfbar ist. Deren Einhaltung erkennt man in der Regel daran, dass öffentliche Methoden gut lesbar und verständlich sind, da sie auf einem hohen Abstraktionsniveau formuliert sind und sich aus Aufrufen von anderen Methoden geringerer Sichtbarkeit (`private`, Package-private und `protected`) zusammensetzen. Diese (Hilfs-)Methoden bilden in der Regel die Bausteine der öffentlichen Methoden. Wichtig ist es, dass Klassen ihre Methoden auf dem niedrigstmöglichen Sichtbarkeitslevel bereitstellen und somit das API, d. h. die Menge der öffentlichen Methoden, möglichst nur noch aus verhaltensdefinierenden Business-Methoden besteht und keinesfalls aus nur in der Klasse selbst benötigten Hilfsmethoden. Befolgt man das Gesagte, so führt dies meistens zu einer klaren Aufrufhierarchie von `public` nach `private`. ***Erfolgen dann durch private Methoden noch Aufrufe an öffentliche Methoden, so sollte man diese genau prüfen, da dies ein Zeichen schlech-***

ten Designs und falsch gewählter Abstraktionsebenen sein kann. Es gibt jedoch verschiedene Anwendungsfälle, in denen in privaten Methoden bestimmte Funktionalität aus dem API benötigt wird. Dann kann ein Aufruf öffentlicher Methoden aus einer privaten Methode sinnvoll und unkritisch sein, da insbesondere das öffentliche API in der Regel stabil bleibt. Wichtig dabei ist allerdings, dass man ein Aufrufchaos vermeidet und das lässt sich am einfachsten über die Einhaltung der oben genannten strengen Hierarchie erreichen. Eine Möglichkeit zur Vermeidung eines Aufrufs einer öffentlichen Methode aus einer privaten besteht darin, eine zusätzliche private Verarbeitungsmethode mit der gewünschten Funktionalität zu erzeugen, die dann sowohl von öffentlichen Methoden als auch privaten Methoden aufgerufen werden kann. Auch kann man spezielle Methoden in (package-private) eigenständige Klassen auslagern, die dann aus der ursprünglichen Klasse genutzt werden.

- **Fehlersicherheit und -toleranz**
  Bei der Fehlersicherheit und -toleranz lässt sich das extern beobachtete Verhalten und die tatsächlichen internen Abläufen unterscheiden. Werden Fehler nicht nach außen sichtbar, so bedeutet das zunächst wenig. Es ist durchaus möglich, dass diese »verschluckt« werden: Man spricht dabei auch von ***Fehlermaskierung***. Viel wichtiger ist demnach die Reaktion auf Fehler im Programm selbst, die man nicht von außen sieht. Durch eine Analyse des Sourcecodes kann man zumindest feststellen, ob Parameter in öffentlichen Methoden oder an Modulgrenzen geprüft werden und angemessen auf Exceptions oder Fehlersituationen reagiert wird, etwa um Ressourcen auch im Fehlerfall wieder freizugeben.

19.4.2 Sourcecode-Prüfung im Build-Prozess

Wie bereits erwähnt, ist es sinnvoll, die Qualität des Sourcecodes regelmäßig zu analysieren und dabei die Einhaltung gewisser Standards zu beachten. In einem ersten Schritt kann man auf die in Eclipse integrierte Sourcecode-Prüfung zurückgreifen. Umfangreichere Tests sollten mit Tools zur statischen Analyse wie Checkstyle, PMD und SonarQube erfolgen. Die ersten beiden werden in dieser Auflage nur kurz beschrieben, da SonarQube am meisten Komfort bietet und eine Vielzahl an Prüfungen bündelt.

Das Tool Checkstyle

Das Tool Checkstyle hilft dabei, die Einhaltung von Formatierungsvorgaben und Coding Conventions zu überprüfen. Im Eclipse Marketplace steht ein Eclipse-Plugin frei zur Verfügung.

Hilfreich beim Einsatz von Checkstyle ist, dass sich die gewünschten Codierungsregeln feingranular konfigurieren lassen, wodurch nahezu alle möglichen Varianten von Coding Conventions realisiert werden können. Zum Einstieg existiert ein vordefinierter Regelsatz für die Coding Conventions von Sun. Praktischerweise kann Checkstyle auch

Reports im HTML-Format erzeugen. Weiterführende Informationen sind online unter `http://checkstyle.sourceforge.net/` verfügbar.

Integration in den Build-Prozess Es ist sinnvoll, die Prüfung des Sourcecodes als Bestandteil des Build-Prozesses auszuführen. Checkstyle lässt sich bequem in den Gradle-Build integrieren, indem man das passende Plugin einbindet. Empfehlenswert ist es, die Prüfung bei festgestellten Verstößen nicht abzubrechen, sondern gegebenenfalls noch weitere Prüftools auszuführen (und dazu den Parameter `ignoreFailures` auf `true` zu setzen):

```
apply plugin: 'checkstyle'

checkstyle {
    toolVersion '8.36.1'
    configFile file("config/checkstyle/checkstyle.xml")
}

tasks.withType(Checkstyle) {
    ignoreFailures = true
    reports {
        xml.enabled false
        html.enabled true
    }
}
```

Die Konfiguration der Regeln in Form der Datei `checkstyle.xml` sucht Checkstyle – wie angegeben – in einem Verzeichnis, z. B. `config/checkstyle`. Die Prüfung schlägt fehl, wenn dieser Ordner nicht existiert. Eine minimale Konfiguration als Ausgangsbasis kann etwa wie folgt aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="Checker">
    <module name="TreeWalker">
        <module name="AvoidStarImport"/>
        <module name="ConstantName"/>
        <module name="EmptyBlock"/>
    </module>
</module>
```

Die Checkstyle-Seite `http://checkstyle.sourceforge.net/config.html` liefert weitere Infos. Eine ausführliche Prüfung, die von Google verwendet wird, findet man unter `https://code.google.com/p/google-api-java-client/source/browse/checkstyle.xml?repo=samples`.

HTML-Report erzeugen Für eine Untersuchung führt man Folgendes aus, was einen Report im Verzeichnis `build/reports/checkstyle` erzeugt:

```
gradle checkstyleMain
```

Das Ergebnis sieht grob wie folgt aus:

CheckStyle Audit

Designed for use with CheckStyle and Ant.

Summary	
Files	**Errors**
645	59

Files	
Name	**Errors**
/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-quelltext/src/main/java/ch13_i18n/utils/MultipleDateFormatParsingExample1.java	4
/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-quelltext/src/main/java/ch13_i18n/utils/MultipleDateFormatParsingExample1Jdk8.java	4
/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-quelltext/src/main/java/ch13_i18n/utils/MultipleDateFormatParsingExample2.java	4
/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-quelltext/src/main/java/ch99_swing_gui/multithreading_and_swing/SwingWorkerDemo.java	3
/Users/michaeli/Desktop/java-profi-Auflage5_aktuell/JP5-quelltext/src/main/java/ch06_collections/traps/ArrayCloneExample.java	2

Abbildung 19-3 *Checkstyle-Report*

Java-14/15-Kompatibilität Checkstyle kommt in der im September 2020 aktuellen Version 8.36.1 ohne weitere Konfiguration sowohl mit Java 14 als auch Java 15 zurecht. Allerdings werden einige Java-15-Features bei der Analyse einfach ignoriert, etwa Sealed Types.

Das Tool PMD

PMD ist ein Tool zur statischen Analyse und untersucht den Sourcecode. PMD lässt sich in diverse IDEs integrieren. Im Eclipse Marketplace steht ein Eclipse-Plugin frei zur Verfügung.

Integration in den Build-Prozess PMD kann man sich mit wenigen Zeilen in den Gradle-Build einbinden. Das hatte ich bereits in Abschnitt 2.7.2 kurz beschrieben. Hier wiederhole ich die wesentlichen Angaben, insbesondere auch diejenigen, um HTML oder XML als Ausgabeformate zu wählen:

```
apply plugin: 'pmd'

pmd {
    consoleOutput = true
    toolVersion = "6.27.0"
    rulePriority = 5
    ruleSets = ["category/java/errorprone.xml", "category/java/bestpractices.xml
          "]
}

tasks.withType(Pmd) {
    ignoreFailures = true

    reports {
        xml.enabled = false
        html.enabled = true
    }
}
```

HTML-Report erzeugen Für eine Untersuchung führt man Folgendes aus, was einen Report im Verzeichnis `build/reports/pmd` erzeugt:

```
gradle pmdMain
```

Java-14/15-Kompatibilität PMD kommt in der im September 2020 aktuellen Version 6.27 sowohl mit Java 14 als auch Java 15 zurecht. Allerdings muss man dazu gewisse Konfigurationsarbeiten und Parametrierungen vornehmen.

Damit PMD mit Preview-Features aus Java 15 umgehen kann, müssen zwei Dinge konfiguriert bzw. parametriert werden:

1. Umgebungsvariable `PMD_JAVA_OPTS=--enable-preview`
2. Übergabe der neuen Sprachversion `15-preview`

Laut PMD-Webseite (`https://pmd.github.io/`) kann dies auf der Konsole wie folgt geschehen:

```
export PMD_JAVA_OPTS=--enable-preview
./run.sh pmd -language java -version 15-preview ...
```

Dies gilt es, für Gradle bzw. Maven abzubilden.

Besonderheiten beim Gradle-Plugin Zunächst müssen Sie die Umgebungsvariable wie benötigt setzen:

```
tasks.withType(Pmd) {

    doFirst {
       System.setProperty "PMD_JAVA_OPTS", "--enable-preview"
    }

    ignoreFailures = true // execute all the other checkers too

    reports {
        xml.enabled = false
        html.enabled = true
    }
}
```

Nun wollen wir die Sprachversion setzen. Zwar ist es mit Gradle sehr einfach auf das aktuellste PMD 6.27 zu verweisen, jedoch lässt sich die Sprachversion nicht setzen, da diese normalerweise von Gradle automatisch gemäß dem im Build-Lauf angegebenen JDK gesetzt wird. Hier benötigt es aber mit "15-preview" eine spezielle Angabe. Dies lässt sich leider nicht mit einer Konfiguration über `targetJdk` wie folgt erledigen:

```
pmd {
    targetJdk = "15-preview" // Could not determine targetjdk from '15-preview'.
}
```

Führen wir einen Build- und Prüflauf aus (`gradle pmdMain`), so erhalten wir jedoch die als Kommentar angedeutete Fehlermeldung. Hier macht sich unschön bemerkbar, dass man das `targetJdk` mit Gradle nicht auf beliebige Werte, sondern nur auf die vordefinierten und bis JDK 7 bereitgestellten Werte setzen kann.[6] Wie erwähnt, wird das in der Regel nicht benötigt und es greift die Automatik. Jedoch benötigt PMD hier nicht die Angabe von `15`, sondern von `15-preview`, weshalb PMD dann einige Verarbeitungsfehler produziert.[7] Leider ist nach Auftreten des ersten Preview-Features nicht immer eine saubere Fertigstellung der Prüfung gegeben, wodurch die restliche Sourcecode-Basis leider nicht verlässlich geprüft werden kann.

Sobald dieses Problem im Gradle-Plugin gelöst ist, können Sie PMD wieder für sämtliche Prüfungen verwenden. Ein Update finden Sie dann auf der Webseite zu diesem Buch.

Besonderheiten bei Maven Auch Maven besitzt ein Plugin für PMD, das in seiner Version 3.13.0 jedoch auf PMD 6.23 konfiguriert ist. Um Java 15 prüfen zu können, muss die PMD-Version über neue Dependency-Einträge auf PMD 6.27 geändert werden:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-pmd-plugin</artifactId>
    <version>3.13.0</version>
    <configuration>
        <failOnViolation>true</failOnViolation>
        <printFailingErrors>true</printFailingErrors>
        <targetJdk>15-preview</targetJdk>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>net.sourceforge.pmd</groupId>
            <artifactId>pmd-core</artifactId>
            <version>6.27.0</version>
        </dependency>
        <dependency>
            <groupId>net.sourceforge.pmd</groupId>
            <artifactId>pmd-java</artifactId>
            <version>6.27.0</version>
        </dependency>
    </dependencies>
</plugin>
```

[6] Diesbezüglich stehe ich bereits in Kontakt mit den Gradle-Entwickler und es gibt dazu einen Bug-Eintrag.

[7] Hier sieht man wieder einmal die Problematik des schnellen Releasezyklus und der dadurch nicht immer ganz problemlosen Migration.

Außerdem muss die Umgebungsvariable passend wie folgt gesetzt werden:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>properties-maven-plugin</artifactId>
    <version>1.0.0</version>
    <executions>
        <execution>
            <phase>initialize</phase>
            <goals>
                <goal>set-system-properties</goal>
            </goals>
            <configuration>
                <properties>
                    <property>
                        <name>PMD_JAVA_OPTS</name>
                        <value>--enable-preview</value>
                    </property>
                </properties>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Durch Aufruf von

```
mvn verify
```

entsteht dann der Report `target/site/pmd.html`.

Hinweis: Kompatibilitätsprobleme durch häufige Releasewechsel

Eines sei noch angemerkt: Zwar ist Maven oftmals ein wenig geschwätziger, jedoch haben gerade die letzten Releasewechsel bei Java gezeigt, dass Maven gegenüber Gradle deutlich problemloser die Migrationen unterstützt hat.

Das Tool SonarQube

Die bisher beschriebenen Tools adressierten jeweils einen spezifischen Aspekt der Sourcecode-Qualität – wobei eine gewisse thematische Streuung zu finden ist. Das Tool Checkstyle prüft nicht nur die Einhaltung von Coding Conventions, sondern zudem auch gewisse Designaspekte oder Programmierfehler. Bei Letzteren hat vor allem PMD seine Stärken.

Alle Tools müssen bzw. sollten aber separat und geeignet konfiguriert werden, was glücklicherweise durch Nutzung von Gradle erleichtert wird. Wäre es nicht noch angenehmer, wenn sich ein Tool um den gesamten Qualitätssicherungsaspekt kümmern und einen konsistenten Bericht aufbereiten würde?

Ein solches Tool existiert mit SonarQube, ehemals nur Sonar genannt. SonarQube bietet eine ***Plattform für die statische Analyse*** und integriert dazu verschiedene Prüfungen und deren Befunde. Dabei hilft eine komfortable Weboberfläche. In Abbildung 19-4 sehen wir die Analyseergebnisse der Sourcen zu diesem Buch, wobei auch diejenigen der Bad Smells und anderer Fallstricke mit enthalten sind.

Abbildung 19-4 *SonarQube-Dashboard der Sourcen zu diesem Buch*

Installation und Start SonarQube steht unter `http://www.sonarqube.org/` frei als ZIP zum Download bereit – im Juli 2020 ist Version 8.4.2 aktuell. Diese unterstützt Java 14. Das Archiv kann in ein beliebiges Verzeichnis entpackt werden.

SonarQube besteht aus einer Server- und einem Webclient-Teil. Der Server-Teil befindet sich in einem Unterordner für das jeweilige Betriebssystem im `bin`-Ordner. Dort existiert ein Kommandozeilenskript, mit dem man den Server per `sonar start` (Windows) bzw. `./sonar.sh start` (MAC) starten kann. Nach Abschluss des Starts lässt sich dieser Server über eine Weboberfläche ansprechen. Standardmäßig erreicht man diese über `http://localhost:9000` (mit Default-Credentials `admin`/`admin`). Danach kann man sich die generierten Reports – wie oben angedeutet – mit einem Webbrowser ansehen und verschiedene Konfigurationsarbeiten vornehmen.

SonarQube und Gradle SonarQube lässt sich recht einfach in einen Gradle-Build einbinden. Dazu fügt man am Anfang der Datei `build.gradle` folgende Zeilen hinzu:

```
plugins {
    id "org.sonarqube" version "3.0"
}
```

Außerdem muss noch die Konfiguration des Projekts und des SonarQube-Servers erfolgen. Dazu erzeugt man eine Datei namens `gradle.properties` im selben Verzeichnis wie die Build-Datei mit folgendem Inhalt:

```
systemProp.sonar.host.url=http://localhost:9000

systemProp.sonar.projectKey=Java-Profi-5
systemProp.sonar.projectName=Java-Profi-5
```

Nach diesen Konfigurationsarbeiten können wir einen Prüflauf mit

```
gradle sonar
```

starten. Das erzeugt auch automatisch ein korrespondierendes Projekt im SonarQube-Server. Auf dessen Webseite können wir dann das Dashboard anschauen und später noch ein paar Konfigurationsänderungen vornehmen.

Das Eclipse-Plugin SonarLint Wenn wir Projekte direkt innerhalb von Eclipse analysieren wollen, benötigen wir das im Eclipse Marketplace angebotene Tool SonarLint – im September 2020 in Version 5.3. Dieses kann entweder stand-alone für Prüfungen innerhalb der IDE ohne expliziten SonarQube-Server oder aber in Kombination mit einem SonarQube-Server betrieben werden. Ersteres ist für schnelle Untersuchungen gut, Letzteres hilft, die Entwicklung und positive oder negative Tendenzen aufdecken zu können.

Nach der Installation sollten wir die Verbindung zum SonarQube-Server einrichten (`Connect to a server` und die URL `http://localhost:9000` nutzen. Für erste Experimente kann man dafür die Variante mit Username und Passwort und den Default-Credentials (`admin`/`admin`) verwenden.). Dann konfigurieren wir die Verbindung zu den Projekten im SonarQube-Server über das Kontextmenü eines Projekts SONARLINT –> BIND TO A SONARQUBE PROJECT. Einen Analyselauf startet man über das Kontextmenü SONARLINT –> ANALYZE CHANGED FILES oder aber über das Tastaturkürzel CTRL + ALT + Q. Ein Beispiel für eine generierte Menge an Meldungen zeigt Abbildung 19-5.

Problems | Search | Console | Progress | Gradle Executions | History | Coverage | Gradle Tasks | SonarLint Report | SonarQube Servers

Filter matched 100 of 2386 items

Resource	Date	Description
BadDefaultExample.java	3 hours ago	End this switch case with an unconditional break, return or throw statement.
MinValueTest.java	53 minutes ago	Add some tests to this class.
ResourceManagerExample.java	3 hours ago	"menuBar" is the name of a field in "Frame".
ResourceManagerV2Example.java	3 hours ago	"menuBar" is the name of a field in "Frame".
AssertExample.java	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
AssertExampleWithMethods.java	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
AutoBoxingAndMethodSignatur...	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
AutoBoxingCacheExample.java	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
AutoBoxingCacheExample.java	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
BadCaseExample.java	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
BinaryOctalHexadecimalExampl...	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
BinaryOctalHexadecimalExampl...	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
BinaryOctalHexadecimalExampl...	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.
DateAPIProblems.java	3 hours ago	Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.

857 files of project Java-Profi-4 (at 26/08/2017 15:14)

Abbildung 19-5 *Anzeige der SonarLint-Analyseergebnisse in Eclipse*

Empfehlenswerte Anpassungen in der Konfiguration Bei der Messung zu diesem Buch habe ich ein paar Regeln leicht angepasst. Dazu muss man ein eigenes Quality Profile anlegen und dieses dem eigenen Projekt zuweisen. Danach bietet es sich an, ein paar Anpassungen am Regelwerk vornzuehmen – unpraktischerweise heißen die Regeln anders als die Warnmeldungen. Nachfolgend gebe ich beides an, um die Orientierung und Konfiguration zu erleichtern.

Folgende Anpassungen wurden vorgenommen:

- **Meldung:** `Replace this use of System.out or System.err by a logger.`
  Regel: `Standard outputs should not be used directly to log anything`
  Änderung: Major => Minor
- **Meldung:** `Use try-with-resources or close this "ServerSocket" in a "finally" clause.`
  Regel: `Resources should be closed`
  Änderung: Blocker => Major
- **Meldung:** `Change this "try" to a try-with-resources.`
  Regel: `Try-with-resources should be used`
  Änderung: Critical => Major
- **Meldung:** `Add a nested comment explaining why this method is empty, throw an UnsupportedOperationException or complete the implementation.`
  Regel: `Methods should not be empty`
  Änderung: Critical => Minor, ist auf keinen Fall ein kritischer Fehler
- **Meldung:** `Rename this constant name to match the regular expression ?^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$.`
  Regel: `Constant names should comply with a naming convention`
  Änderung: Critical => Minor, ist auf keinen Fall ein kritischer Fehler

Ein paar bewusst verursachte Fehler, wie eine Endlosschleife zur Demonstration, habe ich von Hand zusätzlich als False Positive gekennzeichnet.

Historisierung Bei der Analyse der Qualität interessiert neben dem Ist-Stand vor allem der zeitliche Verlauf, also an welchen Stellen hat sich die Qualität verbessert und wo eventuell sogar verschlechtert und wie ist der Trend. Dazu kann man ein Projekt einfach mehrmals im Verlaufe eines Monats oder eines Jahres analysieren, z. B. auch idealerweise im Nightly Build. Für die zuvor beschriebenen Tools müsste man dann selbst Statistik führen. SonarQube hilft dabei, Trends zu erkennen und Vergleiche mit vorherigen Versionen anzustellen. Dazu bietet SonarQube verschiedene Dashboards an, wo sich die Daten leicht miteinander vergleichen und insbesondere Trends erkennen lassen.

Java-14/15-Kompatibilität SonarQube und auch SonarLint kommen mit den im September 2020 aktuellen Versionen 8.4.2 bzw. 5.3 ohne weitere Konfiguration sowohl mit Java 14 als auch Java 15 zurecht. Allerdings werden einige Java-15-Features bei der Analyse einfach ignoriert, etwa Sealed Types. Vermutlich mit den nächsten Update-Versionen sollte Java 15 vollumfänglich unterstützt werden.

20 Unit Tests

Dieses Kapitel stellt das Themengebiet Unit Tests vor. Abschnitt 20.1 gibt einen Überblick über das Testen im Allgemeinen und mögliche Auswirkungen auf die Qualität. In Abschnitt 20.2 beschäftigen wir uns mit dem Erstellen von Testfällen. Nach diesem Einstieg stelle ich in Abschnitt 20.3 einige Erweiterungen in Form spezieller Assertions vor und gehe in Abschnitt 20.4 auch auf sogenannte parametrierte Tests ein, deren Erstellung mit JUnit 5 wesentlich komfortabler und effizienter möglich ist als mit dem Vorgänger JUnit 4.

In der Praxis sind häufig kompliziertere Objektgebilde zu testen, wofür die zuvor dargestellten Basistechniken nicht ausreichen. Deshalb beschreibt Abschnitt 20.5 fortgeschrittenere Möglichkeiten zum Lösen von Abhängigkeiten, etwa die Techniken EXTRACT AND OVERRIDE sowie Stellvertreterobjekte, sogenannte Test-Doubles, vor allem Stubs und Mocks. Danach betrachten wir in Abschnitt 20.6 einige Fallstricke beim Formulieren von Unit Tests, sogenannte Test Smells. Im Anschluss daran beleuchtet Abschnitt 20.7 verschiedene nützliche Tools, deren Einsatz das Unit-Testen erleichtern kann. Zwar ist JUnit 5 schon vor einigen Jahren veröffentlicht worden und bereits gut in der Entwicklergemeinde angekommen (insbesondere begünstigt auch durch die Rückwärtskompatibilität zu JUnit 4), allerdings existiert wohl noch ein breite Basis an Unit Tests, die weiterhin JUnit 4 nutzen. Schließlich geht Abschnitt 20.8 auf Unterschiede zwischen JUnit 4 und JUnit 5 ein und gibt einige Hinweise für Migrationen.

20.1 Testen im Überblick

Im nachfolgenden einführenden Abschnitt gebe ich einen Überblick zum Testen, beschreibe dazu kurz unterschiedliche Arten von Tests sowie deren beteiligte Akteure. Dabei erläutere ich auch die Begriffe Blackbox- und Whitebox-Tests und konkretisiere das Unit-Testen. Zudem werden die Zuständigkeiten beim Testen vorgestellt und wir widmen uns der Thematik des Testens und der Qualität.

20.1.1 Was versteht man unter Testen?

Unter ***Testen*** versteht man den Vorgang, das tatsächliche Verhalten eines Programms oder eines Teils davon (***Ist***) mit dem geforderten Verhalten (***Soll***) zu vergleichen. Demnach entspricht Testen nicht dem einmaligen Start eines Programms mit ein paar will-

kürlichen Bedienhandlungen. Es geht vielmehr darum, das Verhalten der Software unter verschiedenen Bedingungen zu prüfen. Dabei muss man fast ein wenig »bösartig« agieren, um versteckte Probleme aufdecken oder Fehlverhalten provozieren zu können. Dazu gehört beispielsweise eine bewusste Fehlbedienung, etwa die Eingabe ungültiger Werte sowie von Extrem- oder Randwerten. Unter einem ***Fehler***[1] verstehen wir, wenn es beim Testen zu Abweichungen zwischen den produzierten Ergebnissen und den gewünschten Resultaten kommt, sich das geprüfte Softwaresystem also anders als spezifiziert verhält oder gar abstürzt – weniger formal: wenn die gelieferten Ergebnisse nicht den Bedürfnissen oder Anforderungen der Benutzer entsprechen. ***Testen dient somit der Sicherstellung von Softwarequalität***. Je besser die Anforderungen bekannt und spezifiziert sind, desto einfacher können basierend darauf Tests definiert werden, weil sich dann produzierte Ergebnisse leichter mit den gelieferten vergleichen lassen. Damit eine Aussage über die Qualität oder das Testergebnis möglich wird, muss der Vorgang des Testens systematisch erfolgen. Idealerweise werden in einem Testdokument gegebene Rahmenbedingungen, etwa zu nutzende Eingabewerte, auszuführende Aktionen und erwartete Ergebnisse, festgehalten.

Hinweis: Testen erfordert Kenntnisse der Anforderungen

Um eine qualitative und quantitative Aussage über die Qualität und umgesetzte Funktionalität zu ermöglichen, hilft es, die Erwartungen an das entstehende Softwaresystem so weit zu spezifizieren, dass daraus Testfälle erstellt werden können. Außerdem ist es hilfreich, sogenannte Non Functional Requirements (NFR) = nicht funktionale Anforderungen, etwa »Antwortzeiten kleiner 2 Sekunden«, als zu erreichende Ziele festzulegen und diese auch zu testen.

Diese einleitenden Gedanken zum Testen möchte ich mit einem Zitat von Edsger W. Dijkstra aus »The Humble Programmer, ACM Turing Lecture, 1972« [15][2] beschließen: »Durch Testen kann man stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen.«

20.1.2 Testarten im Überblick

Um ein wenig mehr Klarheit in den weit gefassten Begriff des Testens zu bringen, ist es sinnvoll, die verschiedenen Arten von Tests und die Akteure kurz vorzustellen.

Begrifflichkeiten beim Testen

Für Entwickler sind vor allem ***Tests einzelner Programmbausteine*** in Form von ***Unit Tests*** oder ***Komponententests*** interessant sowie das Prüfen auf ein fehlerfreies Zusammenwirken von Systemkomponenten, ***Integrationstest*** genannt. Nach erfolgreichen

[1] Laut ISTQB-Glossar (International Software Testing Qualifications Board) entspricht dies der Fehlerwirkung. Nachfolgend unterscheide ich aber der Einfachheit halber nicht so genau.

[2] `https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html`

Tests in der Entwicklung wird die Programmfunktionalität durch sogenannte ***Systemtests*** geprüft. Oftmals wird dazu die Software – oder je nach Umfang auch nur ein funktional in sich abgeschlossener Teil – an eine dedizierte Testabteilung mit Testern bzw. Testingenieuren übergeben.[3] Beim ***Applikationstest*** wird das Programm unter verschiedenen Aspekten aus Anwendersicht (auch ***Domain*** oder fachliche Ebene genannt) untersucht. Zuletzt erfolgt gewöhnlich ein ***Abnahmetest*** oder ***Akzeptanztest***, der in etwa eine Wiederholung des Applikationstests durch den oder unter Mitwirkung des Kunden ist. Abbildung 20-1 visualisiert die zuvor beschriebene Testhierarchie und zeigt dabei sowohl einige Eigenschaften der jeweiligen Tests als auch die ausführenden Personen.

Abbildung 20-1 *Übersicht über die Arten von Tests*

Aufgrund der gerade identifizierten Eigenschaften ergibt sich fast automatisch die sogenannte ***Testpyramide***. Diese repräsentiert und visualisiert einen anzustrebenden Zustand bei der Art und Anzahl von Tests: Demnach sollte die Funktionalität der Basisbausteine eines Systems mit einer Vielzahl an Unit Tests geprüft werden, die sich zudem schnell und automatisiert ausführen lassen. Eine geringere, aber angemessene Zahl von Integrationstests stellt das Zusammenspiel der Systembestandteile sicher. Schließlich wird das Gesamtsystem durch aufwendigere und oftmals manuell auszuführende System- oder GUI-Tests geprüft. Dies ist in Abbildung 20-2 dargestellt, die ich in Anlehnung an `http://agilejazz.blogspot.ch/p/the-devops-handbook-summary-1-of-4.html` erstellt habe.

[3] Agile Vorgehensmodelle, wie etwa Scrum, sehen das Testen als Aufgabe des Entwicklungsteams an, dem auch Tester angehören.

Abbildung 20-2 *Ideale und weniger ideale Testpyramide*

Blackbox- vs. Whitebox-Tests

Beobachten und testen wir die Interaktion mit dem System von außen, so spricht man von einem sogenannten »***Blackbox-Test***« – die »Innereien« bleiben verborgen. Dieses Vorgehen kann allerdings Fehler verschleiern: Das geschieht etwa, wenn zwei Programmteile fehlerhaft sind und sich die Effekte gegenseitig aufheben.[4] Man spricht auch von ***Fehlermaskierung***. Um so etwas möglichst ausschließen und genauere Aussagen über die korrekte Funktion der einzelnen Bestandteile eines Softwaresystems machen zu können, sind weitere Formen des Testens sinnvoll, die auch das Innenleben der Software berücksichtigen. Man spricht dann von einem sogenannten »***Whitebox-Test***«. Abbildung 20-3 stellt dies schematisch dar.

Abbildung 20-3 *Blackbox- und Whitebox-Test*

Bitte bedenken Sie, dass gewöhnlich beide Arten des Testens benötigt werden, um eine gute Qualität zu erreichen. Auf Ebene von Applikationstests nutzt man eher das Blackbox-Testen. Möchte man mehr das Innenleben prüfen, so helfen Whitebox-Tests, insbesondere in Form feingranularer Unit Tests. Aber selbst auf Komponenten- oder Klassenebene kann man wieder folgendermaßen differenzieren: Entweder testet man

[4] Das klingt zunächst abwegig, tritt aber durchaus auf – aus der Natur kennt man es: Wenn zwei Wellen gegeneinander verschoben sind, können sich Wellentäler und -berge auslöschen.

das öffentliche API auf gelieferte Ergebnisse oder man schaut auf Zustandsänderungen im Inneren.

Konkretisierung des Begriffs Unit-Testen

Dem Namen entsprechend, wird beim Unit-Testen eine kleine Einheit (Unit) betrachtet. Ziel dabei ist es, einen einzelnen Baustein des Programms separat und isoliert vom Rest des Gesamtsystems auf korrekte Funktionalität zu prüfen, also möglichst ohne Effekte und Auswirkungen durch das Zusammenspiel mit anderen Komponenten.

Beim Unit-Testen werden die Testfälle als Programme implementiert. Das erlaubt es Entwicklern, ihre Stärken in der Programmierung zu nutzen. Bei moderner Softwareentwicklung sollte das Testen ein elementarer Teil der Entwicklungsarbeit sein, gemäß dem Motto »Code a Little, Test a Little«. Beide Tätigkeiten werden eng miteinander verknüpft und Testfälle entstehen in Form von Unit Tests parallel zum Programmieren. So verfährt man auch bei den Programmierparadigmen ***Extreme Programming*** (XP) und ***Test-Driven Development*** (TDD). TDD propagiert sogar, zunächst einige Testfälle als Unit Tests zu erstellen, bevor der Applikationscode entwickelt wird.

Zusammenspiel von Unit Tests und Integrationstests

Unit Tests können die Aufwände für Integrationstests senken, da viele, idealerweise alle Komponenten für sich alleine getestet sind. Daher müssen die Integrationstests hauptsächlich noch die Schnittstellen und die Kommunikation sowie die Benutzung gemeinsamer Ressourcen der zu integrierenden Komponenten prüfen. Das sollte man allerdings nicht unterschätzen, denn gerade bei der Interaktion zwischen verschiedenen Komponenten treten des Öfteren noch unerwartet Fehler auf, etwa dadurch, dass zeitliche Randbedingungen oder andere Implementierungsdetails zum Tragen kommen. ***Da die Überprüfung solcher Interaktionsprobleme per Definition nicht Bestandteil von Unit Tests ist, sollten zusätzlich immer Integrationstests erfolgen***.

Ein Fehlschlagen von Integrationstests kann aber nicht nur durch das fehlerhafte Zusammenspiel von Komponenten, sondern auch durch verbliebene Implementierungsprobleme in einzelnen Programmteilen verursacht werden, für die keine oder nicht geeignete Unit Tests existieren. In dem Fall sollte man die Unit Tests passend erweitern.

Integrationstests einer Komponente sind vielfach erst dann sinnvoll, wenn die jeweiligen Einzelbestandteile eine ausreichend hohe Qualität besitzen.

20.1.3 Zuständigkeiten beim Testen

Immer mal wieder hört man die Frage, ob Entwickler ihre Programme selbst testen sollten.[5] Als Gegenargument wird eine mögliche ***Betriebsblindheit*** oder Voreingenommenheit des Erstellers ins Feld geführt. Obwohl diese Bedenken grundsätzlich richtig sind,

[5] In manchen regulierten Domänen wie z. B. bei Eisenbahnsystemen ist genau definiert, wann der Entwickler selbst testen darf bzw. wann dies ein Tester machen muss.

ist es sinnvoll, wenn Entwickler die von ihnen programmierte Funktionalität selbst prüfen. Dabei kommt es jedoch auf die richtige Form an: Applikationstests sollten von dedizierten Testern durchgeführt werden (siehe folgenden Meinungskasten). Zuvor sollten Entwickler aber mit Unit Tests sowie Integrationstests ihrer Komponenten eine gute Softwarequalität absichern. Unprofessionell ist dagegen, das Testen als lästiges Übel anzusehen und nach dem von Jon Bentley [3] als *falsch* erkannten Motto zu verfahren: »Schreibe deinen Code, [...] überlasse die Fehler der Qualitätskontrolle oder Testabteilung.« Vielmehr sollte man als Entwickler das ***Testen als Chance verstehen, qualitativ hochwertigen Sourcecode zu schreiben***. Bedenken Sie, dass Sie durch eine gute Qualität Ihrer Implementierungen viel weniger Zeit mit Fehlersuche und anschließender Behebung verbringen müssen. Tatsächlich werden Sie durch entwicklungsbegleitendes Testen mehr programmieren können und deutlich seltener über Problemen brüten oder in Stress geraten, wenn Sie einen Fehler finden und fixen sollen. Das gilt allerdings nur, falls Sie die richtigen und wichtigen Dinge testen, also komplexe Abläufe und Berechnungen, nicht aber Trivialitäten wie z. B. Zugriffsmethoden.

Zuständigkeiten bei Applikationstests

Wenn Projektleiter von den Entwicklern fordern, umfangreiche Applikationstests, also Tests aus Anwendersicht und auf fachlicher Ebene, durchzuführen, wird das mitunter als unangenehm und langwierig empfunden, weil zur Ausführung von Applikationstests oftmals spezielle, aufwendige Konfigurationen erforderlich sind, um gewisse Rahmenbedingungen zu schaffen. Weiterhin sind komplexe Testfälle, die möglicherweise auch einiges an Fachwissen benötigen, abzuprüfen. Es entsteht schnell eine gewisse Abneigung gegen diese Art des Testens, wenn das erforderliche Know-how oder die Zeit zur Einarbeitung fehlt. Das trifft für Entwickler in der Regel zu, da deren Hauptfokus auf dem Entwickeln und dem entwicklungsbegleitenden Unit-Testen liegt.

Meinung: Applikationstests durch dedizierte Tester bzw. Testabteilung

Während agile Vorgehensmodelle propagieren, dass auch System- und Applikationstests vom Entwicklungsteam ausgeführt werden sollten, habe ich dazu eine abweichende Meinung. Ich persönlich sehe das skeptisch, da jeder Mensch gewisse Stärken und Neigungen hat und somit nicht jeder Entwickler auch ein guter Tester ist: Ein guter Koch ist ja auch noch lange kein geeigneter Restaurantkritiker.

Meiner Meinung nach sollten Applikationstests bevorzugt von dedizierten Testern (z. B. aus einer Testabteilung) durchgeführt werden, die Erfahrung beim Konfigurieren und Testen besitzen sowie zudem die nötige Zeit für eine gründliche Durchführung haben. Auch profitiert man von einem kritischen, unabhängigen Blick.

Zuständigkeiten bei Unit Tests

Es ist wünschenswert, Fehler möglichst schnell, noch während der Entwicklung aufzudecken, um die System- und Applikationstests zu erleichtern und eher selten mit (einfachen) Fehlern zu behindern. Dabei helfen Unit Tests und Integrationstests. Mit beiden können Probleme noch während der Entwicklung erkannt und meistens mit wenig Aufwand behoben werden. Der Grund ist einfach: Der testende Entwickler besitzt ein gutes Kontextwissen sowie ein tiefes Know-how über seine Implementierung. Zudem kann durch Unit Tests ein Sicherheitsnetz aufgebaut werden, das den Entwickler vor dem erneuten Einbau desselben Fehlers schützt.[6] Durch die Existenz und wiederholte, möglichst häufige Ausführung von Unit Tests profitiert man somit besonders bei Weiterentwicklungen, Korrekturen und umfangreicheren Refactorings oder Redesigns.

Problem Betriebsblindheit

Kommen wir nun auf das Thema Betriebsblindheit zurück. Natürlich ist diese gegeben und ein Entwickler wird zuvor bei der Implementierung begangene eigene Denkfehler wahrscheinlich auch beim Schreiben der Unit Tests machen und somit gewisse Fehler eventuell nicht entdecken können. Allerdings muss man beim Formulieren von Unit Tests über das Ganze nochmals nachdenken, wodurch auch die Chance besteht, eben doch einen Denkfehler zu entdecken.

Ergänzend können zwei Dinge helfen: Zum einen lässt sich dem durch Codereviews des zu testenden Systemteils sowie der zugehörigen Unit Tests entgegenwirken. Zum anderen sollten neben den Unit Tests ergänzend Systemtests von einer anderen Person (oder einer Testabteilung) durchgeführt werden, um die Funktionalität auf einem höheren Abstraktionsniveau und im Zusammenspiel mit anderen Systemkomponenten zu überprüfen.

Außerdem ist zu bedenken, dass es durchaus auch abweichende Interpretationen der Spezifikation geben kann, da gewisse Dinge einfach (adäquat) nicht in Prosa beschrieben werden können. Somit existiert als weiteres Problem beim Testen die Ursache »Missverständnis« bzw. Interpretationsunterschied. Das fällt nur dann auf, wenn jemand Unabhängiges auch über das Ganze nachdenkt und sich die beteiligten Personen austauschen.

Es gibt noch eine Möglichkeit: Zur systematischen Erstellung sinnvoller Testfälle bietet sich die Nutzung von Testverfahren an. Diverse davon werden im ISO-29119-Standard online unter `http://softwaretestingstandard.org/part4.php` aufgeführt – später gehe ich auf Äquivalenzklassen- und Grenzwerttests ein.

[6] Niemand wird das bewusst machen, aber manchmal kommt es bei komplexeren Änderungen doch zu solchen Problemen – gerade bei unübersichtlichem oder unverständlichem, insbesondere altem Sourcecode (auch ***Legacy-Code*** genannt).

20.1.4 Testen und Qualität

Wenn wir Software nutzen, dann freut uns als Benutzer, wenn wir gute Qualität vorfinden. Aber was macht eigentlich Qualität aus? Sicherlich spielen dabei Eigenschaften wie Zuverlässigkeit, Benutzerfreundlichkeit, Performance usw. eine wichtige Rolle. Darüber hinaus kann sich ein Softwaresystem durch klares und verständliches Design, gute Wartbarkeit und Erweiterbarkeit sowie viele automatisch ausgeführte Tests und damit eine relative große Sicherheit über die korrekte Funktionalität auszeichnen. Letztere Punkte sind gerade für uns als Softwareentwickler von gesteigertem Interesse, weil sie uns das Leben leichter machen und Probleme und Ärger zu vermeiden helfen. Mit diesem Vorwissen blicken wir auf zwei Arten von Qualität.

Äußere vs. innere Qualität

Die sogenannte ***innere Qualität***, oder auch technische Softwarequalität, hat durch die Programmierparadigmen XP und TDD und die damit verbundenen Unit Tests in den letzten Jahren verstärkt Aufmerksamkeit erfahren. Früher stand das Testen nicht so zentral im Blickpunkt und wurde zum Teil auch erst nach Abschluss der Entwicklung durchgeführt. Mitunter wurden Tests sogar nur auf System- oder Applikationsebene ausgeführt. Diese Tests überprüfen jedoch nicht (im ausreichenden Maß und nur indirekt) die einzelnen Bausteine, sondern lediglich das nach außen sichtbare Verhalten, die sogenannte ***äußere Qualität***. Diese umfasst unter anderem die für Anwender wichtigen Aspekte korrekte Funktionalität, Benutzbarkeit sowie Performance, sagt aber zu wenig über das Innenleben aus – weshalb die innere Qualität ein wichtiges Maß für uns als Entwickler ist, erkläre ich im übernächsten Abschnitt.

Normalerweise führt eine gute innere Qualität auch zu einer guten äußeren Qualität. Der Umkehrschluss gilt jedoch nicht: Es gibt Systeme, die mit ungeheurem Aufwand getestet und überarbeitet werden (müssen), bis eine akzeptable äußere Qualität wahrnehmbar ist – intern wird diese Qualität allerdings nicht erreicht. Mit zunehmender Lebenszeit eines solchen Produkts wirkt sich dessen schlechte innere Qualität immer stärker negativ aus. Erweiterungen sind dann nur noch unter großen Anstrengungen und mit enormem Aufwand zu realisieren. Dagegen erlaubt es eine gute innere Qualität, Erweiterungen in der Regel einfach umzusetzen, die zudem kaum zu Folgeänderungen in anderen Programmteilen führen.

Anfangs kostet es sicher einiges an Mehraufwand, Qualitätssicherungsmaßnahmen zu etablieren. Diese Mühe wird aber mittel- bis langfristig belohnt und wirkt sich oft extrem positiv auf die innere und äußere Qualität der Software aus. Die gestiegene innere Qualität kann zu deutlichen Kosteneinsparungen führen, da sich der Aufwand für System- und Applikationstests verringert und außerdem weniger Fehler beim Kunden auftreten, wodurch wiederum die Kundenzufriedenheit steigt und es zu weniger Servicefällen kommt. Darüber hinaus lassen sich bei Bedarf Erweiterungen leichter realisieren.

Qualitätsansprüche

Unter Qualität versteht jeder leicht etwas anderes. Für den einen ist eine gute Benutzbarkeit wichtig, ein anderer legt den Fokus auf umfangreiche Funktionalität. Immer jedoch wird man sich ein hohes Maß an Erwartungskonformität und Zuverlässigkeit wünschen. Was meine ich damit? Im richtigen Leben erwarten wir fast immer eine Qualität zwischen 90 und 100 %. Einen Kugelschreiber, der nur ab und zu schreibt, werfen wir weg. Eine Leiter, die bis 30 Kilo zugelassen ist, kaufen wir vermutlich nicht. Eine Hängebrücke, an der ein Warnschild hängt, dass die Seile alt und die Bretter morsch sind, überqueren wir wohl nicht freiwillig, sondern nur in allergrößter Not.

Wieso geben wir uns bei (eigener) Software oftmals mit nicht immer ausreichender Qualität zufrieden? Was könnten die Ursachen für die Qualitätsprobleme sein?

- Die Komplexität ist in Software häufig recht hoch.
- Die Informatik hat (leider) noch nicht den Ingenieursgrad wie die Autoindustrie oder der Maschinenbau erreicht.
- Die Einzelteile sind zum Teil nicht gut getestet.

Einfluss der Qualität der Einzelteile

Die Industrie besitzt Normen für z. B. Schrauben, Muttern und Gewinde. Weil es eben Normen sind, variieren diese nicht von Hersteller zu Hersteller, sondern man kann sich darauf verlassen, dass eine durch die Norm geforderte Qualität sowie Interoperabilität gegeben ist. In der Automobilindustrie hat man Folgendes herausgefunden: ***Je höher die Qualität der Einzelteile, desto höher ist auch die Qualität des Gesamtprodukts***. Um dies zu verstehen, betrachten wir ein einfaches lineares System mit den drei Bausteinen A, B und C (vgl. Abbildung 20-4), die jeweils eine Qualität von 90 % aufweisen.

Abbildung 20-4 *Drei voneinander abhängige Systeme mit jeweils 90 % Qualität*

Schätzen Sie mal, wie hoch die Gesamtqualität ist? Auch 90 %? Etwas weniger? Laut der Systemtheorie ergibt sich diese als das Produkt der Einzelqualitäten als:

$$0.9 * 0.9 * 0.9 = 0.73 => 73\ \%$$

Wenn wir das auf Software übertragen, werden folgende Schwierigkeiten deutlich:

1. Wer hat schon einmal Systeme mit nur drei Klassen oder Bestandteilen erstellt?
2. Mehr noch: Normale Anwendungen bestehen nicht nur aus einer Vielzahl an Klassen, sondern diese besitzen insbesondere auch komplizierte, teils verzwickte Abhängigkeiten untereinander.

Qualität – Positive Eigenschaften von Unit Tests

Bekanntlich besitzt die Qualität der Einzelbausteine einen großen Einfluss auf die Qualität des Gesamtsystems. Mit Unit Tests betreiben wir Qualitätssicherung auf Kleinteilebene und erhöhen die innere Qualität. Dabei helfen folgende Eigenschaften von Unit Tests:

- **Klares Ergebnis** – Die Durchführung der Tests liefert ein eindeutiges Ergebnis: bestanden (Grün) oder Fehler (Rot).
- **Messbar** – Die Anzahl der Testfälle und die damit geprüften Programmteile (auch Testabdeckung genannt – vgl. Abschnitt 20.7.5) lassen sich leicht auswerten. Führt man Statistik, kann man ungewünschte Trends erkennen und ihnen durch passende Maßnahmen entgegenwirken.
- **Implementierungsnah und fokussiert** – Wenn man Unit Tests parallel zur Implementierung erstellt und ausführt, erhält man schnell Feedback und oftmals lassen sich Fehler auch leicht korrigieren, weil man sich besser an Details erinnert und es wenig Mühe kostet, sich in das Thema einzudenken. Zudem ist der verursachende Sourcecode eng umrissen. Wird erst Wochen nach der Implementierung getestet, so ist die Fehlersuche oftmals um einiges aufwendiger.
- **Know-how-Gewinn** – Beim Erstellen von Unit Tests muss man sich automatisch intensiver mit den Anforderungen und den bereits bestehenden Implementierungen beschäftigen. Es baut sich dadurch kontinuierlich ein gutes Verständnis auf. Änderungen fallen anschließend leichter und lassen sich zudem mit vorhandenen Unit Tests sofort überprüfen. Das führt meistens zu durchdachteren Implementierungen, die dann wiederum weniger fehleranfällig sind und weniger Test- und Wartungsaufwand verursachen.
- **Wiederholbar** – Nach Änderungen im Sourcecode können Tests problemlos erneut ausgeführt werden. Bei positivem Ergebnis vergrößert dies die Sicherheit[7], keine Defekte eingefügt zu haben.

Die obigen Punkte legen Folgendes nahe: Von gut formulierten Unit Tests profitiert man am meisten, wenn man diese regelmäßig ausführt. Diese Form des Testens wird ***Regressionstest*** genannt. Allerdings wäre es aufwendig, wenn man dazu nach jeder Änderung alle Tests von Hand ausführen müsste. Diesen Prozess automatisiert z. B. das Tool Infinitest (vgl. Abschnitt 20.7.4) innerhalb der IDE. Ebenfalls sollten die Tests natürlich auch als Teil des automatischen Build-Laufs ausgeführt werden, wie ich es bereits in Abschnitt 2.7 motiviert habe.

[7] Für eine absolute Sicherheit müsste eine Testabdeckung von 100 % vorliegen und die Tests müssten fehlerfrei und vollständig sein. Selbst diese Aussage ist nicht ganz korrekt: Für jedes komplexere Programm garantiert eine Testabdeckung von 100 % keine absolute Sicherheit (vgl. Abschnitt 20.7.5). Im Besonderen gilt dies beim Einsatz von Multithreading. Außerdem sei noch angemerkt, dass es verschiedene Varianten bei der Berechnung der Testabdeckung gibt, etwa die der ausgeführten Anweisungen, die der geprüften Verzweigungen und Bedingungen usw.

Auswirkungen von Unit Tests auf das API-Design

Beim Schreiben von Unit Tests spielen auch Entwurfsentscheidungen, etwa solche zu Kohäsion, Kopplung und zum Design des APIs, eine Rolle. Durch das Implementieren von Testfällen nutzt man das API der eigenen Klassen, wodurch die Beurteilung leichter fällt, ob die angebotenen Schnittstellen sinnvoll und handhabbar sind. ***Unit Tests können also mögliche Schwächen in den von den Tests angesprochenen APIs vor einer Nutzung in anderen Komponenten aufdecken. Durch eine Korrektur erhält man somit gelungenere APIs***. Weiterhin kann man Unit Tests als Dokumentation des erwarteten Programmverhaltens ansehen. Diese Dokumentation ist automatisch immer aktuell, da die Testfälle ansonsten fehlschlagen würden. Das gilt jedoch nur für die mit Tests geprüften Programmteile.

Tipp: Entwicklungsbegleitend oder nach der Implementierung testen?

Man kann sich fragen, warum entwicklungsbegleitendes Testen durch Unit Tests vorteilhafter sein sollte, als Tests erst nach der Implementierung durchzuführen. Tatsächlich entsteht anfangs durch das Schreiben von Tests zusätzlicher Aufwand. Andererseits amortisiert sich der Aufwand, weil sich dadurch Fehler noch während der Implementierungsphase finden lassen. Das ist positiv, weil durch den kurzen Abstand zum Implementieren noch mehr Kontextwissen vorhanden ist und sich so mögliche Ursachen (oft sogar) intuitiv erschließen lassen. Vielfach wird die Fehleranalyse stark erleichtert, was wiederum die Behebung vereinfacht. Wird mit dem Testen dagegen erst nach Abschluss der Entwicklung begonnen, so können Fehler erst spät im Projektzyklus erkannt werden. Die zur Korrektur benötigten Informationen sind schwieriger zusammenzutragen, wodurch es teilweise zu größeren Aufwänden kommt. Diese pflanzen sich dann auch negativ in den Integrations-, System- und Applikationstests fort. Entsprechend teuer ist eine Behebung. Demzufolge amortisieren sich die Aufwände zum Erstellen von Unit Tests oft schon nach kurzer Zeit.

Außerdem gibt es in der Praxis durchaus Situationen, in denen ein Entwickler noch während des Projekts zu einem anderen Projekt oder sogar zu einer anderen Firma wechselt. Erweiterungen und Korrekturen fallen schwerer, wenn sich ein anderer Entwickler zunächst in den Sourcecode einarbeiten muss. Zudem fehlt die Sicherheit, bei Änderungen keine Fehler zu machen. Wird das Testen jedoch als integraler Bestandteil der Entwicklung angesehen, so hinterlässt ein ausscheidender Entwickler zumindest einen Teil seines Know-hows in Form von Testfällen. Dadurch können andere Entwickler Erweiterungen mit größerer Sicherheit vornehmen.

20.2 Wissenswertes zu Testfällen

Nach der einführenden Theorie geht es jetzt mit dem praktischen Teil los. Bereits in Abschnitt 2.4 habe ich eine Einführung in das Test-Framework JUnit 5 gegeben. Lesen Sie dort bitte kurz nach, um ihr Grundlagenwissen gegebenenfalls leicht aufzufrischen. Nachfolgend zeige ich das Formulieren von Testfällen mit JUnit anhand eines konkreten Beispiels und gehe dabei auf wichtige Dinge wie Namensgebung und Strukturierung ein. Danach befassen wir uns dann mit dem Thema Komplexität beim Testen.

20.2.1 Testfälle mit JUnit definieren

Bekanntermaßen prüft man mit Unit Tests kleine Bausteine, meistens einzelne Klassen, eines Softwaresystems. Ebenso haben wir gelernt, dass man durch das Schreiben von Tests dazu angehalten ist, sich Gedanken über die gewünschte zu implementierende Funktionalität zu machen. Darüber hinaus sind Unit Tests hilfreich, um spätere Änderungen abzusichern: Passende Testfälle prüfen, dass alles auch weiterhin korrekt arbeitet, jedenfalls so weit wie durch Tests abgedeckt.

Erstellen von Testfällen

Die nachfolgend gezeigte Klasse `SumWithOffsetCalculator` dient lediglich zur Demonstration erster Schritte beim Erstellen von Testfällen und bietet als Funktionalität nur eine rudimentäre Summenberechnung unter Beachtung eines Offsets:

```
public class SumWithOffsetCalculator
{
    private final int offset;

    public SumWithOffsetCalculator(final int offset)
    {
        this.offset = offset;
    }

    public int getOffset()
    {
        return offset;
    }

    public long sum_with_offset(final List<Integer> values)
    {
        Objects.requireNonNull(values, "parameter 'values' must not be null");

        return values.stream().mapToLong(value -> value).sum() + offset;
    }
}
```

Für diese Implementierung erstellen wir mit Eclipse eine passende Testklasse. Dazu wählen wir das Menü FILE –> NEW –> JUNIT TEST CASE. Es erscheint ein Dialog. Dort wählen wir `New JUnit Jupiter test`, um einen auf JUnit 5 basierenden Test zu erzeugen. Dann klicken wir auf den Button `Next`. In der Baumdarstellung der möglichen zu testenden Methode wählen wir die beiden Methoden `getOffset()` und

`sum_with_offset(List<Integer>)` aus. Dadurch entsteht für beide je eine Testmethode mit folgender initialen Implementierung:

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class SumWithOffsetCalculatorTest
{
    @Test
    void testGetOffset()
    {
        fail("Not yet implemented");
    }

    @Test
    void testSum_with_offset()
    {
        fail("Not yet implemented");
    }
}
```

Eine Testklasse besitzt keine `main()`-Methode. Stattdessen sind die von JUnit auszuführenden Testmethoden mit der Annotation `@Test` gekennzeichnet. In den Testmethoden lassen sich verschiedene Bedingungen über den erwarteten Zustand mithilfe spezifischer `assertXyz()`-Methoden prüfen. Aus Gründen der Nachvollziehbarkeit und Fokussierung auf eine spezifische Funktionalität sollte man möglichst wenige, im besten Fall nur eine `assertXyz()`-Methode zur Prüfung aufrufen. Die Methode `fail()` ist ein Spezialfall einer `assertXyz()`-Methode. Über sie wird hier initial nur signalisiert, dass der Testfall noch zu implementieren ist.

Um mit dem Handling vertraut zu werden, können Sie den eben erstellten Unit Test in Eclipse einmal ausführen. Wählen Sie im Kontextmenü RUN AS –> JUNIT TEST oder das Tastaturkürzel ALT+SHIFT+X,T. Die auszuführenden Testmethoden werden vom JUnit-Framework ermittelt und aufgerufen. Fehler werden nicht durch Rückgabewerte oder Konsolenausgaben angezeigt, sondern durch fehlgeschlagene Asserts, wobei `fail()` – wie bereits erwähnt – eine spezielle Form eines Asserts ist.

Lernen wir zunächst ein paar Tipps zum Erstellen von Testfällen kennen, bevor wir uns an die Implementierung sinnvoller Prüfungen der Klasse `SumWithOffsetCalculator` machen.

Tipps für Klarheit und Verständlichkeit Um Unit Tests wartbar und übersichtlich zu halten, ***sollte jede Testmethode jeweils nur eine Funktionalität prüfen***. Das trägt zur Kürze und Verständlichkeit bei. Zudem sollte es idealerweise nur einen Grund geben, warum ein Testfall fehlschlägt. Ebenso wichtig ist es, die Testfälle systematisch zu erstellen und nicht einfach beliebige Dinge zu testen, sondern bevorzugt die »dicken Brocken« oder die potenziell problematischen Teile zu prüfen. Um die richtigen Eingaben für die Testfälle zu wählen, bedarf es einiger Erfahrung und passend gewählter Strategien. Im nachfolgenden Abschnitt 20.2.2 lernen Sie dazu Äquivalenzklassen- und Grenzwerttests kennen.

Als Faustregel gilt, dass für eine zu testende Methode zumindest eine, oft aber besser mehrere Testmethoden existieren, die jeweils einzelne Testfälle repräsentieren. Gewöhnlich erfolgen in jeder Testmethode dazu drei Schritte (AAA-Stil):

1. ARRANGE – Zunächst wird der gewünschte Startzustand, die Ausgangsbasis des Testfalls, auch ***Testfixture*** genannt, hergestellt.
2. ACT – Danach folgen dann idealerweise ein Aufruf oder selten auch mehrere Aufrufe von Methoden der zu testenden Klasse.
3. ASSERT – Abschließend prüfen wir, ob das erwartete Ergebnis erzielt wurde. Dabei helfen verschiedene Prüfmethoden, unter anderem die bereits in Abschnitt 2.4 vorgestellten `assertEquals()`, `assertTrue()` und `assertNotNull()`, die man zur besseren Lesbarkeit der Testfälle statisch importieren sollte.

Erweiterung des Beispiels Wir wollen uns nun daranmachen, die automatisch generierte Version der Testklasse zu einem wirklichen Unit Test auszubauen, um die durch die Klasse `SumWithOffsetCalculator` implementierte Funktionalität zu prüfen. Wie schon angedeutet, sollte man eher komplexere Methoden testen statt einfache. In unserem Beispiel wollen wir daher die Methode `getOffset()` mit nur einem Testfall prüfen und die Methode `sum_with_offset(List<Integer>)` mithilfe von mehreren Testfällen, wobei sich die Frage stellt, wie und was wollen wir testen? Als erster und einfachster Test wird der »***Normalfall***« getestet. Für dieses Beispiel wird eine Liste von Zahlen addiert und anschließend das korrekte Ergebnis geprüft. Dann sollten wir uns fragen: Was ist die korrekte Summe, wenn gar keine Zahlen angegeben werden? Für diesen Fall definieren wir als Resultat den Wert 0 (zweiter Testfall). Zudem bleibt noch die Möglichkeit, dass an die Methode `sum_with_offset(List<Integer>)` versehentlich anstelle einer Liste eine `null`-Referenz übergeben wird: Dann erwarten wir eine `NullPointerException`.[8] Auch dieses Verhalten im Fehlerfall wollen wir überprüfen. Dazu dient der dritte Testfall. Gibt es weitere sinnvolle Tests? Zur Beantwortung solcher Fragen können in der Praxis die bereits kurz erwähnten Testverfahren (siehe auch nachfolgenden Praxistipp) herangezogen werden.

Wie häufig die Schleife durchlaufen wird, spielt kaum eine Rolle. Allenfalls könnte eine einelementige Liste interessant sein sowie auch eine Mischung von positiven und negativen Werten. Beides wird durch den ersten Testfall adäquat abgedeckt und mit dem Wissen über die Implementierung muss hier keine weitere Prüfung erfolgen. Interessant wäre es sicherlich, einen Überlauf von `int` zu prüfen – welche Fallstricke dort lauern, haben wir bereits in Abschnitt 5.2.1 kennengelernt. Der Einfachheit halber wollen wir die drei genannten Testfälle als ausreichend für die Methode `sum_with_offset(List<Integer>)` betrachten. Damit ergibt sich in etwa folgende Implementierung der Testklasse:

[8] Man kann darüber diskutieren, ob eine `NullPointerException` oder eine `IllegalArgumentException` die bessere Wahl darstellt.

```
class SumWithOffsetCalculatorTest
{
    @Test
    void testCalcSum_NormalInputs()
    {
        final int offset = 4711;
        var objectToTest = createSumWithOffsetCalculatorObjectForTest(offset);

        final List<Integer> values = Arrays.asList(1, 2, 3, -4);
        final long result = objectToTest.sum_with_offset(values);

        final long expected = 2 + offset;
        assertEquals(expected, result, "sum should be calculated correctly");

    }

    @Test
    void testCalcSum_SpecialCase_NoValuesGiven()
    {
        final int offset = 0;
        var objectToTest = createSumWithOffsetCalculatorObjectForTest(offset);

        final long result = objectToTest.sum_with_offset(List.of());

        final long expected = offset;
        assertEquals(expected, result, "sum should be zero for empty list");
    }

    @Test
    void testCalcSum_When_InputIsNull_ThenThrowException()
    {
        var objectToTest = createSumWithOffsetCalculatorObjectForTest(4711);

        final Executable executable = () -> objectToTest.sum_with_offset(null);

        assertThrows(NullPointerException.class, executable);
    }

    @Test
    void testGetOffset()
    {
        var objectToTest = createSumWithOffsetCalculatorObjectForTest(42);

        final int result = objectToTest.getOffset();

        final int expected = 42;
        assertEquals(expected, result);
    }

    private SumWithOffsetCalculator
            createSumWithOffsetCalculatorObjectForTest(int offset)
    {
        return new SumWithOffsetCalculator(offset);
    }
}
```

Listing 20.1 *Ausführbar als* '**SUMWITHOFFSETCALCULATORTEST**'

In diesem Beispiel erkennen wir verschiedene Dinge, auf die ich nachfolgend noch einmal separat eingehen möchte:

- **Namensgebung** – Die Testfälle besitzen sprechende Methodennamen, die etwas über die Eingabe und die erwarteten Resultate aussagen. Auf die Abweichungen zu den in Java sonst üblichen Konventionen gehe ich gleich ein.
- **AAA-Stil und Ablauf im Testfall** – Die hier gezeigten Testfälle sind immer nach einem ähnlichen Muster implementiert, das dem bereits kurz erwähnten AAA-Stil (ARRANGE - ACT - ASSERT) folgt: Zunächst werden Vorbedingungen und Initialisierungen vorgenommen (ARRANGE), danach wird eine (oder mehrere) Aktion(en) ausgeführt (ACT) und schließlich wird geprüft, ob der erwartete Zustand eingetreten ist (ASSERT). Dies lässt sich beim Test auf eine Exception nicht zu 100 % erreichen, weil die Aktion eigentlich erst im `assertThrows()` ausgeführt wird – durch die Definition der Hilfsvariablen ist die Aktion aber klar und der AAA-Stil so gut es geht bewahrt.
- **Hilfsmethoden zur Initialisierung und Testfixture** – Manchmal besitzen Testfälle eine ähnliche Initialisierung. Dafür können wir eine Hilfsmethode oder mit `@BeforeAll` bzw. `@BeforeEach` annotierte Set-up-Methoden nutzen.

Hinweis: Testverfahren zur Ermittlung von Testfällen

Um das Beispiel prägnant zu halten, wurden etwas willkürlich drei relevante Testfälle präsentiert und ich habe darauf hingewiesen, dass gegebenenfalls noch weitere sinnvolle Testfälle hinzugefügt werden könnten.

Möchte man den Prozess professionalisieren, so bietet es sich an, einige der durch ISO 29119 vorgegebenen Testverfahren zu nutzen. Das jeweilige Verfahren liefert mögliche Kriterien zur Beendigung des Testens: Dabei muss dann das Erreichen des Kriteriums nachgewiesen werden und der Entwickler muss nicht (aus dem Bauch heraus) entscheiden, ob er genug Testfälle erstellt hat.

Namensgebung von Testfällen

Wie im vorangegangenen Beispiel bereits angewendet, sollte die Benennung von Testmethoden einem gewissen Schema folgen. Ob Testmethoden mit dem Kürzel `test` starten sollten oder nicht, kann zu hitzigen Diskussionen führen. Ich habe mich durch die für JUnit 3 geltende Konvention daran gewöhnt und empfinde es oft als nützliches Unterscheidungsmerkmal zu Hilfsmethoden in Unit Tests, sodass ich es gerne verwende – aber auch nicht konsequent. Unabhängig davon sollte sich der Methodenname der Testmethode aus dem Namen der zu testenden Methode und dem zu prüfenden Sachverhalt sowie dabei geltenden Bedingungen zusammensetzen. Dadurch werden die Namen allerdings oftmals recht lang und durch die ***CamelCase-Notation auch unleserlich***. Als

Abhilfe empfiehlt es sich, einzelne Namensbestandteile per _ voneinander zu separieren und dazwischen jeweils groß zu beginnen. Roy Osherove beschreibt in seinem Blog[9] eine Variante wie folgt:

```
[UnitOfWork_StateUnderTest_ExpectedBehavior]
```

Man findet auch diese Varianten:

```
MethodName_StateUnderTest_ExpectedBehavior
MethodName_ExpectedBehavior_WhenTheseConditions
```

Besonders wichtig ist, dass der Testkontext gut beschrieben ist, also inklusive erwartetem Verhalten und möglichen Randbedingungen. Zusammen mit meinen Anmerkungen ergeben sich dann in etwa folgende geeignete Methodennamen für Testfälle:

- `calcSum_WithValidInputs_ShouldSumUpAllValues()`
- `calcSum_ThrowsException_WhenNullInput()`

Obwohl diese Namenskonvention schon ein guter Schritt in die richtige Richtung ist, geht JUnit 5 weiter. Mithilfe der Annotation `@DisplayName` unterliegt man nicht mehr den Beschränkungen der für Methodennamen gültigen Zeichen. Dadurch ist man in der Lage, sehr aussagekräftige und auch ausgefallene Namen zu vergeben, wie es in der folgenden Abbildung angedeutet ist. Diese Thematik greife ich später nochmals auf und stelle dann die Annotation `@DisplayName` detaillierter vor.

Abbildung 20-5 *Vielfältige Möglichkeiten zur Namensgebung*

Gestaltung von Testfällen: GWT- und AAA-Stil

Beim Schreiben unserer Programme haben wir Coding Conventions (vgl. Kapitel 19) kennengelernt, um den Sourcecode besser verständlich und leichter wart- sowie erweiterbar zu halten. Obwohl jeder Testfall in der Regel recht kurz sein sollte, empfiehlt es sich auch beim Implementieren von Testfällen, gewisse Standards zu befolgen, um lesbare und verständliche Unit Tests zu erstellen.

[9] http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html

Dazu hat sich der bereits kurz vorgestellte und genutzte AAA- bzw. GWT-Stil als probates Designmittel etabliert. Bekanntermaßen steht AAA für ARRANGE - ACT - ASSERT und GWT für GIVEN - WHEN - THEN. Beides meint im Prinzip das Gleiche:

- ARRANGE / GIVEN – Zunächst stellt man die Voraussetzungen her.
- ACT / WHEN – Dann führt man die zu prüfende(n) Aktion(en) des Testfalls aus.
- ASSERT / THEN – Abschließend prüft man, ob die erwarteten Ergebnisse mit den berechneten Werten übereinstimmen.

Betrachten wir ein Beispiel, um die beschriebene Vorgehensweise noch besser zu verstehen und für eigene Implementierungen in Fleisch und Blut übergehen zu lassen:

```
@Test
void demonstrateGWTStyle()
{
    // GIVEN: An empty list
    final List<String> names = new ArrayList<>();

    // WHEN: 2 elements are added
    names.add("Tim");
    names.add("Mike");

    // THEN: list should contain 2 elements
    assertEquals(2, names.size(), "list should contain 2 elements");
}
```

Je nach Komplexität des Testfalls kann man neben der einfachen Markierung durch Kommentare wie `// GIVEN` usw. ergänzend auch die Voraussetzung, Aktion und Erwartung aufführen. Die Markierung mit `// GIVEN`, `// WHEN` und `// THEN` lohnt sich bei einfachen Testfällen nicht sonderlich, weil die Implementierung schon kurz und klar ist. Um sich aber an diese Methodik zu gewöhnen, kann man die Markierung anfangs als Hilfestellung trotzdem immer nutzen.

Hilfsmethoden in Unit Tests definieren

Alle Methoden, die nicht mit `@Test` annotiert sind, werden von JUnit nicht als Testfall ausgeführt. Dadurch lassen sich in Testklassen auch Hilfsmethoden definieren, die von verschiedenen Testmethoden gemeinsam benötigte Funktionalität implementieren. Somit ist es möglich, die eigentlichen Testmethoden möglichst kurz und prägnant zu schreiben. Zur leichteren Trennung von Hilfs- und Testmethoden nutze ich für Testmethoden gerne das Präfix `test` – aber das ist Geschmackssache.

Oftmals benötigt man zur Durchführung der Tests immer wiederkehrende, oft auch aufwendige Initialisierungen, etwa eine konsistente Wertebelegung verschiedener Attribute des zu prüfenden Objekts. Man spricht bei einer Wertebelegung für eine konsistente Testumgebung von einer ***Testfixture***. Als Beispiel dafür haben wir bereits die Hilfsmethode `createSumWithOffsetCalculatorObjectForTest()` gesehen. Das lässt sich aber durch spezielle Annotations deutlicher im Sourcecode darstellen, worauf ich nun eingehe.

Spezielle Initialisierungen und Aufräumarbeiten

JUnit bietet zum Erstellen einer Textfixture und für Aufräumarbeiten folgende beiden Varianten:

- Vor bzw. nach jeder Testmethode werden, sofern vorhanden, die mit `@BeforeEach` bzw. `@AfterEach` annotierten Methoden aufgerufen.
- Sollen Aktionen nur einmalig vor bzw. nach der Durchführung aller Tests ausgeführt werden, so kann man dazu spezielle statische Methoden mit `@BeforeAll` bzw. `@AfterAll` annotieren. Diese werden dann, sofern vorhanden, aufgerufen.

Geeignet genutzt, lassen sich damit Initialisierungen vornehmen und Aufräumarbeiten durchführen bzw. Ressourcen allozieren und später wieder freigeben.

Nachfolgend ist gezeigt, wie sich unser Unit Test durch den Einsatz einer Testfixture verändert – hier auf zwei Testmethoden beschränkt:

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.function.Executable;

class SumWithOffsetCalculatorWithTestFixtureTest
{
    // Testfixture
    private int offset;
    private SumWithOffsetCalculator objectToTest;

    @BeforeEach
    void initTestFixture()
    {
        offset = 4711;
        objectToTest = new SumWithOffsetCalculator(offset);
    }

    @Test
    void testGetOffset()
    {
        final int result = objectToTest.getOffset();

        final int expected = offset;
        assertEquals(expected, result);
    }

    @Test
    void testCalcSum_NormalInputs()
    {
        final List<Integer> values = Arrays.asList(1, 2, 3, -4);
        final long result = objectToTest.sum_with_offset(values);

        final long expected = 2 + offset;
        assertEquals(expected, result, "sum should be calculated correctly");

    }
    // ...
}
```

Durch den Einsatz einer Testfixture können Initialisierungen, also die ARRANGE-Aktionen, in Testmethoden entfallen oder aber deutlich kürzer ausfallen. Allerdings erkennt man die Abläufe und Zusammenhänge nicht mehr so deutlich, was wiederum gegen deren Einsatz spricht. Der nachfolgende Praxistipp geht noch etwas genauer darauf ein.

Tipp: `@BeforeXyz` und `@AfterXyz`: Fluch oder Segen?

Tatsächlich scheinen die von JUnit gebotenen Möglichkeiten zur Initialisierung und für Aufräumarbeiten ziemlich praktisch zu sein. Wenn man rein nach der Vermeidung von Sourcecode-Duplikation geht, trifft dies zu. Das Ganze geht jedoch oft auch mit einer schlechteren Verständlichkeit der einzelnen Testfälle einher. Es ist nämlich nicht mehr möglich, auf einen Blick zu erfassen, mit welchen Werten der Test arbeitet. ***Somit empfiehlt es sich für Unit Tests mitunter, mit einer gewissen Duplikation von Sourcecode zugunsten einer besseren Verständlichkeit zu leben***.

Schauen wir uns ein abschreckendes, wenn auch künstliches Beispiel an, bei dem drei Initialisierungen vorgenommen werden – verständlich geht definitiv anders:

```
public class StrangeMultiInits
{
    static String global;

    int i;
    double d;

    @BeforeAll
    static void initGlobal()
    {
        global = "STATIC VALUE";
    }

    @BeforeEach
    void initInt()
    {
        i = 42;
    }

    @BeforeEach
    void initDouble()
    {
        d = 25375d;
        // auch möglich, i wird aber nicht gesetzt ...
        i = 13;
    }

    @Test
    @DisplayName("strange setup with @Before")
    public void strangeTestFixtureSetup()
    {
        assertEquals("STATIC VALUE", global);
        assertEquals(42, i);
        assertEquals(25375, d, 0.1);
    }
}
```

20.2.2 Problem der Kombinatorik beim Bestimmen von Testfällen

In diesem Abschnitt wollen wir uns ein grundlegendes Problem beim Schreiben von Tests anschauen, nämlich den Umfang an Eingabedaten. Um die Fehlerfreiheit eines Programms sicherzustellen, müsste man theoretisch alle möglichen Kombinationen von Eingabewerten überprüfen. Der dazu notwendige Aufwand wächst exponentiell. Selbst wenn eine Methode lediglich zwei Eingabewerte vom Typ `int` hätte, wären $2^{32} * 2^{32} = 2^{64}$ Kombinationen auszutesten.[10] Das widerspricht insbesondere auch der Anforderung, dass Unit Tests möglichst schnell ausgeführt werden sollen.

Aus der geführten Argumentation lässt sich folgern, dass ein Programm niemals (jedenfalls nicht mit vernünftigem und bezahlbarem Aufwand) vollständig getestet werden kann. ***Es gilt daher, diejenigen Testfälle zu bestimmen, die eine gute und möglichst verlässliche Aussage über die Qualität sowie den Umfang an Funktionalität erlauben.***[11] Zudem wird durch die Nutzung von Testverfahren eine sinnvolle Auswahl von Testfällen gefördert.

Reduktion des Testaufwands durch Fokussierung

Um die riesige Menge an theoretisch zu berücksichtigenden Testkombinationen *drastisch* zu reduzieren, muss man gezielt nach passenden Kandidaten für Eingabewerte suchen. ***Sinnvollerweise konzentriert man sich beim Testen bevorzugt auf die kritischen und komplexen Teile eines Softwaresystems***. Diese sollte man besonders gründlich prüfen und eher triviale Dinge (z. B. Accessor-Methoden) weniger (oder gar nicht) testen. Darüber hinaus kann man durch die geschickte Wahl von Eingaben möglichst viele gleichartige Wertebelegungen überprüfen, wie wir es im Folgenden sehen werden.

Reduktion des Testaufwands durch Äquivalenzklassentests

Betrachten wir an einem einfachen Beispiel, wie man den Testaufwand auf sinnvolle Weise reduzieren kann, indem man statt einer Vielzahl an möglichen Eingaben für den Test lediglich typische Vertreter für Eingabewerte nutzt. Nehmen wir an, eine Methode `calcDiscount(int count)` berechne den Rabatt für eine Bestellung basierend auf der Anzahl der Waren nach folgenden Regeln:

$$\begin{aligned} count < 50 \quad &=> \quad 0\,\% \\ 50 <= count <= 1000 \quad &=> \quad 4\,\% \\ count > 1000 \quad &=> \quad 7\,\% \end{aligned}$$

[10] Wollte man die Effekte durch Fehler im Betriebssystem oder der Ablaufumgebung (JVM, Application Server, Webcontainer) sicher ausschließen, so müsste man tatsächlich alle Werte durchprüfen. Das wäre zeitlich nicht realisierbar und auch schon etwas neurotisch.

[11] Man kann Fehlerfreiheit mit Tests nicht sicherstellen, sondern nur Fehler aufdecken.

Schauen wir uns die Implementierung der Methode an, die hier bewusst ein paar Kleinigkeiten falsch realisiert, um auf Probleme aufmerksam machen zu können.

```
// ACHTUNG: Enthält bewusst ein paar kleine Fehler
public int calcDiscount(final int count)
{
    if (count <= 50)
        return 0;
    if (count > 50 && count < 1000)
        return 4;
    if (count > 1000)
        return 7;

    throw new IllegalStateException("programming problem: should never " +
                "reach this line. value " + count + " is not handled!");
}
```

Anhand der drei Gruppierungen können wir drei sogenannte ***Äquivalenzklassen*** bilden. Weil diese für unterschiedliche Eingaben gleiche Ergebnisse produzieren, reicht es, wenn man jeweils nur einen Repräsentanten daraus wählt und diesen prüft. Für die Fälle 0 %, 4 % und 7 % sind in Tabelle 20-1 typische Vertreter angegeben.

Tabelle 20-1 *Testfälle aus drei Äquivalenzklassen*

Typischer Repräsentant für die Anzahl	Erwartetes Ergebnis
20	0
200	4
2000	7

Optimistisch machen wir uns ans Werk, um die Rabattberechnung mithilfe folgender drei Tests abzuprüfen – wobei hier ausnahmsweise auf den AAA-Stil verzichtet wird, weil die Aufrufe als Einzeiler trotzdem verständlich bleiben:

```
@Test
void testCalcDiscount_SmallOrder_NoDiscount()
{
    final int smallAmount = 20;
    assertEquals(0, calculator.calcDiscount(smallAmount), "No discount");
}

@Test
void testCalcDiscount_MediumOrder_MediumDiscount()
{
    final int mediumAmount = 200;
    assertEquals(4, calculator.calcDiscount(mediumAmount), "4 % discount");
}

@Test
void testCalcDiscount_BigOrder_BigDiscount()
{
    final int bigAmount = 2000;
    assertEquals(7, calculator.calcDiscount(bigAmount), "7 % discount");
}
```

Listing 20.2 *Ausführbar als* '**DISCOUNTCALCULATORV1TEST**'

Wir sehen einige Magic Numbers bzw. Strings. Das ist in kleinerem Umfang wie hier zu tolerieren, wenn es für eine gute Lesbarkeit sorgt – später dazu mehr.

Für die drei Eingaben wird der Rabatt – wie erwartet – korrekt berechnet. Sind wir damit schon fertig? Nein! Ergänzend bietet es sich an, mit ungültigen Werten bzw. fehlerhaften Eingaben zu testen, also einer Anzahl kleiner 0 oder größer als einem erwarteten Maximalwert. Zur Vereinfachung des Beispiels beschränke ich mich aber auf die drei genannten Äquivalenzklassen. ***Die Erfahrung in der Praxis zeigt, dass neben Äquivalenzklassentests weitere Prüfungen sinnvoll sind. Erfahrungsgemäß treten an den Grenzen von Wertebereichen immer wieder Probleme auf***.

Qualitätssicherung durch Grenzwerttests

Daher sollte man zusätzlich zu den Äquivalenzklassen auch Grenzwerte prüfen. Dazu erweitern wir den vorherigen Test und prüfen, ob auch die direkten Nachbarn der Intervallgrenzen, hier die Werte 49, 50, 51 sowie 999, 1000, 1001, der richtigen Äquivalenzklasse zugeordnet werden und damit den korrekten Rabatt liefern:

```
// ACHTUNG: Hier noch einfache Implementierung mit Magic Numbers / Strings

@Test
void testCalcDiscount_Input_49_ShouldHaveNoDiscount()
{
    assertEquals(0, calculator.calcDiscount(49), "No discount");
}

@Test
void testCalcDiscount_Input_50_ShouldHaveMediumDiscount()
{
    assertEquals(4, calculator.calcDiscount(50), "4 % discount");
}

@Test
void testCalcDiscount_Input_51_ShouldHaveMediumDiscount()
{
    assertEquals(4, calculator.calcDiscount(51), "4 % discount");
}

@Test
void testCalcDiscount_Input_999_ShouldHaveMediumDiscount()
{
    assertEquals(4, calculator.calcDiscount(999), "4 % discount");
}

@Test
void testCalcDiscount_Input_1000_ShouldHaveMediumDiscount()
{
    assertEquals(4, calculator.calcDiscount(1000), "4 % discount");
}

@Test
void testCalcDiscount_Input_1001_ShouldHaveBigDiscount()
{
    assertEquals(7, calculator.calcDiscount(1001), "7 % discount");
}
```

Listing 20.3 *Ausführbar als* '**DISCOUNTCALCULATORV1TEST_CORNERCASES**'

Beim Grenzwerttest gilt, dass der Test als ausreichend anzusehen ist, wenn jede Grenze mit 2 (oder 3) Testfällen überprüft wird – mit 3 ist man immer auf der sicheren Seite. Das Kriterium erfüllen wir schon einmal, allerdings könnten und sollten wir uns langsam an den ganzen Magic Numbers und Strings stören. Darauf komme ich gleich zurück. Schauen wir zunächst einmal, was passiert, wenn man diese Testfälle ausführt.

Abbildung 20-6 *Ergebnisse von* DISCOUNTCALCULATORV1TEST_CORNERCASES

Die Tests für die Eingabe 50 als auch für den Wert 1000 schlagen fehl. Im ersten Fall wird dem Wert 50 ein falscher Rabatt zugeordnet, was sich durch ein Blick in die Implementierung leicht nachvollziehen lässt. Dort sieht man auch, dass der Wert 1000 von keinem `if` abgedeckt wird, weshalb eine Exception ausgelöst wird.

Softwaretechnische Verbesserungen

Im Sourcecode sehen wir einige Duplikationen und Magic Numbers bzw. Strings. Das ist in kleinerem Umfang in Unit Tests dann zu tolerieren, wenn man die Implementierung der Tests dadurch kurz und verständlich halten kann, wie wir es auch eingangs gesehen haben. Mit zunehmender Zahl an hartcodierten Werten kippt das Ganze und man sollte sich Abhilfemaßnahmen überlegen. Das ist hier empfehlenswert, weil wir schon fast zehn Testfälle definiert haben.

In unserem Beispiel erkennen wir folgende Schwachstellen bzw. Duplikationen:

1. Werte für Rabatte – Sollten als Konstanten definiert werden.
2. Hinweistexte – Für jede Art von Rabatt wird immer wieder der gleiche Text angegeben, der sich zudem ständig wiederholt. Würde sich der Rabatt von 4 auf 5 % ändern, müssten wir an vielen Stellen ändern.
3. Kombination Rabatt und Hinweistext – Momentan sind Hinweistext und Rabatt zwar semantisch verbunden, im Programm aber nicht. Somit ist es möglich, dass man Fehler macht und dies später bei den Tests irritiert.

Diese Probleme werden wir nun Schritt für Schritt angehen und dabei sowohl die eigentliche Implementierung als auch die Tests softwaretechnisch verbessern.

Definition von Konstanten Wir definieren Konstanten zu unseren Rabatten:

```
/* private */ static final int NO_DISCOUNT = 0;
/* private */ static final int MEDIUM_DISCOUNT = 4;
/* private */ static final int BIG_DISCOUNT = 7;
```

Korrektur der Implementierung basierend auf den Testresultaten Unsere gerade implementierten Grenzwerttests decken die zur Demonstration bewusst integrierten Implementierungsfehler auf: Wie erwartet, werden die Werte 50 und 1000 nicht korrekt abgebildet. Der Wert 50 wird (versehentlich) noch auf keinen Rabatt abgebildet, was zur Fehlermeldung »`java.lang.AssertionError: Medium discount expected:<4> but was:<0>`« führt. Außerdem wird der Wert 1000 durch derzeit noch durch keinen Testfall abgedeckt und wir erhalten folgende Exception: »`java.lang.IllegalStateException: programming problem: should never reach this line. value 1000 is not handled!`« Basierend auf dem Hinweistext fällt es nicht schwer, die Implementierung wie folgt zu korrigieren und dabei direkt auch unsere zuvor definierten Konstanten zu nutzen:

```
public int calcDiscount(final int count)
{
    if (count < 50)
        return NO_DISCOUNT;
    if (count >= 50 && count <= 1000)
        return MEDIUM_DISCOUNT;
    if (count > 1000)
        return BIG_DISCOUNT;

    throw new IllegalStateException("programming problem: should never " +
                      "reach this line. value " + count + " not handled!");
}
```

Eigentlich sollte das Auslösen einer `IllegalStateException` gar nicht notwendig sein, wenn man alle Pfade vollständig abgedeckt hat. Da hier aber erfahrungsgemäß immer wieder Fehler lauern, ist man mit dem gezeigten Konstrukt auf der sicheren Seite und es lassen sich Flüchtigkeitsfehler leichter aufdecken.

Verbesserung der Lesbarkeit der Tests durch Konstanten Mithilfe der eingeführten Konstanten kann man auch die Testfälle lesbarer schreiben. Das zeige ich hier exemplarisch für zwei davon:

```
@Test
public void testCalcDiscount_Input_49_ShouldHaveNoDiscount()
{
   assertEquals(NO_DISCOUNT, calculator.calcDiscount(49), "No discount");
}
```

```
@Test
void testCalcDiscount_Input_1000_ShouldHaveMediumDiscount()
{
   assertEquals(MEDIUM_DISCOUNT, calculator.calcDiscount(1000), "4 % discount");
}
```

Zwar nutzen wir die Rabattwerte als Konstanten, jedoch treten jetzt die Punkte 2 und 3 stärker zutage: die Magic Strings für Hinweistexte und die fehlende programmatische Zuordnung.

Denken wir kurz nach, was wir tun können. Zur Kombination von Rabattwert und Hinweistext bietet sich die Definition einer Map an, hier mithilfe des Java-9-Sprachfeatures Collection-Factory-Methoden. Statt jedoch den Text wie vielleicht aus Bequemlichkeit als Magic Strings zu definieren, ist es noch etwas schöner und weniger fehleranfällig, die Prozentzahl im Text aus der Konstante abzuleiten:

```
// Java 9: Collection Factory Methodes
private static final Map<Integer, String> discountToInfoMap = Map.of(
        NO_DISCOUNT, "no discount",
        MEDIUM_DISCOUNT, MEDIUM_DISCOUNT + " % discount",
        BIG_DISCOUNT, BIG_DISCOUNT + " % discount");
```

Um nun die Prüfung und die Zuordnung zu vereinfachen, können wir die Map in folgender Hilfsmethode nutzen, die zur Definition der Variablen das mit Java 10 eingeführte `var` nutzt (vgl. Abschnitt 5.7.2).

```
private void assertDiscount(final int expected, final int amount)
{
    // Java 10: var
    var infoMsg = discountToInfoMap.getOrDefault(amount, "n/a");

    assertEquals(expected, calculator.calcDiscount(amount), infoMsg);
}
```

Schauen wir uns wieder exemplarisch für drei Werte an, wie dies die Formulierung der Testfälle vereinfacht:

```
@Test
void testCalcDiscount_Input_49_ShouldHaveNoDiscount()
{
    assertDiscount(NO_DISCOUNT, 49);
}

@Test
void testCalcDiscount_Input_1000_ShouldHaveMediumDiscount()
{
    assertDiscount(MEDIUM_DISCOUNT, 1000);
}

@Test
public void testCalcDiscount_Input_1001_ShouldHaveBigDiscount()
{
    assertDiscount(BIG_DISCOUNT, 1001);
}
```

Listing 20.4 *Ausführbar als* '**DISCOUNTCALCULATORV2TEST**'

Durch unsere Refactoring-Arbeiten haben wir den Sourcecode deutlich kürzer und verständlicher machen können. Das war schon ein sehr guter Schritt.

Weniger Testmethoden durch parametrierte Tests Nachfolgend möchte ich vorgreifend die Technik der parametrierten Tests zur Reduktion der Menge an Sourcecode nutzen. Parametrierte Tests lernen wir später genauer kennen, weshalb ich diese hier nicht im Detail beschreibe. In aller Kürze: Diese erlauben die Ausführung der Tests für verschiedene Werte für eine annotierte Testmethode:

```
public class DiscountCalculatorV2AsParameterizedTest
{
    private final DiscountCalculatorV2 calculator = new DiscountCalculatorV2();

    @ParameterizedTest(name = "{index}: amount {0} => discount {1}")
    @CsvSource({ "20, 0", "200, 4", "2000, 7" })
    void testCalcDiscount_EquivalenceClasses(int amount, int expectedDiscount)
    {
        assertEquals(expectedDiscount, calculator.calcDiscount(amount));
    }

    @ParameterizedTest(name = "{index}: amount {0} => discount {1}")
    @CsvSource({ "49, 0", "50, 4", "51, 4",
                 "999, 4", "1000, 4", "1001, 7" })
    void testCalcDiscount_CornerCases(int amount, int expectedDiscount)
    {
        assertEquals(expectedDiscount, calculator.calcDiscount(amount));
    }
}
```

Listing 20.5 *Ausführbar als* '**DISCOUNTCALCULATORV2ASPARAMETERIZEDTEST**'

Führen wir den Test als DISCOUNTCALCULATORV2ASPARAMETERIZEDTEST aus, so erhalten wir folgende Darstellung in Eclipse, die die mehrfache Ausführung mit unterschiedlicher Wertebelegung gut zeigt:

Abbildung 20-7 *Ergebnisse von* DISCOUNTCALCULATORV2ASPARAMETERIZEDTEST

Durch das Einführen von Konstanten konnten wir die Tests übersichtlicher gestalten und dann durch den Einsatz von parametrierten Tests die Anzahl der Methoden reduzieren. Wenn wir genau hinschauen, könnte uns eine Sache noch stören. Es gibt nämlich einen Wermutstropfen: In den parametrierten Tests erfolgt die Angabe der Werte in Form von Strings und somit wird das Ganze wieder fragil bei Änderungen. Wie wir auch diesen Nachteil beheben, schauen wir uns später im Rahmen der Beschreibung der parametrierten Tests an.

Sinnvolle weitere Prüfungen und Vorbedingungen

Generell sollte man ergänzend Über- oder Unterschreitungen von Grenzwerten prüfen, etwa negative Werte oder die Überschreitung einer maximalen Anzahl. Dabei handelt es sich um Eingaben, die fachlich keinen Sinn machen, aber zu unsinnigen Ergebnissen führen können. In unserem Beispiel sollte man zumindest negative Werte für die Anzahl durch eine Vorbedingung ausschließen. Dabei hilft die Utility-Klasse `Preconditions` aus Google Guava (vgl. Abschnitt 9.2.4):

```
int calcDiscount(final int count)
{
    // Absicherung von Vorbedingungen
    // if (count < 0)
    //     throw new IllegalArgumentException("count should not be negative");
    Preconditions.checkArgument(count >= 0, "count should not be negative");

    if (count < 50)
        return NO_DISCOUNT;
    if (count >= 50 && count <= 1000)
        return MEDIUM_DISCOUNT;
    if (count > 1000)
        return BIG_DISCOUNT;

    throw new IllegalStateException("programming problem: should never " +
                    "reach this line. value " + count + " not handled!");
}
```

Im Unit Test ergänzen wir folgenden Testfall, der sich zusammen mit den anderen als DISCOUNTCALCULATORV3TEST starten lässt:

```
@Test
void testCalcDiscount_NegativeInput_ShouldThrowException()
{
    assertThrows(IllegalArgumentException.class,
                 ()-> calculator.calcDiscount(-1));
}
```

Fazit

Auf Basis diverser einzelner Testmethoden haben wir unter Zuhilfenahme der passenden Techniken aus JUnit 5 die Prüfungen deutlich vereinfachen und den gesamten Test so gestalten können, dass problemlos weitere Testfälle ergänzt werden können. Adäquate Techniken sowie diverse hilfreiche Erweiterungen lernen wir im Anschluss kennen.

20.3 Besondere Assertions und Annotations

Bislang haben wir uns vorwiegend auf einfache Assertions beschränkt. In der Praxis benötigt man des Öfteren aber weiter gehende Möglichkeiten, um in Testfällen erwartete Testresultate abzuprüfen. Statt der in JUnit 4 dazu üblichen JUnit Rules existieren in JUnit 5 andere, ähnlich mächtige Konzepte, die sogar oftmals noch eleganter in der Handhabung sind. Schauen wir uns diese nun genauer an.

Benennung von Testmethoden und die Annotation `@DisplayName`

Standardmäßig ergibt sich der Name eines Testfalls aus dem Methodennamen. Dadurch ist man recht eingeschränkt, unter anderem sind so Sonderzeichen nicht möglich. Mit JUnit 5 wird diese Beschränkung durch die Annotation `@DisplayName` aufgehoben. Es ist möglich, auch Zeichen zu verwenden, die im Methodennamen nicht gültig wären, etwa Leerzeichen, spezielle Unicode-Zeichen oder auch die Zeichen '!' sowie '+', '-', '/' und '*', die man etwa zur Benennung bei mathematischen Operationen nutzen kann:

```
@Test
@DisplayName("This is a name with unicode \u1234")
public void changeMyName()
{
    assertTrue(true, "name is changed to that of @DisplayName");
}

@Test
@DisplayName("Simple Math: (1 * 2 * 3) / 4 = 6/4 = 3/2 = 1.5")
void divideResultOfMultiplication()
{
    assertEquals((1 * 2 * 3) / 4d, 1.5, 0.1);
}

@Test
@DisplayName("GET 'http://localhost:8080/products/4711' user: Peter Müller")
public void getProductFor4711()
{
    // ..
}
```

Listing 20.6 *Ausführbar als* '**DISPLAYNAMETEST**'

Führen Sie das Programm DISPLAYNAMETEST aus, um verschiedene Möglichkeiten sprechender Namen für Testfälle zu erkennen, etwa für REST-Abfragen.

Abbildung 20-8 *Vielfältige Möglichkeiten zur Namensgebung*

Prüfung mehrerer Asserts mit `assertAll()`

Bei der Formulierung von Testfällen sind immer mal wieder mehrere Bedingungen zu prüfen. Das scheint gegen die Regel »ein Testfall, ein Assert« zu verstoßen. Allerdings sollte man das nicht immer ganz so streng auslegen, insbesondere dann nicht, wenn nur semantisch zusammengehörende Daten geprüft werden, wie etwa die Bestandteile einer Adresse. Mit JUnit 5 lässt sich diese semantische Klammer durch die Methode `assertAll()` ausdrücken.

Im Beispiel prüfen wir die drei Adressbestandteile PLZ, Ort und Land mit Lambdas in Kombination wie folgt:

```
@Test
void assertAachenZipAndCityAndCountry()
{
    final Address address = // ...

    assertAll("Adressbestandteile",
              () -> assertEquals(52070, address.getZipCode()),
              () -> assertEquals("Aachen", address.getCity()),
              () -> assertEquals("Deutschland", address.getCountry()));
}
```

Das Ganze hat aber noch eine weitere Implikation, die wir uns an der Definition einer Person und dem Vergleich der Attribute anschauen wollen, wobei alle drei in den `assert()`-Methoden geprüften Werte bewusst abweichen:

```
@Test
void multipleAssertsforOneTopic_Diff1()
{
    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertEquals("Tim", mike.name);
    assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth);
    assertEquals("Kiel", mike.homeTown);
}

@Test
void multipleAssertsforOneTopic_Diff2()
{
    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertAll(() -> assertEquals("Tim", mike.name),
              () -> assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth),
              () -> assertEquals("Kiel", mike.homeTown));
}
```

Während wir uns im ersten Fall nach der Korrektur des ersten Fehlers weiter von Fehler zu Fehler hangeln müssen, erhalten wir im zweiten Fall direkt eine Auflistung mit allen fehlschlagenden Asserts und können so das Ganze leichter korrigieren.

Behandlung erwarteter Exceptions mit `assertThrows()`

Manchmal wird das Auftreten von Exceptions während der Abarbeitung der Aktionen eines Testfalls erwartet und ein Ausbleiben würde einen Fehler darstellen. Ein Beispiel ist der bewusste Zugriff auf ein nicht existentes Element eines Arrays. Eine `ArrayIndexOutOfBoundsException` sollte die Folge sein. Um erwartete Exceptions im Testfall so zu behandeln, dass diese einen Testerfolg und keinen Fehlschlag darstellen, besitzt man verschiedene Alternativen.

Seit JUnit 5 dient die Methode `assertThrows()` zur Verarbeitung von Exceptions. Sie schlägt fehl (produziert einen Testfehler), wenn die ausgeführte Methode nicht die erwartete Exception auslöst. Zudem gibt die Methode die ausgelöste Exception zurück, sodass man noch weitere Prüfungen vornehmen kann, beispielsweise, ob im Text der Exception die gewünschten und erwarteten Informationen bereitgestellt werden. Im ersten Testfall prüfen wir einen unzulässigen Indexzugriff. Im zweiten lösen wir bewusst eine Exception aus und prüfen auf erwartete Inhalte wie Text und Ursache:

```
public class AssertThrowsTest
{
    @Test
    void arrayIndexOutOfBoundsExceptionExpected()
    {
        var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7 };

        final Executable action = () ->
        {
            numbers[1_000] = 13;
        };

        assertThrows(ArrayIndexOutOfBoundsException.class, action);
    }

    @Test
    void illegalStateExceptionWithMessageTextExpected()
    {
        final String errorMsg = "XYZ is not initialized";

        final Executable action = () ->
        {
            throw new IllegalStateException(errorMsg,
                                            new IOException("IO"));

        };

        final IllegalStateException exception =
                                    assertThrows(IllegalStateException.class,
                                                 action);
        assertEquals(errorMsg, exception.getMessage());
        assertEquals(IOException.class, exception.getCause().getClass());
    }
}
```

Listing 20.7 *Ausführbar als* '**ASSERTTHROWSTEST**'

Berücksichtigung von Timeouts mit `assertTimeout()`

Per Definition sollten unsere Testfälle schnell abgearbeitet werden, um zeitnah Feedback zu bekommen, den sogenannten ***Feedbackzyklus*** also möglichst kurz zu halten. Sofern Unit Tests kurz, isoliert und fokussiert sind, so ist dies oft automatisch gegeben. Mitunter wird aber – insbesondere für Integrationstests – auch Funktionalität getestet, die längere Ausführungszeiten besitzt oder gar in eine Endlosschleife gerät.

Praktischerweise bietet JUnit 5 die Möglichkeit, die Ausführungsdauer durch die Angabe von Timeouts zu begrenzen. Dazu dienen die Methoden `assertTimeout()` und `assertTimeoutPreemptively()`, denen man als Parameter eine Timeout-Zeit sowie die auszuführende Aktion mitgibt. Nachfolgend simulieren die Methoden `longRunningAction()` und `shortRunningAction()` eine lang bzw. kurz dauernde Verarbeitung. Diese werden mithilfe von `assertTimeout()` sowie `assertTimeoutPreemptively()` entsprechend geprüft. Schließlich wird noch der Abbruch einer Endlosschleife gezeigt:

```
public class AssertTimeoutTest
{
    @Test
    void timeout_finshed_before()
    {
        final Executable action = () -> shortRunningAction();

        assertTimeout(Duration.ofMillis(100), action);
    }

    @Test
    void timeoutExceeded_Keep_On_Running()
    {
        final Executable action = () -> longRunningAction();

        // wartet so lange, bis die Aktion fertig ist, hier also 2 Sekunden
        // und nicht nur die Timeout-Zeit
        assertTimeout(Duration.ofMillis(500), action);
    }

    @Test
    void timeoutExceeded_Preemptive_Termination()
    {
        final Executable action = () -> longRunningAction();

        // unterbricht nach angegebener Timeout-Zeit
        assertTimeoutPreemptively(Duration.ofMillis(500), actioin);
    }

    @Test
    void loopForever() throws InterruptedException
    {
        final Executable endlessLoopAction = () ->
        {
            // Bewusste Endlosschleife zur Demonstration
            for (;;)
            {
                TimeUnit.SECONDS.sleep(1);
            }
        };
```

```
        // ACHTUNG: Ganz wichtig, hier assertTimeoutPreemptively(), weil
        // man ansonsten unendlich auf das Ende der Testausführung wartet ...
        assertTimeoutPreemptively(Duration.ofMillis(500), endlessLoopAction);
    }

    void longRunningAction() throws InterruptedException
    {
        TimeUnit.SECONDS.sleep(2);
    }

    void shortRunningAction() throws InterruptedException
    {
        TimeUnit.MILLISECONDS.sleep(50);
    }
}
```

Listing 20.8 *Ausführbar als* '**ASSERTTIMEOUTTEST**'

Ein Testfall wird als nicht bestanden gewertet, wenn die Ausführung länger als die Timeout-Zeit dauert. Demnach ist nur der erste Testfall erfolgreich. Die anderen Testfälle mit einem Timeout-Wert von 500 ms schlagen alle fehl.

Das Beispiel verdeutlicht noch Folgendes: Die Methode `assertTimeout()` prüft lediglich, ob es zu einem Timeout kommt, die Ausführung der Aktion des Testfalls wird jedoch auch beim Überschreiten des Timeout-Werts im Gegensatz zu JUnit 4 nicht unterbrochen. Dafür gibt es die Methode `assertTimeoutPreemptively()`. Diese unterbricht die Abarbeitung nach Ablauf der angegebenen Timeout-Zeit. Dies ist immer dann zu bevorzugen, wenn man nicht sicher ist, dass der zu prüfende Applikationscode tatsächlich terminiert oder gar eine Endlosschleife ausführt.

Wieso benötigt es beide Methoden? Wieso gibt es überhaupt die Methode `assertTimeout()`? Das ist eine berechtigte Frage. Zur Beantwortung stellen Sie sich bitte vor, Sie müssten eine ziemlich komplexe und aufwendige Berechnung überarbeiten. Nehmen wir weiterhin vereinfachend an, es gäbe Unit Tests, die das korrekte Ergebnis prüfen, etwa mit `assertEquals()`. Nun gibt es vom Business aber die Vorgabe, die bisherige Laufzeit zu reduzieren. Somit lautet die Antwort wie folgt: Manchmal möchte man prüfen, dass die Aktionen schnell genug ausgeführt werden und zudem das korrekte Ergebnis liefern. Das lässt sich mit `assertTimeoutPreemptively()` nicht prüfen, sondern nur mit `assertTimeout()`. Dazu erweitern wir einen existierenden Unit Test um einen Aufruf von `assertTimeout()`, und übergeben die Aktion im Gegensatz zu den Beispielen zuvor als `ThrowingSupplier`. Die Berechnung `complexCalc()` könnte anfangs noch zu langsam sein – falls dann aber die benötigte Optimierung erfolgt ist, kann man wie folgt prüfen, ob das Berechnungsergebnis stimmt:

```
@Test
void timeoutExceeded_ComplexCalc()
{
    final ThrowingSupplier<String> action = () -> complexCalc(50);

    String result = assertTimeout(Duration.ofMillis(500), action);
    assertEquals("EXPECTED_RESULT", result);
}
```

Prüfung von Collections mit `assertIterableEquals()`

Mitunter besteht beim Testen der Bedarf, die Inhalte zweier Collections miteinander zu vergleichen. Handelt es sich um den gleichen Typ, etwa `List<E>`, so lässt sich dies einfach mit `assertEquals()` schreiben:

```
@Test
void assertSameContent()
{
    final List<String> result = List.of("Mike", "Peter", "Tim");
    final List<String> expected = List.of("Mike", "Peter", "Tim");

    assertEquals(expected, result);
}
```

Manchmal möchte man beispielsweise die Elemente aus einem `TreeSet<E>` mit denjenigen aus einer `ArrayList<E>` vergleichen. Für derartige Fälle bietet sich `assertIterableEquals()` an. Damit wird geprüft, ob die beiden als `Iterable<E>` übergebenen Collections die gleiche Anzahl und Reihenfolge von Elementen aufweist:

```
@Test
void assertSameContent_ChangedOrder_IterableEquals()
{
    var result = Map.of("Mike", 49, "Peter", 17, "Tim", 49);
    final Set<String> keys = new TreeSet<>(result.keySet());
    final List<String> expected  = List.of("Mike", "Peter", "Tim");

    assertIterableEquals(expected, keys);
}
```

Prüfung von Arrays mit `assertArrayEquals()`

Während Collections eine passend überschriebene `equals()`-Methode besitzen, gilt dies für Arrays nicht – eine Sache, die man gerne einmal vergisst. Schauen wir uns ein einfaches Beispiel für jeweils offensichtlich identische Arrays an:

```
// ACHTUNG: Fallstrick fehlerhafter Vergleich
@Test
void arraysCompare()
{
    final String[] words = { "Word1", "Word2" };
    final String[] expected = { "Word1", "Word2" };

    assertEquals(expected, words);
}

@Test
void nestedArraysCompare()
{
    final String[][] nested = { { "Line1", "Word1" },
                                { "Line2", "Word2" } };
    final String[][] expected = { { "Line1", "Word1" },
                                  { "Line2", "Word2" } };

    assertEquals(expected, nested);
}
```

In beiden Fällen schlägt der Vergleich fehl, da auf Referenzen und nicht der Inhalt verglichen wird.

Die Abhilfe erfolgt ganz einfach durch den Einsatz der Methode `assertArrayEquals()`, die auch mehrdimensionale Arrays korrekt verarbeiten kann:

```
@Test
void arraysCompare()
{
    final String[] words = { "Word1", "Word2" };
    final String[] expected = { "Word1", "Word2" };

    assertArrayEquals(expected, words);
}

@Test
void nestedArraysCompare()
{
    final String[][] nested = { { "Line1", "Word1" },
                                { "Line2", "Word2" } };
    final String[][] expected = { { "Line1", "Word1" },
                                  { "Line2", "Word2" } };

    assertArrayEquals(expected, nested);
}
```

Kategorisierung von Tests und die Annotation `@Tag`

Bei der Gruppierung und Kategorisierung von Tests ist die Annotation `@Tag` nützlich, etwa um beim Ausführen der Tests gewisse Kategorien zu inkludieren bzw. zu exkludieren. So kann man beispielsweise aufwendige Integrationstests markieren und im Normalfall ausschließen, um den ***Feedbackzyklus*** kurz zu halten. Nur bei besonderen Build-Ausführungen findet dann eine umfassendere Prüfung statt.

Nachfolgend ist ein Beispiel für die zwei Kategorien `fast` und `slow` gezeigt:

```
public class CategoriesWithTagsTest
{
    @Test
    @Tag("fast")
    public void fast()
    {
        assertTrue(true);
    }

    @Test
    @Tag("slow")
    public void slow() throws InterruptedException
    {
        TimeUnit.SECONDS.sleep(2);

        assertTrue(true);
    }
}
```

Listing 20.9 *Ausführbar als* '**CATEGORIESWITHTAGSTEST**'

Um nur die jeweils gewünschten Tags auszuführen, sind diese passend in den jeweiligen Build-Dateien `build.gradle` und `pom.xml` anzugeben – für Gradle etwa folgendermaßen:

```
useJUnitPlatform() {
    includeTags 'fast', 'COOL', 'unit'
    excludeTags 'regression', ?bug-4711'
}
```

Für Maven ergänzt man die Konfiguration für das Surefire-Plugin um die Informationen in `groups` und `excludedGroups`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <configuration>
        <argLine>@{argLine} --enable-preview</argLine>
        <testFailureIgnore>true</testFailureIgnore>
        <!-- include tags -->
        <groups>fast, unit</groups>
        <!-- exclude tags -->
        <excludedGroups>slow</excludedGroups>
    </configuration>
</plugin>
```

Beachten Sie bitte, dass die Angabe `@argLine` wichtig ist, um nicht andere Werte zu überschreiben. Eine reine Angabe zur Aktivierung der Preview-Features

```
<argLine>--enable-preview</argLine> <!-- ACHTUNG: potenziell ungünstig -->
```

führt dazu, dass dann Tools wie JaCoCo nicht mehr korrekt arbeiten.

Auch in der IDE kann man konfigurieren, welche Tags beim Ausführen von Tests inkludiert bzw. exkludiert werden sollen, nachfolgend exemplarisch für Eclipse gezeigt.

Abbildung 20-9 *Unit Tests mit inkludierten und exkludierten Tags*

Zugriff auf Metainformationen

In nachfolgenden Beispiel erkennen wir eine wesentliche Neuerung von JUnit 5: Testmethoden müssen nicht mehr parameterlos sein, sondern ihnen können Daten übergeben werden. Wie im Listing gezeigt, lassen sich diverse Informationen zu einem Testfall durch JUnit 5 in Form des Interface `org.junit.jupiter.api.TestInfo` injizieren, etwa um daraus den Methodennamen oder weitere Metainformationen zu ermitteln.

Betrachten wir ein Beispiel, das unter anderem den Methodennamen, den Anzeigenamen sowie etwaige Tags ausliest und auf diese prüft:

```
public class InjectionOfMetaDataTest
{
    @Test
    @DisplayName("unicode \u1234")
    @Tag("very")
    @Tag("special")
    public void injectParams(final TestInfo testInfo)
    {
        assertEquals("injectParams", testInfo.getTestMethod().get().getName());
        assertEquals("unicode \u1234", testInfo.getDisplayName());
        assertEquals(Set.of("very", "special"), testInfo.getTags());
    }
}
```

Listing 20.10 *Ausführbar als* **'INJECTIONOFMETADATATEST'**

Ausführungsreihenfolge von Tests festlegen mit `@Order`

Normalerweise sollten Unit Tests voneinander unabhängig sein und die dort enthaltenen Testfälle sollten in beliebiger Reihenfolge ausgeführt werden können. So viel zur Theorie. Tatsächlich ist dies ein durchaus erstrebenswertes Ziel, da es zeigt, dass die Tests wirklich dediziert eine kleine Funktionalität unabhängig prüfen. Zum Teil möchte man jedoch eine Reihenfolge explizit festlegen können, etwa weil man eine Abfolge von Aktionen testen möchte, beispielsweise das Erzeugen, Ändern und Löschen von Elementen. Dafür ist es wünschenswert, dann nicht immer wieder diverse Aktionen wiederholen zu müssen, sondern auf dem Bestehenden aufsetzen zu können – vermutlich bewegt man sich dann eher im Bereich Integrationstests, aber auch diese lassen sich sehr gut mit JUnit erstellen.

Vor JUnit 5 musste man mit einigen Tricks arbeiten, um eine definierte Reihenfolge der Ausführung erreichen zu können. Wieso ist das so? JUnit ermittelt die Testmethoden mithilfe von Reflection (im Speziellen per `getMethods()`). Die Reihenfolge der so gelieferten Methoden hängt jedoch von der jeweiligen JVM-Implementierung ab. Als Abhilfe gab es in JUnit 4 folgenden Trick mit der Annotation `@FixMethodOrder`:

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

Auf diese Weise wurden die Methoden in alphabetischer Reihenfolge ihrer Namen ausgeführt.

JUnit 5 bietet zwei Möglichkeiten. Zunächst kann man in Anlehnung an JUnit 4 die alphabetische Variante wählen:

```
@TestMethodOrder(Alphabetic.class)
public class FixedMethodOrderJUnit4Test
{
    @Test
    public void A_createPerson()
    {
        // create Person Mike
    }

    @Test
    public void B_updatePerson()
    {
        // access and modify Person Mike
    }

    @Test
    public void C_deletePerson()
    {
        // delete Person Mike
    }
}
```

Möchte man die Reihenfolge nicht per Methodenname festlegen, so kann man dies auch mithilfe der Annotation `@Order` wie folgt realisieren:

```
@TestMethodOrder(OrderAnnotation.class)
class MethodOrderJuni5Tests
{
    @Test
    @Order(1)
    void createPerson()
    {
        // create Person Mike
    }

    @Test
    @Order(2)
    void updatePerson()
    {
        // access and modify Person Mike
    }

    @Test
    @Order(3)
    void deletePerson()
    {
        // delete Person Mike
    }
}
```

Interaktion mit temporärem Dateisystem mit `@TempDir`

Mitunter benötigt man in Unit Tests zum Prüfen von Funktionalität auch Zugriffe auf das Dateisystem. Praktischerweise kann JUnit jeweils eigene temporäre Verzeichnisse für einen Testfall bereitstellen, wobei abschließend automatisch Aufräumarbeiten erfolgen. Dadurch lässt sich die Forderung nach Unabhängigkeit der Testfälle untereinander einhalten. Zudem kann man Tests oftmals klarer formulieren, als dies mithilfe der später in Abschnitt 20.5.1 vorgestellten Stellvertreterobjekte möglich ist. Hierzu bietet JUnit 5 die Annotation `@TempDir`, die entweder ein `File` oder ein `Path` repräsentieren kann. Dadurch lässt sich dann das bekannte API verwenden.

Zur Demonstration von `@TempDir` legen wir nun ein Unterverzeichnis sowie eine Datei in einem temporären Verzeichnis an und prüfen dessen Inhalt sowie die Existenz der Datei:

```
import org.junit.jupiter.api.io.TempDir;

public class TempDirTest
{
   @Test
   public void testUsingTempFolder(@TempDir File folder) throws IOException
   {
      // Ermittle den (noch leeren) Inhalt des Temp-Folders, normales File-API
      final String[] dirContentsInitial = folder.list();
      assertEquals(0, dirContentsInitial.length);

      // Erzeuge Verzeichnis und Datei mit normalem File-API
      final File subFolder = new File(folder, "subfolder");
      subFolder.mkdir();
      final File createdFile = new File(folder, "abc.txt");
      createdFile.createNewFile();

      // Ermittle den Inhalt des Temp-Folders, normales File-API
      final String[] dirContents = folder.list();

      // Prüfe, ob File existiert und Verzeichnisinhalt übereinstimmt
      // Als Menge, damit die Reihenfolge keinen Einfluss hat
      assertTrue(createdFile.exists());
      assertEquals(Set.of("subfolder", "abc.txt"), Set.of(dirContents));
   }
}
```

Listing 20.11 *Ausführbar als* **'TEMPDIRTEST'**

Hinweis: Interaktionen mit dem Dateisystem in Unit Tests

Wenn lediglich wenige und vor allem kleinere Interaktionen mit dem Dateisystem zu testen sind, ist diese Erweiterung praktisch, weil sie von den ansonsten benötigten Aufräumarbeiten, z. B. dem Löschen von temporär erzeugten Dateien und Verzeichnissen, befreit. Im Allgemeinen sollte man Dateiaktionen in Unit Tests jedoch besser vermeiden. Derartiges fällt eher in den Bereich Integrationstests.

20.4 Parametrierte Tests mit JUnit 5

Wir haben bereits beim Test der Rabattberechnung festgestellt, dass man teilweise eine Vielzahl an Wertebelegungen testen muss und dass sich die Testklasse ziemlich aufbläht und unübersichtlich wird, sofern man für jede Wertebelegung eine eigene Testmethode erstellt. Um das eleganter zu lösen, gibt es verschiedene Varianten. Alle davon haben ihre spezifischen Stärken und Schwächen. In diesem Unterkapitel schauen wir uns die Thematik genauer an. Dazu starten wir mit einführenden Beispielen, die dann immer komplexer werden und schließlich in einer kleinen, aber feinen Sammlung von Tipps und Tricks für die Praxis münden.

20.4.1 Einstieg

Nachfolgend nehmen wir an, es sollen Berechnungen für festdefinierte Wertebereiche oder für eine ausgesuchte Menge an Eingaben geprüft werden.[12]

Einfache Variante: Selbstgestrickte parametrierte Tests

Nehmen wir an, wir sollen für alle Eingabewerte von einstelligen gültigen hexadezimalen Zahlen, repräsentiert als ASCII, den entsprechenden Zahlenwert durch Aufruf einer Methode `hexCodedByteToInt()` ermitteln. Eine einfache Variante ist es, die Verarbeitung der Eingabewerte mithilfe eines Arrays erwarteter Werte in einer Schleife zu prüfen:

```
@Test
public void testHexCodedByteValueToInt_With_AllValidInputs()
{
    final byte[] inputs = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                            'A', 'B', 'C', 'D', 'E', 'F' };
    final int[] expected = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                             10, 11, 12, 13, 14, 15 };

    for (int i = 0; i < inputs.length && i < expected.length; i++)
    {
       assertEquals(expected[i], Controller.hexCodedByteToInt(inputs[i]));
    }
}
```

Diese Art der Realisierung ist leicht verständlich und übersichtlich. ***Allerdings besteht hier der Nachteil, dass die Testmethode beim ersten Fehler abgebrochen wird***. Enthält die Funktionalität mehrere Fehler, so muss man sich sukzessive durch immer weitere Fehler kämpfen, bis der Testfall (endlich) bestanden wird. Zudem erfolgt die Zuordnung zwischen Eingabewert und Resultat aufgrund der Position – das wird schnell unübersichtlich.

[12] Bei einer (sehr) großen Anzahl an Werten ist es keine gute Idee, für alle diese explizit eine Prüfung vorzunehmen, da dies die Ausführungsdauer der Unit Tests oftmals verlängert – ohne jedoch einen (größeren) Mehrwert zu bringen. Hier empfiehlt sich der Einsatz von Repräsentanten aus Äquivalenzklassen, wodurch sich die Zahl der Testfälle drastisch reduzieren sollte.

Einführung in parametrierte Tests mit JUnit 5 und `@ValueSource`

Die Möglichkeit der Parameterübergabe für Testmethoden ist insbesondere auch für die parametrierten Tests von Interesse. In Abschnitt 2.4.2 haben wir diese kennengelernt und gesehen, dass die Definition von parametrierten Tests sich mit JUnit 5 einfach gestaltet. Schauen wir uns das nun genauer an. Zum Einstieg betrachten wir den Fall, dass wir lediglich Parameter für unsere Testmethoden vorgeben wollen, aber kein Ergebnis. Das ist für solche Testfälle praktisch, die nur eine Bedingung prüfen, etwa ob ein String nicht leer ist oder eine Zahl eine Primzahl darstellt. Einen Unit Test realisieren wir mithilfe der Annotations `@ParameterizedTest` und `@ValueSource` wie folgt:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

public class FirstParametrizedTest
{
    @ParameterizedTest(name = "run {index}: ''{0}'' is not empty")
    @ValueSource(strings = { "Tim", "Tom", "", "Mike" })
    void isNotEmpty(String value)
    {
        assertFalse(value.isEmpty());
    }

    @ParameterizedTest(name = "run {index}: {0} is a prime")
    @ValueSource(ints = {1, 2, 3, 4, 5, 6, 7})
    void ensureIsPrime(int value)
    {
        assertTrue(MathUtils.isPrime(value));
    }
}
```

Rekapitulieren wir: Im Attribut `name` kann man für `@ParameterizedTest` einen String mit Platzhaltern verwenden, der bei der Testausführung angezeigt wird. Dabei bedeuten die Platzhalter Folgendes: `{index}` entspricht dem Index in den Testdaten, und `{0}`, `{1}`, `{2}` usw. referenzieren die Parameter bzw. die entsprechenden Datenelemente – häufiger wird es auch mehrere Eingaben geben, wie wir es im Anschluss sehen werden. Zudem muss der Testgenerator wissen, welche Eingaben zu prüfen sind. Hier wird das per `@ValueSource` mit den Spezialisierungen für Strings bzw. `int`-Werte spezifiziert. Zudem gibt es vordefinierte Varianten für `longs` und `doubles`. Werfen wir kurz noch ein Blick auf die Primzahlberechnung:

```
public static boolean isPrime(int n)
{
    if (n <= 1)
        return false;
    if (n == 2)
        return true;

    for (int i = 2; i < n; i++)
    {
        if (n % i == 0)
            return false;
    }
    return true;
}
```

Schauen wir uns an, wie sich das Ganze auswirkt: Für jeden angegebenen Parameter wird ein eigener Testfall erstellt und ausgeführt. Wie das dann in Eclipse aussieht, zeigt Abbildung 20-10. Bedenken Sie bitte, dass zu Demonstrationszwecken ein paar Testfehler eingebaut wurden.

Abbildung 20-10 *Parametrierte Testfälle des Programms* FIRSTPARAMETERIZEDTEST

Praxisnähere parametrierte Tests mit `@CsvSource`

Abweichend zu den beiden einfachen Beispielen aus der Einführung benötigt man in der Praxis bei fast allen Tests ein Tupel aus Eingaben und Ergebnissen. Dabei hilft die Annotation `@CsvSource`. Hier kann man jeweils in sich geschlossene kommaseparierte Angaben für die gewünschten Kombinationen der jeweiligen Eingaben bzw. Parameter für die Testmethode machen. Sinnvollerweise repräsentiert der erste oder bevorzugt der letzte dann das erwartete Ergebnis – nutzt man den letzten Parameter, so folgt dies besser der (europäischen) Denkweise von links nach rechts.

Nachfolgend zeige ich eine mögliche Parametrierung für das Umkehren eines Strings:

```
@ParameterizedTest(name = "reverse({0}) => {1}")
@CsvSource({ "ABCD, DCBA", "OTTO, OTTO", "PETER, RETEP" })
void testReverse(final String input, final String expectedOutput)
{
    final String result = new StringBuilder(input).reverse().toString();

    assertEquals(expectedOutput, result);
}
```

Ein weiteres Beispiel ist die Addition zweier Werte mitsamt des erwarteten Ergebnisses. Zur Parametrierung nutzen wir Tripel von Werten. Hier erkennt man, dass die textuellen Werte automatisch in die von den Parametern benötigten Typen konvertiert werden:

```
@ParameterizedTest(name = "{index}: {0} + {1} = {2}")
@CsvSource({ "1, 1, 2", "2, -2, 0", "3, 4, 7" })
void testAdd(int first, int second, int expected)
{
    int sum = first + second;

    assertEquals(expected, sum);
}
```

Implizite Parameterkonvertierungen

Wir haben in den Beispielen schon implizit die Konvertierungen der als Strings in den Annotations angegebenen Werte in die jeweils benötigten Typen in der Signatur der Testmethode gesehen und einfach so intuitiv genutzt.

Schauen wir uns noch ein paar weitere Beispiele an, zunächst einmal die Umwandlung in `LocalDate`-Objekte:

```
@ParameterizedTest
@CsvSource({ "2014-03-15, 2014-01-01", "2014-06-16, 2014-04-01",
             "2014-09-15, 2014-07-01", "2014-11-15, 2014-10-01"})
void adjustToFirstDayOfQuarter(LocalDate startDate, LocalDate expected)
{
    final Temporal result = new FirstDayOfQuarter().adjustInto(startDate);

    assertEquals(expected, result);
}
```

Nachfolgend sehen wir dann die automatische Konvertierung von textuellen Monatsnamen in Objekte vom Typ `Month` – als kleine Besonderheit sind hier die Werte durch ':' voneinander getrennt, was wir durch die Angabe von `delimiter` spezifizieren. Bewusst habe ich hinter dem Trennzeichen in den Werteangaben mal Leerzeichen hinzugefügt bzw. weggelassen, um zu zeigen, dass diese praktischerweise bei der Extraktion der Werte ohne Bedeutung sind – für eine gegebenenfalls optisch bessere Trennung können sie aber eine entscheidende Rolle spielen.

```
@ParameterizedTest
@CsvSource(value = { "APRIL: 30", "JUNE : 30", "JULY:31" }, delimiter = ':')
void checkLengthOfMonth(Month month, int length)
{
    assertEquals(length, month.length(false));
}
```

Explizite Parameterkonvertierung mit `ArgumentConverter`

Bei der Angabe innerhalb einer `@CsvSource` sind wir bezüglich der Typen der Werte leicht eingeschränkt. Wir werden noch verschiedene Abhilfen kennenlernen. Nachfolgend nutzen wir als Grundlage das Interface `ArgumentConverter` aus dem Package `org.junit.jupiter.params.converter`, das wie folgt definiert ist:

```
public interface ArgumentConverter
{
    Object convert(Object input, ParameterContext context)
           throws ArgumentConversionException;
}
```

Um eine Konvertierung zu realisieren, muss man lediglich das Interface implementieren. Zum Einstieg in die Thematik wollen wir textuelle Hexadezimalzahlen in den Typ `int` wandeln und an die Testmethode übergeben. Dazu schreiben wir uns folgenden Konverter. In der `convert()`-Methode wird auf Basis des Eingabewerts `source` die Konvertierung ausgeführt, hier ist es eine einfache Umwandlung mit `Integer.parseInt()`:

```
class HexToInt implements ArgumentConverter
{
    @Override
    public Object convert(Object source, ParameterContext context)
           throws ArgumentConversionException
    {
        try
        {
           return Integer.parseInt((String) source, 16);
        }
        catch (final NumberFormatException e)
        {
           throw new ArgumentConversionException("invalid value: " + source, e);
        }
    }
}
```

Verbleibt noch die Frage, wie man diesen Konverter in der Testmethode nutzt. Dazu gibt es die Annotation `@ConvertWith`, die wir folgendermaßen einsetzen (den Binärzahlkonverter `BinaryToInt` finden Sie in den Sourcen zum Buch):

```
@ParameterizedTest
@CsvSource({ "F, 15", "10, 16", "AA, 170", "FF, 255" })
void convertWithCustomHexConverter(@ConvertWith(HexToInt.class) int input,
                                   int expected)
{
    assertEquals(expected, input);
}

@ParameterizedTest
@CsvSource({ "1, 1", "10, 2", "111, 7", "11111111, 255" })
void convertWithCustomBinaryConverter(@ConvertWith(BinaryToInt.class) int input,
                                      int expected)
{
    assertEquals(expected, input);
}
```

Umfangreichere Datenbereitstellungen mit `@CsvFileSource`

Mitunter nimmt die Anzahl der in `@CsvSource` anzugebenden Werte durch Erweiterung um Testfälle mit der Zeit zu. Schauen wir uns als Beispiel die Konvertierung von römischen in arabische Zahlen an:

```
@ParameterizedTest(name = "fromRomanNumber(''{1}'') => {0}")
@CsvSource({ "1, I", "2, II", "3, III", "4, IV", "5, V", "7, VII",
             "9, IX", "17, XVII", "40, XL", "90, XC", "400, CD",
             "444, CDXLIV", "500, D", "900, CM", "1000, M",
             "1666, MDCLXVI", "1971, MCMLXXI", "2018, MMXVIII",
             "2019, MMXIX", "2020, MMXX", "3000, MMM" })
@DisplayName("Konvertiere römische in arabische Zahl")
void fromRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.fromRomanNumber(romanNumber);

    assertEquals(arabicNumber, result);
}
```

Es hat sich schon einiges an Werten angesammelt. Dadurch wird das Ergänzen von Testfällen doch leicht unhandlich. Erschwerend kommt hinzu, dass wir eventuell auch eine Rückabbildung implementieren und dann die Werte für andere Testmethode(n) nochmals angeben müssten. Obwohl ein kleines Maß an Sourcecode-Duplikation in Tests durchaus zu tolerieren ist, wenn dies die Lesbarkeit und Verständlichkeit verbessert, wäre die Grenze hier dann doch überschritten.

JUnit 5 bietet als Abhilfe das Einlesen von Testdaten aus einer CSV-Datei mit `@CsvFileSource`:

```
@ParameterizedTest(name = "fromRomanNumber(''{1}'') => {0}")
@CsvFileSource(resources = "arabicroman.csv", numLinesToSkip = 1)
@DisplayName("Konvertiere römische in arabische Zahl")
void fromRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.fromRomanNumber(romanNumber);

    assertEquals(arabicNumber, result);
}
```

Die CSV-Datei sieht dann (auszugsweise) wie folgt aus:

```
arabic,roman
1, I
2, II
3, III
4, IV
5, V,
7, VII
9, IX
17, XVII
...
```

Parametrierte Tests mit `@MethodSource`

Es verbleibt noch eine potenzielle Schwierigkeit bei der Angabe der Werte. Manchmal wird die textuelle Angabe der Werte unübersichtlich oder ist vielleicht sogar nicht möglich. Das gilt etwa für Arrays oder Listen, nachfolgend für eine Liste von Eingabewerten sowie ein erwartetes Ergebnis gezeigt. Dafür bietet sich eine andere Art der Bereitstellung der Daten in Form eines `Stream<Arguments>` an, der von einer statischen Methode zurückgeliefert wird. Den Methodennamen spezifiziert man dann mit `@MethodSource` – leider ist dies nur textuell möglich, aber trotzdem sehr intuitiv.

```
@ParameterizedTest(name = "removeDuplicates({0}) = {1}")
@MethodSource("listInputsAndExpected")
void removeDuplicates(List<Integer> inputs, List<Integer> expected)
{
    List<Integer> result = ListRemove.removeDuplicates(inputs);

    assertEquals(expected, result);
}

static Stream<Arguments> listInputsAndExpected()
{
    return Stream.of(Arguments.of(List.of(1, 1, 2, 3, 4, 1, 2, 3),
                                  List.of(1, 2, 3, 4)),
                     Arguments.of(List.of(1, 3, 5, 7),
                                  List.of(1, 3, 5, 7)),
                     Arguments.of(List.of(1, 1, 1, 1),
                                  List.of(1)));
}
```

Schauen wir uns noch ein Beispiel an, nämlich den Test einer selbstgeschriebenen Variante von Insertion Sort für ein `int[]`. Zum Test benötigen wir die Aufbereitung von Arrays als Eingabe- und Erwartungswert:

```
@ParameterizedTest
@MethodSource("inputAndExpectedIntArrays")
void testInsertionSort(int[] input, int[] expected)
{
    int[] result = InsertionSort.sort(input);

    assertArrayEquals(expected, result);
}
```

Wie sieht nun die Methode `inputAndExpectedIntArrays()` aus? Nun, diese definiert lediglich die Kombinationen zu übergebender Werte als `int[]` und stellt diese dann als `Stream<Arguments>` bereit:

```
private static Stream<Arguments> inputAndExpectedIntArrays()
{
    int[] values1 = { 1, 3, 2, 4 };
    int[] expected1 = { 1, 2, 3, 4 };

    int[] values2 = { 1, 4, 2, 6, 5, 7, 2 };
    int[] expected2 = { 1, 2, 3, 4, 5, 6, 7 }

    return Stream.of(Arguments.of(values1, expected1),
                     Arguments.of(values2, expected2));
}
```

Ganz nebenbei lernen wir noch einen Implementierungstrick kennen, der für mehr Lesbarkeit sorgen kann. Im vorherigen Beispiel wurden die Werte direkt in `Stream.of()` angegeben, was nicht ganz so gut lesbar und intuitiv ist. Je mehr Parameter oder Wertekombinationen zu testen sind, desto mehr gewinnt die hier gezeigte explizite Definition von Variablen an Klarheit – von trivialen Eingaben einmal abgesehen.

20.4.2 Verbesserung des Tests der Rabattberechnung

Zuvor hatten wir in Abschnitt 20.2.2 schon einiges zur Verbesserung der Lesbarkeit und Reduktion der Testmethoden zur Prüfung der Rabattberechnung unternommen. Allerdings verblieben noch ein paar Details, auf die ich nun eingehen möchte. Durch die Angabe der Werte als textuelle Informationen in `@CsvSource` wurden wieder mögliche Inkonsistenzen, eine gewisse Fragilität und schwierigere Zuordnung zu Rabatten eingeführt. Zur Erinnerung ist hier noch mal der Test abgebildet:

```
public class DiscountCalculatorV2ParameterizedTest
{
    private final DiscountCalculatorV2 calculator = new DiscountCalculatorV2();

    @ParameterizedTest(name = "{index}: amount {0} => discount {1}")
    @CsvSource({ "20, 0", "200, 4", "2000, 7" })
    void testCalcDiscount_EquivalenceClasses(int amount, int expectedDiscount)
    {
        assertEquals(expectedDiscount, calculator.calcDiscount(amount));
    }

    @ParameterizedTest(name = "{index}: amount {0} => discount {1}")
    @CsvSource({ "49, 0", "50, 4", "51, 4",
                 "999, 4", "1000, 4", "1001, 7" })
    void testCalcDiscount_CornerCases(int amount, int expectedDiscount)
    {
        assertEquals(expectedDiscount, calculator.calcDiscount(amount));
    }
}
```

Listing 20.12 *Ausführbar als* **'DISCOUNTCALCULATORV2PARAMETERIZEDTEST'**

Verbesserung der Implementierung

Unser Ziel ist es aber, sowohl die Vorteile der parametrierten Tests als auch ein sauberes objektorientiertes Design zu nutzen. Zunächst arbeiten wir an Letzterem, indem wir eine `enum`-Aufzählung für die Rabatte wie folgt definieren:

```
public enum Discount
{
    NO_DISCOUNT(0), MEDIUM_DISCOUNT(4), BIG_DISCOUNT(7);

    public final int amount;

    Discount(final int amount)
    {
        this.amount = amount;
    }
```

```
    @Override
    public String toString()
    {
        return name() + "(" + amount + ")";
    }
}
```

Danach nutzen wir dies in der Rabattberechnung folgendermaßen:

```
public class DiscountCalculatorV4
{
    public Discount calcDiscount(final int count)
    {
        Preconditions.checkArgument(count >= 0, "count should not be negative");

        if (count < 50)
            return Discount.NO_DISCOUNT;
        if (count >= 50 && count <= 1000)
            return Discount.MEDIUM_DISCOUNT;
        if (count > 1000)
            return Discount.BIG_DISCOUNT;

        throw new IllegalStateException("programming problem: should never " +
                        "reach this line. value " + count + " not handled!");
    }
}
```

Verbesserung des Tests

Machen wir uns an das Umgestalten des parametrierten Tests. Anstatt die Werte textuell und ohne Typ mit einer `@CsvSource` anzugeben, verwenden wir nun je eine `@MethodSource`. Zur Bereitstellung der Daten implementierten wir zwei Methoden. Insgesamt werden die Testmethoden an sich sehr einfach und man kann sich auf die Wertekombinationen konzentrieren:

```
public class DiscountCalculatorV4Test
{
    private final DiscountCalculatorV4 calculator = new DiscountCalculatorV4();

    @ParameterizedTest(name = "{index}: amount {0} => discount {1}")
    @MethodSource("amountAndDiscount")
    void testCalcDiscount_EquivalenceClasses(int amount,
                                             Discount expectedDiscount)
    {
        Discount result = calculator.calcDiscount(amount);

        assertEquals(expectedDiscount, result);
    }

    @ParameterizedTest(name = "{index}: amount {0} => discount {1}")
    @MethodSource("cornerCasesAndDiscount")
    void testCalcDiscount_CornerCases(int amount,
                                      Discount expectedDiscount)
    {
        Discount result = calculator.calcDiscount(amount);

        assertEquals(expectedDiscount, result);
    }
```

```
    private static Stream<Arguments> amountAndDiscount()
    {
        return Stream.of(Arguments.of(20, Discount.NO_DISCOUNT),
                         Arguments.of(200, Discount.MEDIUM_DISCOUNT),
                         Arguments.of(2000, Discount.BIG_DISCOUNT));
    }

    private static Stream<Arguments> cornerCasesAndDiscount()
    {
        return Stream.of(Arguments.of(49, Discount.NO_DISCOUNT),
                         Arguments.of(50, Discount.MEDIUM_DISCOUNT),
                         Arguments.of(51, Discount.MEDIUM_DISCOUNT),
                         Arguments.of(999, Discount.MEDIUM_DISCOUNT),
                         Arguments.of(1000, Discount.MEDIUM_DISCOUNT),
                         Arguments.of(1001, Discount.BIG_DISCOUNT));
    }

    @Test
    void testCalcDiscount_NegativeInput_ShouldThrowException()
    {
        assertThrows(IllegalArgumentException.class,
                     ()-> calculator.calcDiscount(-1));
    }
}
```

Durch ein paar kleine Umbaumaßnahmen sind sowohl der Applikationscode als auch der Test lesbarer und stützen sich nicht mehr auf Magic Numbers ab. Eine Änderung der Rabattprozente müsste nur an einer Stelle in der Klasse `Discount` erfolgen.

20.4.3 Praxisbeispiel: Berechnung in Testfall vereinfachen

Nachfolgend wollen wir uns anschauen, wie wir mithilfe der bisher kennengelernten Techniken eine Testmethode so umbauen können, dass sie immer eleganter und einfacher verständlich wird.

Ausgangsbasis

Schauen wir uns als Beispiel eine Berechnung aller Sonntage in einem Bereich gegeben durch zwei `LocalDate`s an:

```
@Test
void sundaysBetween()
{
    final LocalDate start = LocalDate.of(2020, 1, 1);
    final LocalDate end = LocalDate.of(2020, 3, 3);

    var sundays = SundayCalculator.allSundaysBetween(start, end);

    var expectedSundays = List.of(LocalDate.parse("2020-01-05"),
            LocalDate.parse("2020-01-12"), LocalDate.parse("2020-01-19"),
            LocalDate.parse("2020-01-26"), LocalDate.parse("2020-02-02"),
            LocalDate.parse("2020-02-09"), LocalDate.parse("2020-02-16"),
            LocalDate.parse("2020-02-23"));

    assertEquals(expectedSundays, sundays);
}
```

Eigentlich ist die Implementierung ganz passabel, aber die Angabe der erwarteten Werte wird durch die vielen Aufrufe von `LocalDate.parse()` unübersichtlich.

Verbesserung 1: Aufbereitung vereinfachen

Wir gestalten die Aufbereitung der Ergebniswerte mit dem Stream-API und einer passenden Hilfsmethode `toListOfLocalDates()` wie folgt um:

```
@Test
void sundaysBetweenImproved()
{
    final LocalDate start = LocalDate.of(2020, 1, 1);
    final LocalDate end = LocalDate.of(2020, 3, 1);

    var sundays = SundayCalculator.allSundaysBetween(start, end);

    var dates = List.of("2020-01-05", "2020-01-12", "2020-01-19", "2020-01-26",
                        "2020-02-02", "2020-02-09", "2020-02-16", "2020-02-23");

    var expectedSundays = toListOfLocalDates(dates);

    assertEquals(expectedSundays, sundays);
}

private List<LocalDate> toListOfLocalDates(List<String> dates)
{
    return dates.stream().map(day -> LocalDate.parse(day)).
                          collect(Collectors.toList());
}
```

Der Test ist nun klarer, allerdings noch stark auf genau die Eingaben und Ergebnisse festgelegt. Schöner wäre eine allgemeingültigere Variante.

Verbesserung 2: Umwandlung in parametrierten Test

Als weitere Verbesserung extrahieren wir die Eingaben und erwarteten Ergebnisse in eine Methode, sodass wir nicht nur auf eine Wertebelegung fixiert sind. Weil wir gerade dabei sind, ergänzen wir eine andere Konstellation von Eingaben und Ergebnissen:

```
@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndResults")
void sundaysBetweenV3(LocalDate start, LocalDate end,
                      List<String> expected)
{
    var sundays = SundayCalculator.allSundaysBetween(start, end);

    var expectedSundays = toListOfLocalDates(expected);

    assertEquals(expectedSundays, sundays);
}

private static Stream<Arguments> startAndEndDateAndResults()
{
    var args1 = Arguments.of(LocalDate.of(2020, 1, 1), LocalDate.of(2020, 3, 1),
                             List.of("2020-01-05", "2020-01-12", "2020-01-19",
                                     "2020-01-26", "2020-02-02", "2020-02-09",
                                     "2020-02-16", "2020-02-23"));
```

```
    var args2 = Arguments.of(LocalDate.of(2020, 4, 1), LocalDate.of(2020, 5, 1),
                            List.of("2020-04-05", "2020-04-12", "2020-04-19",
                                    "2020-04-26"));

    return Stream.of(args1, args2);
}
```

Mit dieser Verbesserung können wir zufrieden sein, da jedes Programmteil nun dem Motto »kurz, knapp, präzise« folgt.

Was könnte man noch verbessern? In der Testmethode erfolgt noch die Konvertierung aus einer `List<String>` in eine `List<LocalDate>`.

Verbesserung 3: Parameter mit Typ `List<LocalDate>`

Um den Testfall so einfach wie möglich zu schreiben, wäre es wünschenswert, wenn die erwarteten Ergebnisse als `List<LocalDate>` geliefert würden. Tatsächlich ist dies sehr leicht durch Verlagerung des Aufrufs `toListOfLocalDates()` wie folgt erreichbar:

```
@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndResultsV2")
void sundaysBetweenV3b(LocalDate start, LocalDate end,
                       List<LocalDate> expected)
{
    var sundays = SundayCalculator.allSundaysBetween(start, end);

    assertEquals(expected, sundays);
}

private static Stream<Arguments> startAndEndDateAndResultsV2()
{
    var args1 = Arguments.of(LocalDate.of(2020, 1, 1), LocalDate.of(2020, 3, 1),
                             toListOfLocalDates(List.of("2020-01-05",
                             "2020-01-12", "2020-01-19", "2020-01-26",
                             "2020-02-02", "2020-02-09", "2020-02-16",
                             "2020-02-23")));

    var args2 = Arguments.of(LocalDate.of(2020, 4, 1), LocalDate.of(2020, 5, 1),
                             toListOfLocalDates(List.of("2020-04-05",
                             "2020-04-12", "2020-04-19", "2020-04-26")));

    return Stream.of(args1, args2);
}
```

Als Alternative kann man auch einen Argumentkonverter schreiben und folgendermaßen nutzen:

```
@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndResults")
void sundaysBetweenV4(LocalDate start, LocalDate end,
     @ConvertWith(ToListOfLocalDatesConverter.class) List<LocalDate> expected)
{
    var result = SundayCalculator.allSundaysBetween(start, end);

    assertEquals(expected, result);
}
```

```
static class ToListOfLocalDatesConverter implements ArgumentConverter
{
    @Override
    public Object convert(Object source, ParameterContext context)
           throws ArgumentConversionException
    {
        final List<String> dates = (List<String>) source;
        return toListOfLocalDates(dates);
    }
}
```

Fazit

Schritt für Schritt haben wir eine Testmethode verbessert, sodass die Aussagekraft klar hervortritt und der eigentliche Test wenig nebensächliche Details enthält und leicht um beliebige Eingaben ergänzt werden kann.

20.4.4 Praxis-Trickkiste

Mittlerweile sollten Sie einen recht guten Überblick über und Eindruck von den Möglichkeiten parametrierter Tests mit JUnit 5 gewonnen haben. Die nachfolgenden Beispiele behandeln jeweils einen spezifischen Aspekt und zeigen, wie sie dabei helfen, die Problematik geeignet zu adressieren.

Trick: Parameterkonvertierung mit `ArgumentConverter`

Wir hatten bereits erkannt, dass man bei der Angabe innerhalb einer `@CsvSource` bezüglich der Typen der Werte ein wenig eingeschränkt ist. Zwar kann man sich des Öfteren mit einer `@MethodSource` helfen, etwa, wenn Arrays oder Listen als Parameter erforderlich sind.

Manchmal möchte man in der Testmethode die textuell vorliegenden Informationen als Objekt erhalten. Bekanntermaßen kann man sich dazu eines sogenannten `ArgumentConverter`s bedienen, was wir an einem simplen Beispiel für Hexadezimalzahlen schon kennengelernt haben.

Zum ersten Kennenlernen war das gut, aber wir wollen uns nun eine praxisnähere Konvertierung anschauen. Nehmen wir eine mit drei Attributen nicht allzu komplexe, aber zumindest auch nicht zu simple Datenbehälterrealisierung `SampleDto` wie folgt an – hier unter Einsatz des Java-14-Sprachfeatures `record` (vgl. Abschnitt 14.1.3).

```
record SampleDto(String info, LocalDate date, int count)
{
}
```

Implementierung des Konverters Zur Konvertierung müssen wir das Interface `ArgumentConverter` implementieren und aus einer Stringrepräsentation dann die Werte extrahieren und in ein `SampleDto` wandeln. Das realisieren wir folgendermaßen:

```
static class SampleDtoConverter implements ArgumentConverter
{
    @Override
    public Object convert(Object input, ParameterContext parameterContext)
        throws ArgumentConversionException
    {
        if (input instanceof SampleDto)
            return input;

        // Java 14: instanceof-Verbesserung
        if (input instanceof String strValue)
        {
            // Konvertierung aus Stringrepräsentation in SampleDto
            try
            {
                final String[] values = strValue.split("/");
                final LocalDate date = LocalDate.parse(values[1].trim());
                final int count = Integer.parseInt(values[2].trim());

                return new SampleDto(values[0].trim(), date, count);
            }
            catch (NumberFormatException | IndexOutOfBoundsException ex)
            {
                String message = input
                       + " is no correct string representation of a SampleDto.";
                throw new ArgumentConversionException(message, ex);
            }
        }
        throw new ArgumentConversionException(input + " is no valid SampleDto");
    }
}
```

Schauen wir uns zum Abschluss noch den entsprechenden Test an. Dort muss insbesondere der umzuwandelnde Parameter mit `@ConvertWith` annotiert werden. Zwar sieht man gut, dass aus der textuellen Eingabe ein `SampleDto` wird – hier ist der Vorteil noch nicht so ausgeprägt, da wir die Erwartungswerte alle einzeln ermitteln:

```
@ParameterizedTest
@CsvSource({ "Michael/1971-02-07/11, Michael, 1971-02-07, 11",
             "Peter/1973-06-06/9, Peter, 1973-06-06, 9"})
void testBla(@ConvertWith(SampleDtoConverter.class) SampleDto dto,
             String expectedInfo, LocalDate expectedDate, int expectedCount)
{
    assertEquals(expectedInfo, dto.info());
    assertEquals(expectedDate, dto.date());
    assertEquals(expectedCount, dto.count());
}
```

Trick: JSON als Eingabe

Basierend auf dem vorherigen Trick können wir die Konvertierung sogar auf JSON als Eingabeformat ausbauen. Aus einer JSON-Repräsentation und damit textuellen Angabe der Attribute einer Person wollen wir diese Werte in ein `Person`-Objekt konvertieren, um es in der Signatur der Testmethode zu verwenden. Zur Definition nutzen wir folgenden Record:

```
record Person(String name, LocalDate dateOfBirth, String city)
{
}
```

Um diese Informationen zu verarbeiten, schreiben wir folgenden Unit Test:

```
public class ArgumentConverterTest
{
   @ParameterizedTest
   @CsvSource(value = {
   "{ name:'Peter', dateOfBirth: '2012-12-06', homeTown : 'Köln'} | false",
   "{ name:'Mike', dateOfBirth: '1971-02-07', homeTown : 'Zürich'} | true" },
   delimiter = '|')
   void jsonPersonAdultTest(@ConvertWith(JsonToPerson.class) Person person,
                            boolean expected)
   {
       long age = ChronoUnit.YEARS.between(person.dateOfBirth(),
                                           LocalDate.now());

       assertEquals(expected, age >= 18);
   }
}
```

Bei der Angabe der Werte sehen wir noch eine Besonderheit. Normalerweise erwartet die `@CsvSource` ihre Werte als kommaseparierte Eingabe. Da nun aber Kommata auch im JSON zur Separierung von Attributen dienen, muss man hier auf ein anderes Trennzeichen ausweichen. In diesem Fall wählen wir '|'.

Implementierung des Konverters Schauen wir uns nun eine Konvertierung aus JSON an. Dazu definieren wir wieder einen Konverter, diesmal namens `JsonToPerson`, der sich auf die externe Bibliothek GSON und die Methode `fromJson()` abstützt. Zur Implementierung nutzen wir die Basisklasse `SimpleArgumentConverter` statt des Interface `ArgumentConverter`:

```
class JsonToPerson extends SimpleArgumentConverter
{
    private static final Gson gson = new GsonBuilder().
                                     registerTypeAdapter(LocalDate.class,
                                     new LocalDateAdapter()).create();

    @Override
    protected Person convert(Object source, Class<?> targetType)
    {
        return gson.fromJson((String) source, Person.class);
    }
}
```

Als Besonderheit muss man zur Verarbeitung von `LocalDate`-Objekten noch einen speziellen Adapter implementieren und registrieren. Trotz dieser Zusatzaufwände bleibt die gesamte Umsetzung aber recht kurz und gut verständlich:

```
final class LocalDateAdapter extends TypeAdapter<LocalDate>
{
    @Override
    public void write(final JsonWriter jsonWriter, final LocalDate localDate)
          throws IOException
    {
        jsonWriter.value(localDate.toString());
    }

    @Override
    public LocalDate read(final JsonReader jsonReader) throws IOException
    {
        return LocalDate.parse(jsonReader.nextString());
    }
}
```

Damit sich diese Klassen kompilieren lassen, fügen Sie bitte folgende Abhängigkeit in die `build.gradle`-Datei ein:

```
implementation group: 'com.google.code.gson', name: 'gson', version: '2.8.6'
```

Trick: Spezielle Testnamen

Nehmen wir an, wir hätten einen Komparator für Versionsnummern im Format $MAJOR.MINOR.PATCH$ geschrieben – dabei ist die Angabe von $PATCH$ optional. Allerdings soll der Rückgabewert kompatibel zur Methode `int compare(T, T)` aus dem Interface `Comparator<T>` sein, was allerdings nicht so gut menschenlesbar ist, da hier die Werte -1, 0 und 1 zum Einsatz kommen. Um die Tests besser nachvollziehen zu können, sollen stattdessen <, = und > genutzt werden, wie es nachfolgende Tabelle zeigt.

Version 1	Version 2	Resultat
1.11.17	2.3.5	<
2.1	2.1.3	<
2.3.5	2.4	<
3.1	2.4	>
3.3	3.2.9	>
7.2.71	7.2.71	=

Erinnern wir uns: Normalerweise werden die Namen der Testfälle aus der Parametrierung abgeleitet, können aber geeignet angepasst werden, wie wir dies auch schon genutzt haben. Nachfolgend verwenden wir einen zusätzlichen Parameter `hint`, um den Vergleich sprechend darstellen zu können.

```
@ParameterizedTest(name="''{0}'' {3} ''{1}''")
@CsvSource( {"1.11.17, 2.3.5, -1, <",
             "2.1, 2.1.3, -1, <",
             "2.3.5, 2.4, -1, <",
             "3.1, 2.4,   1, >",
             "3.3, 3.2.9, 1, >",
             "7.2.71, 7.2.71, 0, =" } )
void compareVersions(String v1, String v2, int expected, String hint)
{
    int result = VersionNumberComparator.compareVersions(v1, v2);

    assertEquals(expected, result);
}
```

Die folgende Abbildung zeigt die bessere Lesbarkeit und Verständlichkeit bei der Ausführung der obigen Testmethode.

Abbildung 20-11 *Verbesserte Nachvollziehbarkeit der Vergleiche*

Trick: Zusätzliche Informationen bereitstellen

Gerade haben wir gesehen, wie man den Testfalltext geeignet modifiziert. Es geht noch besser: Als Beispiel dient die Prüfung auf eine saubere geschachtelte Klammerung. Beim Fehlschlagen von Prüfungen soll ein ergänzender Hinweis bereitgestellt werden. Auch dazu nutzen wir wieder einen zusätzlichen Parameter `hint`:

```
@ParameterizedTest(name="checkBraces(''{0}'') -- Hinweis: {2}" )
@CsvSource({ "(()), true, ok",
             "()(), true, ok",
             "((()))(((())), false, nicht sauber geschachtelt",
             "((), false, keine passende Klammerung" })
void checkBraces(String input, boolean expected, String hint)
{
    boolean result = checkBraces(input);

    assertEquals(expected, result);
}
```

Schauen wir der Vollständigkeit halber auf die Implementierung der Klammerprüfung:

```
static boolean checkBraces(final String input)
{
    int openingCount = 0;
```

```
    for (int i = 0; i < input.length(); i++)
    {
        final char ch = input.charAt(i);
        if (ch == '(')
            openingCount++;
        else if (ch == ')')
        {
            openingCount--;
            if (openingCount < 0)
                return false;
        }
    }
    return openingCount == 0;
}
```

Führen wir die obige Testmethode aus, so werden die fehlschlagenden Testfälle mit einem hilfreichen Hinweis versehen, sodass wir die Fehler besser nachvollziehen können.

Abbildung 20-12 *Bereitstellung von Hinweisen in Testfallnamen*

Trick: Randfälle prüfen

Wir haben nun schon verschiedene Möglichkeiten zur Übergabe von Werten kennengelernt. Mitunter sollen auch ein Leerstring, eine Liste ohne Elemente oder gar eine `null`-Referenz als Eingaben genutzt werden. Das ist mit `@CsvSource` nicht möglich und mit `@MethodSource` unhandlich. Als Abhilfe gibt es die beiden Annotations `@NullSource` und `@EmptySource`, deren Einsatz Sie nachfolgend exemplarisch zur Prüfung eines Strings und einer Liste sehen:

```
@ParameterizedTest(name = "test ''{0}'' for null or empty")
@NullSource @EmptySource
void nullAndEmptyStrings(String str)
{
    assertTrue(str == null || str.isEmpty());
}

@ParameterizedTest(name = "test ''{0}'' for null or empty")
@NullSource @EmptySource
void nullAndEmptyLists(List<?> list)
{
    assertTrue(list == null || list.isEmpty());
}
```

Spezialfall Betrachten wir eine typische Datencontainerklasse `DateRange`, die einen Datumsbereich, dessen Grenzen vollständig angegeben sein oder aber einzeln fehlen dürfen, absichern soll – hier bewusst verkürzt ohne `equals()` usw. dargestellt:

```
static class DateRange
{
    // one of both can be null for open interval
    private final LocalDate from;
    private final LocalDate to;

    public DateRange(final LocalDate from, final LocalDate to)
    {
        if (from == null && to == null)
            throw new IllegalArgumentException("invalid range, two nulls");

        this.from = from;
        this.to = to;
    }

    // ...
}
```

Mit Java 14 schreiben wir diese Funktionalität mithilfe von `record` (vgl. Abschnitt 14.1.3) kürzer und vermutlich fehlerfrei sowie kontraktkonfrom folgendermaßen:

```
// one of both parameters allowed to be null for open interval
record DateRange(final LocalDate from, final LocalDate to)
{
    public DateRange
    {
        if (from == null && to == null)
            throw new IllegalArgumentException("invalid range, two nulls");
    }
}
```

Wir sollen nun einen Test schreiben, der folgende Fälle prüft: (`null`, dateTo), (dateFrom, `null`) und (`null`, `null`). Man ist vielleicht geneigt, das mit einem parametrierten Test in Form einer `@MethodSource` zu prüfen:

```
static Stream<Arguments> allCombinations()
{
    return Stream.of(Arguments.of(null, LocalDate.now(), null),
                     Arguments.of(LocalDate.now(), null, null),
                     Arguments.of(null, null, IllegalArgumentException.class));
}
```

Das Ganze erfordert Fallunterscheidungen und wird doch unübersichtlich, da zwei Fälle eine erlaubte Eingabe und einer einen Fehlerfall darstellen.

Demnach ist es natürlicher, diese Prüfung durch zwei Testmethoden ausführen zu lassen, da diese auch besser gültige und fehlerhafte Eingaben trennen. Interessanterweise kann man dann auf den parametrierten Test verzichten und das Ganze enorm vereinfachen – auch beim Testen ist dies ein erstrebenswertes Ziel:

```
@Test
void testInvalidDateRangeConstruction()
{
    final Executable action = () -> new DateRange(null, null);

    assertThrows(IllegalArgumentException.class, action);
}

@Test
void testValidDateRangeConstruction()
{
    assertAll(()-> assertNotNull(new DateRange(null, LocalDate.now())),
              ()-> assertNotNull(new DateRange(LocalDate.now(), null)));
}
```

20.5 Fortgeschrittene Unit-Test-Techniken

In diesem Unterkapitel betrachten wir zunächst mit dem zustandsbasierten und verhaltensbasierten Testen zwei grundsätzlich verschiedene Varianten des Testens. Zudem gehe ich auf mögliche Schwierigkeiten beim Unit-Testen ein. Als Abhilfe lernen wir zum einen die Technik EXTRACT AND OVERRIDE und zum anderen ***Stellvertreterobjekte***, auch ***Test-Doubles*** genannt, kennen. Vielen sind die Begriffe Stubs und Mocks sicher geläufiger. Beide leiten sich aus dem Englischen ab: »stub« = Stumpf, Stummel und »to mock« = nachahmen, vortäuschen. Durch den Einsatz von Stubs und Mocks wird es möglich, andere Komponenten für Testfälle zu ersetzen bzw. deren Verhalten zu simulieren. Anhand von Beispielen zeige ich, wie man mithilfe von Stubs und Mocks das Testen vereinfachen kann. Dabei lernen wir auch das populäre Mocking-Framework Mockito kennen. Abschließend wird kurz auf das Thema »Testen privater Methoden« eingegangen.

Zustandsbasiertes vs. verhaltensbasiertes Testen

In diesem Kapitel haben wir uns bislang mit dem zustandsbasierten Testen beschäftigt. Was heißt das genau? Gemäß dem AAA-Stil wird zunächst für den richtigen Kontext gesorgt (Arrange), dann die gewünschte, zu testende Funktionalität ausgeführt (Act) und schließlich das Ergebnis geprüft (Assert). Dazu werden verschiedene Attribute ausgelesen und mit erwarteten Werten verglichen. So natürlich und eingängig dieses Verfahren ist, so gibt es doch noch eine andere Möglichkeit: das verhaltensbasierte Testen. Dieses wird oftmals als etwas unintuitiver wahrgenommen – jedenfalls, wenn man damit beginnt. Hierbei geht es darum, die Interaktionen zu prüfen und nicht die konkret ausgelösten Zustandsänderungen.

Betrachten wir dazu ein bewusst einfaches, etwas künstliches Beispiel eines Datencontainers, der zur Datenhaltung eine `List<String>` verwendet:

```
public final class Members
{
    private final List<String> members;

    public Members(final List<String> persons)
    {
        this.members = persons;
    }

    public boolean registerMember(final String member)
    {
        return members.add(member);
    }

    public boolean deregisterMember(final String member)
    {
        return members.remove(member);
    }

    // ...
```

Beim verhaltensbasierten Testen betrachtet man die Interaktionen, also die aufgerufenen Methoden, statt Veränderungen im Objektzustand. Demnach ist man beispielsweise nicht daran interessiert, ob sich nach einem Aufruf der Methode `registerMember(String)` in der Liste die Anzahl der gespeicherten Elemente erhöht hat. Beim zustandsbasierten Testen würde man dagegen genau dies durch einen entsprechenden Aufruf einer `assertXyz()`-Methode nutzen.

Wie kann man aber feststellen, ob ein korrektes Verhalten vorliegt? Die Antwort ist verblüffend, obwohl naheliegend: ***Man ermittelt die für eine Aktion notwendige Abfolge von Methoden analog zu einem Ablauf in einem Sequenzdiagramm***. Nehmen wir an, es soll folgender Ablauf geprüft werden: Zunächst werden die beiden Stringobjekte "`Person1`" und "`Person2`" über einen Aufruf der Methode `registerMember(String)` gespeichert. Anschließend wird "`Person1`" über einen Aufruf von `deregisterMember(String)` wieder gelöscht. Aufgrund des beschriebenen Ablaufs ergeben sich die als Sequenzdiagramm in Abbildung 20-13 dargestellten Aufrufe.

Abbildung 20-13 *Testablauf Mock-Beispiel*

Als Erwartung kann man aufgrund der Methodenaufrufe an die Klasse `Members` und basierend auf dem Sequenzdiagramm formulieren, dass es zu zwei Aufrufen von `add(String)`, gefolgt von einem Aufruf von `remove(String)` auf der Liste kommt. Beim verhaltensbasierten Testen prüfen wir genau dies ab. Wir erstellen dazu ein Stellvertreterobjekt, das die Interaktionen protokolliert und es später ermöglicht, diese mit den Erwartungen abzugleichen. Nachfolgend erhalten wir einen Stellvertreter durch Aufruf von `mock()` und prüfen unsere Erwartungen durch Aufruf von `verify()`:

```
// Arrange
final List<String> mockedList = mock(List.class);
final Members members = new Members(mockedList);

// Act
members.registerMember("Person1");
members.registerMember("Person2");
members.deregisterMember("Person1");

// Assert
verify(mockedList).add("Person1");
verify(mockedList).add("Person2");
verify(mockedList).remove("Person1");
```

Bitte beachten Sie, dass im Gegensatz dazu beim zustandsbasierten Testen der resultierende Objektzustand mit nur "`Person2`" als registriertem Mitglied auf verschiedene Arten entstanden sein könnte – etwa nur durch einmaliges Hinzufügen und nicht durch die zuvor dargestellte Abfolge der Methodenaufrufe. Das wiederum kann problematisch sein, wenn es vor allem auf die Abfolge und nicht den Endzustand ankommt.

Schwierigkeiten beim Unit-Testen

Nachdem wir mittlerweile wissen, dass sich Unit Tests in der Regel positiv auf die Qualität auswirken, sollten wir uns noch folgende Frage stellen: »Was erschwert es uns, gute Unit Tests zu schreiben?«

Wenn man ein wenig nachdenkt, so sind es fast dieselben Dinge, die schon beim Entwickeln störend sein können. Rekapitulieren wir kurz, was in den bisherigen Kapiteln dazu gesagt wurde oder was Sie vielleicht schon am eigenen Leib erfahren haben. Bevor man mit dem Implementieren beginnt, sollte man bereits grob wissen, was zu realisieren ist und wie es funktionieren soll. Ansonsten stochert man im Dunkeln und Fehlentwicklungen sind vorprogrammiert. Unabhängig davon sind eine gute Namensgebung und ein problemangepasstes Design mit klaren Zuständigkeiten erstrebenswert. Oftmals sieht die Realität aber anders aus: Klassen bieten zu viel Funktionalität und besitzen meistens zu viele Abhängigkeiten zu anderen Klassen. Das wiederum führt zu Fragilität: Ändert man an einer Stelle, so zerbricht etwas an anderer Stelle.

Kommen wir nun zu Unit Tests zurück. Auch dort helfen uns Informationen zur gewünschten Funktionalität, um adäquate Testfälle erstellen zu können und nicht nur Pseudotests für triviale `get()`-/`set()`-Methoden zu schreiben. Zudem profitieren wir von gelungenen, aussagekräftigen Namen der Testmethoden, um das Wesentliche eines Testfalls erkennen zu können (ohne dazu die Implementierung anschauen zu müssen).

Darüber hinaus sollten Abhängigkeiten auf andere Klassen sinnvoll aufgelöst werden. Zu deren Substituierung und zur Erleichterung des Testens bieten sich verschiedene Formen von Test-Doubles an. Der Name ist recht plakativ und ähnelt dem Stunt-Double aus Filmen. Genau diese Analogie sollten Sie im Hinterkopf behalten: Ein Test-Double vertritt ein Anwendungsobjekt in einem Testfall, um Abhängigkeiten etwa zu einer anderen Klasse, zum Dateisystem oder zu Datenbanken aufzulösen und eine Unit besser für sich alleine testbar zu machen.

20.5.1 Stellvertreterobjekte / Test-Doubles

Wie eingangs erwähnt, bezeichnet man Objekte als Test-Doubles, die als Stellvertreter für Applikationsobjekte oder Fremdsysteme zur Erleichterung der Programmierung von Tests genutzt werden können. Man spricht hier auch von ***Kollaborateuren***. Mit diesen lässt sich Zustand oder Verhalten prüfen, ohne immer ein System mit seinen ganzen Bestandteilen initialisieren bzw. erzeugen zu müssen. Das ist hilfreich, weil Unit Tests nur kleine Ausschnitte aus dem Programmverhalten isoliert überprüfen sollen, d. h. ohne die Interaktion mit anderen Software- oder Hardwarekomponenten. Genau diese werden aber oftmals für die Bereitstellung der Programmfunktionalität benötigt, was wiederum die Testbarkeit erschwert. Durch den Einsatz von Abstraktionen für die Kollaborateure in Form von Test-Doubles kann man eine Unabhängigkeit von externen Systemen erreichen: Wurde dies (initial) nicht bedacht, so muss z. B. durch Einführen eines Interface für eine bessere Testbarkeit gesorgt werden. In den Testfällen nutzt man dann Test-Doubles, die das von der Anwendung benötigte Interface erfüllen. Dabei kommen die nachfolgend kurz vorgestellten Varianten zum Einsatz.[13]

Dummy

Unter einem ***Dummy*** versteht man einen Platzhalter, der (nahezu) keine Funktionalität bietet. Obwohl das zunächst wenig hilfreich scheint, erleichtert dies, Abhängigkeiten aufzulösen, etwa dann, wenn man an eine Methode gewisse Parameter eines speziellen Typs übergeben muss, deren Werte aber später in der Abarbeitung der Methode irrelevant sind. Im einfachsten Fall reicht eine `null`-Referenz, um die Parameterliste zu erfüllen. Bei `null`-Referenzen setzt das aber voraus, dass auf diese nicht ungeprüft zugegriffen wird, da ansonsten `NullPointerException`s drohen. Daher bietet es sich in der Regel an, Null-Objekte (vgl. Abschnitt 18.3.2) als Dummy zu verwenden.

Beispiel Die einfachste Variante eines Dummys ist nachfolgend gezeigt. Hier wird ein Parameter zum Aufruf der Methode `doSomething()` benötigt, der dort aber optional ist, somit kann hier `null` als Wert und simpelster Stellvertreter angegeben werden:

```
final Object optionalDataDummy = null;
doSomething(mandatoryValue, optionalDataDummy);
```

[13] Bitte beachten Sie, dass man verschiedenste abweichende Definitionen und Verwendungen der obigen Begriffe für Test-Doubles findet.

Stub oder Fake

Deutlich komplexer als Dummys sind ***Stub*** und Fake. Diese möchte ich in diesem Buch nicht groß unterscheiden. Für mich stellen ein Stub und ein Fake jeweils eine rudimentäre, manchmal auch nur leicht abgespeckte, aber funktionierende Implementierung einer anderen Klasse dar. Dabei kann ein Stub beispielsweise vordefinierte Rückgabewerte liefern, aber auch ein ausgeklügelteres Verhalten besitzen, wie Daten und Zustand speichern oder Methodenaufrufe protokollieren. Ein Fake wäre ein Exemplar eines Test-Doubles mit einer abgewandelten Funktionsweise, etwa eine In-Memory-Datenbank statt einer realen Datenbank.

Beispiel Betrachten wir Ausgabegeräte, für die mit dem Interface `IDisplay` eine Abstraktion existiert. Wir werden unabhängig von der realen Hardware, wenn wir eine Stub-Klasse `SimulationDisplay` implementieren, die das Interface `IDisplay` erfüllt. Statt Ausgaben auf einem physikalischen Gerät vorzunehmen, dient diese Testklasse dazu, eingehende Nachrichten vom Typ `MessageDto` entgegenzunehmen. Nachfolgend ist der dazu relevante Teile von `IDisplay` angegeben:

```
public interface IDisplay
{
    public void displayMsg(final MessageDto msg);
    ...
}
```

Eine einfache Variante der Implementierung als Stub könnte man wie folgt realisieren:

```
public final class SimulationDisplayStubBasic implements IDisplay
{
    @Override
    public void displayMsg(final MessageDto msg)
    {
        System.out.println("SimulationDisplay - got msg '" + msg + "'");
    }
}
```

Diese simple Ausprägung ist fast noch eher ein Dummy als ein Stub, insbesondere erlaubt sie nur rudimentäres Testen – die Ausgaben muss man aufwendig selbst prüfen.

Schöner wäre eine etwas ausgefeiltere Variante, die die Nachrichten in einer Liste speichert und für Unit Tests dann Zugriff in Form der Methode `getContents()` darauf bereitstellt, was weiter gehende Prüfungen in Tests ermöglicht. Den Stub realisieren wir wie folgt, wobei die Stub-spezifische Funktionalität mit `STUB` gekennzeichnet ist:

```
public final class SimulationDisplayStubAdvanced implements IDisplay
{
    private final List<MessageDto> messages = new ArrayList<>();

    @Override
    public void displayMsg(final MessageDto msg)
    {
        messages.add(msg);
    }
```

```
    // STUB
    public List<MessageDto> getContents()
    {
        return Collections.unmodifiableList(messages);
    }
}
```

Mock

Schließlich verbleiben ***Mocks***. Diese dienen zum Überprüfen von Verhalten in Form von erwarteten Methodenaufrufen. Werden im Testfall durch die Abarbeitung der Anwendungsfunktionalität nicht die für einen Mock zuvor spezifizierten Methoden aufgerufen, so wird dies als Fehler gewertet.

Beispiel Mocks kann man zwar selbst realisieren, das erfordert jedoch einiges an Arbeit, insbesondere, wenn in der Lage sein möchte, die Interaktionen ausgeklügelt festzulegen und zu prüfen. Für unser Beispiel bestünde ein simpler Mock nur daraus, mithilfe eines Flags zu prüfen, ob eine Methode aufgerufen wurde:

```
public final class SimulationDisplayMock implements IDisplay
{
    private boolean displayMsgCalled = false;

    @Override
    public void displayMsg(final MessageDto msg)
    {
        displayMsgCalled = true;
    }

    // MOCK
    public void verifyDisplayMsgWasCalled() throws AssertionError
    {
        PreConditions.checkState(displayMsgCalled,
                                 "method 'displayMsg' has not been called");
    }
}
```

Im Listing ist die Mock-spezifische Funktionalität mit `MOCK` gekennzeichnet. Damit könnte man aber nicht einmal herausfinden, wie häufig ein Aufruf erfolgt ist. Eine etwas ausgefeiltere Variante speichert die Aufrufe samt Parametern wie folgt:

```
public final class SimulationDisplayMock implements IDisplay
{
    private final Map<String, Object> calls = new LinkedHashMap<>();

    @Override
    public void displayMsg(final MessageDto msg)
    {
        calls.put("displayMsg", msg);
    }

    // MOCK
    public boolean verifyMethodWasCalled(final String methodName)
    {
        final boolean methodCallFound = calls.containsKey(methodName);
```

```
        PreConditions.checkState(methodCallFound,
                                 "method '" + methodName +
                                 "' has not been called");
    }

    // MOCK
    public int verifyInvocationCount(final String methodName,
                                     final int expectedCount)
    {
        final int actualMethodCalls = invocationCount(methodName);

        PreConditions.checkState(actualMethodCalls == expectedCount,
                                 "expected " + expectedCount + " calls to " +
                                 "method '" + methodName + "' but found " +
                                 actualMethodCalls + " calls");
    }

    // MOCK
    private int invocationCount(final String methodName)
    {
        return calls.keySet().stream().
                            filter(key -> key.equals(methodName)).
                            count();
    }
}
```

Unschwer erkennt man, dass sich die eigene Implementierung von Mocks bei zunehmender Anzahl zu beobachtender Methodenaufrufe und umfangreicheren Auswertungsmöglichkeiten immer aufwendiger gestaltet und die Komplexität schnell explodiert. Vermutlich wird man dabei auch Fehler machen und dadurch wird das Testen unzuverlässig. Deshalb bietet sich der Einsatz eines Mocking-Frameworks an. Später mehr dazu.

Kritische Betrachtung zustandsbasiertes vs. verhaltensbasiertes Testen

Rekapitulieren wir kurz: Beim zustandsbasierten Testen mit Stubs geht es darum, Zustand bzw. Resultate zu prüfen, beim verhaltensbasierten Testen mit Mocks sind die Interaktionen im Fokus. Manchmal erlaubt es aber erst die Kombination aus beiden, sinnvolle und prägnante Tests zu schreiben. Beleuchten wir kurz, warum dies so ist.

Um Zustandsänderungen prüfen zu können, werden teilweise kleinere Änderungen in den Business-Klassen erforderlich – das haben wir im Beispiel der Klasse `SimluationDisplayStubAdvanced` und der dort neu eingeführten Methode `getContents()` gesehen. Sofern solche Modifikationen minimal sind, nur lesend zugreifen oder vielleicht sogar zu besserem Design führen, profitiert man davon. Anderweitige Erweiterungen rein aufgrund von Testbedürfnissen sind dagegen möglichst zu vermeiden, weil dann Applikationscode existiert, der nicht zur Business-Funktionalität beiträgt, sondern lediglich zum Testen. Einige IDEs wie IntelliJ IDEA können diesen Sachverhalt prüfen und als Verstoß melden.

Wenn man Funktionalitäten gründlich, jedoch nur zustandsbasiert testen möchte, so ist das Einführen testbedingten Sourcecodes kaum zu vermeiden – wobei `get()`-Methoden dabei nicht wirklich problematisch sind. Allerdings gibt

es ein schwerwiegenderes Problem, auf das ich schon kurz hingewiesen habe: Anhand des geprüften Zustands kann man nicht eindeutig auf die Abfolge von Methodenaufrufen schließen. So hinterlassen die Kommandofolgen $add(x)$ und $add(x), add(y), remove(y)$ den gleichen beobachtbaren Zustand, nämlich den Wert x. Um Derartiges testen zu können, müssen wir auf Mocks zurückgreifen. Durch deren Einsatz können wir ohne Modifikationen an den Business-Klassen die erwarteten Aufrufe prüfen. Allerdings ist es rein mit Mocks nicht möglich, erwartete Zustandsänderungen nachzuvollziehen und sicherzustellen. Das wiederum erlauben Stubs.

20.5.2 Vorarbeiten für das Testen mit Stubs und Mocks

Wir haben nun ein erstes Verständnis für Test-Doubles und deren Vorteile aufgebaut. Um aber statt konkreter Kollaborateure korrespondierende Stellvertreter in Form von Stubs und Mocks nutzen zu können, ist es häufig notwendig, die in Tests verwendeten Komponenten über Interfaces oder abstrakte Klassen anzusprechen. Zudem leidet die Testbarkeit, wenn im Applikationscode konkrete Instanzen erzeugt werden. Damit man mit Test-Doubles arbeiten kann, muss man gegebenenfalls zunächst eine Abstraktion, immer jedoch eine Übergabe als Parameter (***Dependency Injection***) einführen. Das wollen wir nun noch einmal kurz nachvollziehen.

Refresher: Arten der Dependency Injection

Um direkte Abhängigkeiten aufzulösen, kann man die benötigten Typen entweder im Konstruktor oder in Methoden als Parameter übergeben, statt Instanzen der erforderlichen Typen selbst zu erzeugen. Man spricht in diesem Zusammenhang auch von ***Dependency Injection***, da Abhängigkeiten in die nutzende Klasse als Parameter injiziert werden. Je nach Variante nennt man dies ***Constructor Injection*** oder ***Method Injection***.

Ausgangssituation und Vorbereitungen zur Testbarkeit

Nachfolgend betrachten wir als Beispiel eine Utility-Klasse `CSVUtils`, die eine CSV-Datei mithilfe einer `FileInputStream`-Instanz und der Methode `readCSV()` einliest. Abbildung 20-14 zeigt das Klassendiagramm der Ausgangssituation mit den oben genannten Abhängigkeiten auf konkrete Klassen, die zudem im Applikationscode direkt einen `FileInputStream` durch die Klasse `CSVUtils` erzeugen.

Abbildung 20-14 *Einführen eines Stubs – Ausgangssituation*

Ich stelle nun Schritt für Schritt vor, wie man direkte Abhängigkeiten auflösen kann und wie durch das Einführen eines Stubs das Erstellen von Unit Tests erleichtert wird.

Schritt 1: Nutzen von Übergabeparametern Um die Abhängigkeit zwischen Utility-Klasse und der Streamklasse etwas zu lösen, wird zunächst ein Objekt der Klasse `FileInputStream` an die Methode `readCSV(FileInputStream)` übergeben (Method Injection) (vgl. Abbildung 20-15). Dadurch muss eine aufrufende Klasse, insbesondere auch der Unit Test, eine entsprechende Instanz bereitstellen.

Abbildung 20-15 *Einführen eines Stubs – Schritt 1*

Schritt 2: Lösen von Abhängigkeiten Um die Testbarkeit zu verbessern, sollte man statt der konkreten Klasse `FileInputStream` besser die abstrakte Basisklasse `InputStream` nutzen, wie dies Abbildung 20-16 zeigt. Existiert eine solche abstrakte Basisklasse oder ein adäquates Interface nicht, so ist es oftmals vorteilhaft, diese Abstraktion zu erzeugen, um diesen Schritt durchführen zu können.

Abbildung 20-16 *Einführen eines Stubs – Schritt 2*

Schritt 3: Erzeugen eines Test-Stubs Für Unit Tests wollen wir in der Regel nicht mit dem Dateisystem interagieren, sondern Daten aus einer beliebigen Testdatenquelle einlesen. Dazu wird ein Test-Double benötigt. Abbildung 20-17 zeigt, dass hier die Klasse `InputStreamStub` genutzt wird, die alle Methoden der gemeinsamen Basisklasse `InputStream` geeignet implementiert.

Abbildung 20-17 *Einführen eines Stubs – Schritt 3*

20.5.3 Die Technik EXTRACT AND OVERRIDE

Bei der Technik EXTRACT AND OVERRIDE geht es nur am Rande um Stubs als Kollaborateure, vor allem dient sie dazu, Sollbruchstellen in den Sourcecode einzufügen, um die Testbarkeit durch Ableiten und Überschreiben zu erreichen. Schauen wir uns dies nun genauer an.

Einführendes Beispiel

Die folgende Klasse `Calculator` nimmt in einer (nicht allzu clever realisierten) Methode `calc()` zwei Stringparameter entgegen, konvertiert diese in `int`-Werte und liefert deren Summe als Ergebnis zurück. Im Falle eines Fehlers beim Parsing wird eine Warnmeldung in einem Dialog präsentiert und dann eine Exception ausgelöst:

```
public class Calculator
{
    public int calc(final String strNum1, final String strNum2)
    {
        try
        {
            final int num1 = Integer.parseInt(strNum1);
            final int num2 = Integer.parseInt(strNum2);

            return num1 + num2;
        }
        catch (final NumberFormatException ex)
        {
            JOptionPane.showConfirmDialog(null, "Keine gültige Ganzzahl");
            throw new IllegalArgumentException("Keine gültige Ganzzahl");
        }
    }
}
```

Initiales Erstellen eines Unit Tests Wir wollen für die Klasse `Calculator` eine gute Basis an Testfällen erstellen. Bekanntermaßen sollte man zunächst die normale Funktionalität absichern. Idealerweise prüft man zudem ein paar Randfälle.[14] Ergänzend empfiehlt es sich, Fehlersituationen und die korrespondierende Fehlerbehandlung zu testen. Mit diesen Hinweisen im Hinterkopf entsteht folgende erste Variante eines Unit Tests, in dem sowohl eine einfache Addition, der Spezialfall gleicher Zahlen unterschiedlichen Vorzeichens als auch ein Aufruf mit ungültigen Zahlen geprüft werden:

```
public class CalculatorTest
{
    @Test
    public void testCalc_TwoNumbers_ShouldReturnSum()
    {
        final Calculator calculator = new Calculator();
        assertEquals(5, calculator.calc("2", "3"));
    }
```

[14] Bei einer Addition gibt es diese eigentlich nicht (außer durch den begrenzten Wertebereich eines `int` und Effekte beim Überlauf).

```
    @Test
    public void testCalc_WithEqualNumbersButDifferentSigns_ShouldReturn0()
    {
        final Calculator calculator = new Calculator();
        assertEquals(0, calculator.calc("7", "-7"));
    }

    @Test
    public void testCalc_IllegalInputs_ShouldRaiseException()
    {
        final Calculator calculator = new Calculator();

        final Executable calcInvalidInputs = () -> calculator.calc("a2", "b3");

        assertThrows(IllegalArgumentException.class, calcInvalidInputs);
    }
}
```

Wenn wir diesen Unit Test in der IDE oder in unserem automatisierten Build-Lauf ausführen, so erscheint eine Dialogbox ähnlich zu der in Abbildung 20-18.

Abbildung 20-18 *Fehlermeldungsbox*

Während eine Benutzerinteraktion beim manuellen Ausführen eines Unit Tests schon störend ist, so wird sie zu einem Showstopper in einem automatischen Build auf einem Continuous-Integration-Server. Der Build-Lauf wird dort unterbrochen, bis jemand den Dialog schließt. Also was nun? Überlegen wir kurz, was wir machen können.

Schritt 1: Extract Method Generell ist es immer eine gute Idee, an den richtigen und benötigten Stellen mit Abstraktionen zu arbeiten. Was bedeutet das konkret? Wie können wir die Anzeige der Dialogbox herauslösen? Die Antwort ist ziemlich einfach: Wir erstellen eine Methode `showWarning()`: Um das professionelle Entwickeln zu üben, wollen wir Refactorings nutzen. Zunächst markieren wir den Aufruf der Methode `showConfirmDialog()` und verwenden das Basis-Refactoring EXTRACT METHOD. Als Sichtbarkeit für die neue Methode `showWarning()` wählen wir `protected`. Allerdings entsteht eine parameterlose Methode mit fixer Warnmeldung. Um diesen Hinweistext flexibel übergeben zu können, müssen wir einen Parameter einführen. Dazu markieren wir in der neu entstandenen Methode den Hinweistext und nutzen das Basis-Refactoring INTRODUCE PARAMETER. Als Folge sollte die Klasse `Calculator` nun wie folgt aussehen:

```
public class Calculator
{
    public int calc(final String strNum1, final String strNum2)
    {
        try
        {
            final int num1 = Integer.parseInt(strNum1);
            final int num2 = Integer.parseInt(strNum2);

            return num1 + num2;
        }
        catch (final NumberFormatException ex)
        {
            showWarning("Keine gültige Ganzzahl");
            throw new IllegalArgumentException("Keine gültige Ganzzahl");
        }
    }

    protected void showWarning(final String message)
    {
        JOptionPane.showConfirmDialog(null, message);
    }
}
```

Diese Transformation scheint noch nicht sonderlich hilfreich. Wir werden im Anschluss aber sehen, warum genau dies eine entscheidende Verbesserung darstellt, die es uns erlaubt, den Unit Test ohne Interaktion formulieren zu können.

Es sei allerdings darauf hingewiesen, dass es nicht immer so direkt ersichtlich ist, wo das Problem liegt und wo die Sollbruchstelle eingefügt werden sollte.

Schritt 2: Override Die durch die Methode `showWarning()` geschaffene Sollbruchstelle nutzen wir nun, um eine Spezialisierung der Klasse `Calculator` direkt im entsprechenden Testfall geeignet anzupassen. Dazu setzen wir eine anonyme innere Klasse ein, die hier als Stub dient:

```
@Test
public void testCalc_IllegalInputs_ShouldRaiseException()
{
    final Calculator calculator = new Calculator()
    {
        protected void showWarning(final String message)
        {
            // JOptionPane.showConfirmDialog(null, message);
            throw new IllegalArgumentException("Keine gültige Ganzzahl");
        }
    };

    final Executable calcInvalidInputs = () -> calculator.calc("a2", "b3");

    assertThrows(IllegalArgumentException.class, calcInvalidInputs);
}
```

Der Unit Test benötigt durch diese Modifikation keine Interaktion mehr und das Problem ist einigermaßen behoben. Wieso nur einigermaßen? Schauen wir uns mögliche Schwierigkeiten an.

Schritt 3: Behandlung möglicher Schwierigkeiten Zunächst einmal könnte es sein, dass einige Aktionen zur Fehlerbehandlung ausgeführt werden sollen, etwa dass eine E-Mail versendet wird, falls ein Fehler auftritt. Zwar lässt sich auch diese Aktion in unserer Implementierung unterbinden, aber vielleicht wollen wir ja genau diese Aktion im Test prüfen.

Zudem: Was machen wir, wenn wir den Sourcecode der Klasse `Calculator` nicht im Zugriff haben? Dann könnten wir keine Sollbruchstelle einfügen. In diesem Fall können uns Mocking-Frameworks, allen voran das populäre Mockito, unterstützen. Betrachten wir kurz eine damit mögliche Abhilfe: Mockito benutzt die in Abschnitt 12.5 beschriebene Technik der Dynamic Proxies, um Test-Doubles zu erzeugen. Für diese kann man gewünschtes Verhalten vorgeben – Details dazu später. Hier geht es zunächst darum, eine Alternative zu der obigen Variante zu zeigen, die nicht voraussetzt, dass man den Sourcecode im Zugriff hat und diesen ändern kann:

```
@Test
public void testCalc_IllegalInputs_ShouldRaiseException_UsingMockito()
{
    // leere Implementierung erzeugen, tut nichts bei Methodenaufrufen und
    // liefert Defaultwerte
    final Calculator calculator = Mockito.mock(Calculator.class);

    // Verhalten für calc() vorgeben
    Mockito.when(calculator.calc("a2", "b3")).
            thenThrow(IllegalArgumentException.class);

    final Executable calcInvalidInputs = () -> calculator.calc("a2", "b3");

    assertThrows(IllegalArgumentException.class, calcInvalidInputs);
}
```

Durch Aufruf von `mock()` wird ein Dynamic Proxy erzeugt, der als Test-Double für die Klasse `Calculator` arbeitet. Das Test-Double besitzt zunächst kein Verhalten und liefert bei Methodenaufrufen entsprechende Defaultwerte zurück, etwa `false`, `0` oder `null`. Im Beispiel soll aber eine `IllegalArgumentException` für eine ungültige Eingabe ausgelöst werden. Dazu spezifizieren wir mit `when()` den Methodenaufruf samt Parametern und mit `thenThrow()` das Auslösen einer Exception.

Führen wir nun alle Tests einmal als Gradle-Task CALCULATORTEST_STEP3 aus. Dann erhalten wir ein beruhigendes Grün, was uns motiviert, beim Thema Unit Testing und externe Abhängigkeiten sowie Mocking später noch tiefer ins Detail zu gehen.

Abbildung 20-19 *Ergebnis eines Testlaufs*

Komplexeres Beispiel

Um die zuvor beschriebenen Techniken zu verinnerlichen, betrachten wir ein etwas komplexeres Beispiel: Die folgende Klasse `PizzaService` versendet bei Bestellungen eine SMS-Benachrichtigung – wobei vereinfachend der SMS-Versand durch die Darstellung eines Dialogs simuliert wird:

```
public class PizzaService
{
    private final SmsNotificationService notificationService;

    public PizzaService()
    {
        notificationService = new SmsNotificationService();
    }

    public void orderPizza(final String name)
    {
        notificationService.send("Pizza " + name + " wird in Kürze geliefert.");
    }
}

public class SmsNotificationService
{
    public void send(final String msg)
    {
        // SMS versenden ... hier durch einen Dialog repräsentiert
        JOptionPane.showConfirmDialog(null, msg);
    }
}
```

Wollten wir dieses Konstrukt testen, so ist dies ziemlich problematisch:

1. Sicherlich soll nicht bei jedem Testlauf eine SMS versendet werden. Oder stellen Sie sich vor, es würde ständig eine Abbuchung von einem Bankkonto vorgenommen. Dadurch könnten Kunden schnell sehr verärgert sein.
2. Auch das Darstellen eines Dialogs behindert die Automatisierung von Tests. Das haben wir bereits diskutiert.

Versuchen wir trotzdem einen Unit Test zu schreiben und erinnern uns an die ARRANGE-ACT-ASSERT-Struktur:

```
public class PizzaServiceTest
{
    @Test
    void orderPizza_should_send_notification()
    {
        // Arrange
        final PizzaService service = new PizzaService();

        // Act
        service.orderPizza("Diavolo");
        service.orderPizza("Surprise");

        // Assert: Was soll man hier prüfen ???
    }
}
```

An dem Test sieht man Verschiedenes: Mit zustandsbasiertem Testen (also dem Abfragen von Daten) lässt sich nicht prüfen, ob es zu einem Versand der Nachrichten gekommen ist oder nicht. Zudem bestehen die oben genannten Negativpunkte weiterhin. Was können wir also machen?

Um dem Problem Herr zu werden, muss man folgende Dinge tun: Wir müssen ...

1. eine Sollbruchstelle einführen,
2. eine Stub-Implementierung für den `SMSNotificationService` erstellen und
3. einige Anpassungen im Unit Test vornehmen.

Schritt 1: Sollbruchstelle einführen Wie schon zuvor führen wir eine Methode als Sollbruchstelle ein – hier die Methode `createNotificationService()`:

```
public class PizzaServiceV2
{
    private final SmsNotificationService notificationService;

    public PizzaServiceV2()
    {
        notificationService = createNotificationService();
    }

    public void orderPizza(final String name)
    {
        notificationService.send("Pizza " + name + " wird in Kürze geliefert.");
    }

    protected SmsNotificationService createNotificationService()
    {
        return new SmsNotificationService();
    }
}
```

Schritt 2: Stub-Implementierung erstellen Zum Protokollieren und Nachvollziehen der Änderungen erstellen wir folgenden Stub, der die zu versendenden Nachrichten in einer Liste speichert und diese für eine spätere Prüfung in Testfällen bereitstellen kann:

```
public class StubNotificationService extends SmsNotificationService
{
    private final List<String> messages = new ArrayList<>();

    public void send(final String msg)
    {
        messages.add(msg);
    }

    public List<String> getMessages()
    {
        return Collections.unmodifiableList(messages);
    }
}
```

Schritt 3: Anpassungen im Unit Test vornehmen Mit den zuvor realisierten Erweiterungen sind wir in der Lage, den Unit Test unabhängig vom SMS-Versand zu halten. Dazu nutzen wir EXTRACT AND OVERRIDE, injizieren damit unseren Stub und können dadurch im Assert-Teil verschiedene Prüfungen vornehmen:

```
public class PizzaServiceV2Test
{
    @Test
    public void orderPizza_should_send_sms()
    {
        // Arrange
        final StubNotificationService stub = new StubNotificationService();
        final PizzaServiceV2 service = new PizzaServiceV2()
        {
            @Override
            protected SmsNotificationService createNotificationService()
            {
                return stub;
            }
        };

        // Act
        service.orderPizza("Diavolo");
        service.orderPizza("Surprise");

        // Assert
        assertEquals(2, stub.getMessages().size());
        assertTrue(stub.getMessages().
                   contains("Pizza Diavolo wird in Kürze geliefert."));
        assertTrue(stub.getMessages().
                   contains("Pizza Surprise wird in Kürze geliefert."));
    }
}
```

Listing 20.13 *Ausführbar als* **'PIZZASERVICEV2TEST'**

Zwar haben wir die Abhängigkeiten durch den Stub gelöst, aber was ist an diesem Test noch unschön und insbesondere fragil? Momentan arbeiten die textuellen Vergleiche auf exakter Übereinstimmung, sodass die Tests bei jeder kleinen Änderung brechen, sofern man die Modifikationen nicht überall nachzieht. Was kann man dagegen tun? Als Abhilfe können wir den Service um folgende Hilfsmethode ergänzen:

```
/* private */ static String createNotificationMsg(final String name)
{
    return "Pizza " + name + " wird in Kürze geliefert.";
}
```

Damit wird der Test weitestgehend unabhängig von textuellen Änderungen und wir können die Asserts wie folgt umschreiben:

```
assertTrue(stub.getMessages().
           contains(PizzaServiceV2.createNotificationMsg("Diavolo")));
assertTrue(stub.getMessages(),
           contains(PizzaServiceV2.createNotificationMsg("Surprise")));
```

Eine kleine Unschönheit verbleibt allerdings. Da zwei Prüfungen mit `assertTrue()` erfolgen, ist die Zuordnung, warum ein Fehler vorlag, manchmal schwieriger. Das wird

als »Test Smell: Unangebrachtes `assertTrue()` und `assertFalse()`« in Abschnitt 20.6.1 thematisiert. Als Abhilfe gilt hier: Einfacher und lesbarer wird das Ganze, wenn wir die Erwartungswerte in einer Liste definieren und auch hier von der gerade erstellten Hilfsmethode profitieren und diese Werte gegen die im Stub gespeicherten vergleichen:

```
var expected = List.of(PizzaService.createNotificationMsg("Diavolo"),
                       PizzaService.createNotificationMsg("Surprise"));

assertEquals(expected, allValues);
```

20.5.4 Einstieg in das Testen mit Mocks und Mockito

Wir haben zuvor die Technik EXTRACT AND OVERRIDE recht genau kennengelernt und dabei gesehen, wie man von Stubs profitieren kann. Schauen wir uns nachfolgend an, wie man mit Mocks zustands- und verhaltensbasiert testen kann.

Ebenso wie Stubs dienen auch Mock-Objekte als Ersatz für Kollaborateure. Im Gegensatz zu Stubs, die Funktionalität bzw. Zustand bereitstellen, sind ***Mocks*** auf das ***Objektverhalten*** ausgerichtet und werden zum Testen von Interaktionen genutzt. ***Es ist demnach eine komplett andere Denkweise als bei den intuitiv besser erfassbaren Stubs gefordert***. Zum Einstieg beschäftigen wir uns zunächst mit den Grundlagen am Beispiel des Mocking-Frameworks Mockito[15].

Vorarbeiten im Build

Für die im Anschluss beschriebenen Funktionalitäten von Mockito müssen wir folgende Abhängigkeit (Stand September 2020) in unsere Build-Datei `build.gradle` einfügen:

```
testImplementation "org.mockito:mockito-core:3.5.10"
```

Um in den Beispielen nicht jedes Mal eine Menge an Imports zeigen zu müssen, gehen Sie bitte von folgenden statischen Imports bzw. einer Teilmenge davon aus:

```
import static org.mockito.Mockito.anyString;
import static org.mockito.Mockito.atLeast;
import static org.mockito.Mockito.atMost;
import static org.mockito.Mockito.doAnswer;
import static org.mockito.Mockito.doThrow;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.when;

import org.mockito.Mockito;

import static org.junit.Assert.assertEquals;
```

[15] https://code.google.com/p/mockito/

Mockito Basics

Mithilfe von Mockito können wir folgende Basisfunktionalitäten ausführen:

- `mock()` – Mock / Stub erstellen
- `when()` und `thenReturn()` – Verhalten beschreiben

Betrachten wir zum besseren Verständnis ein einfaches Beispiel: Eine Klasse `Greeting` besitzt eine Methode `greet()` und ist wie folgt implementiert:

```
public class Greeting
{
    public String greet()
    {
        return "Hello world!";
    }
}
```

Unser Ziel ist es nun, davon ein Test-Double zu erstellen, das bei Aufruf von `greet()` einen von der Originalimplementierung abweichenden Rückgabewert liefert:

```
public class MockitoHelloWorldExample
{
    @Test
    public void testGreetingReturnValue()
    {
        // Arrange
        final Greeting greeting = mock(Greeting.class);
        when(greeting.greet()).thenReturn("Changed by Mockito");

        // Act
        final String result = greeting.greet();

        // Assert
        assertEquals("Changed by Mockito", result);
    }
}
```

Listing 20.14 *Ausführbar als* '**MOCKITOHELLOWORLDEXAMPLE**'

Im Gegensatz zu den in früheren Auflagen dieses Buchs beschriebenen Mocking-Frameworks kann man mit Mockito gut dem ARRANGE-ACT-ASSERT-Stil folgen. Zunächst wird ein Test-Double erstellt und durch Aufruf von `when()` parametriert (Arrange). Danach führen wir die Aktion aus (Act) und können schließlich das Ergebnis prüfen (Assert). Wir nutzen hier zustandsbasiertes Testen. Demnach ist das durch `mock()` erstellte Test-Double kein Mock, sondern ein Stub. Mockito unterscheidet nicht zwischen diesen. Vielmehr besitzt man als Entwickler die Freiheit, selbst zu wählen, wie man die Test-Doubles einsetzen möchte. Das erlaubt es, zustands- sowie verhaltensbasiert und auch beides in Kombination zu testen.

Anpassungen für realitätsnähere Beispiele

Das vorherige Beispiel diente lediglich zur Einführung in die prinzipielle Arbeitsweise von Mockito. In der Praxis verwendet man Stubbing und Mocking jedoch dazu, um Kollaborateure durch Test-Doubles zu ersetzen. Zur besseren Verständlichkeit der nachfolgenden Ausführungen und um auf das Wesentliche bei Mockito fokussieren zu können, verwenden wir eine einfache Applikation, die die Klasse `Greeting` als Kollaborateur nutzt:

```
public class Application
{
    private final Greeting greeting;

    public Application(final Greeting greeting)
    {
        this.greeting = greeting;
    }

    public String generateMsg(final String name)
    {
        return greeting.greet(name);
    }
}
```

Zudem wandeln wir unsere `Greeting`-Klasse minimal ab und führen einen Parameter in `greet()` ein, um weitere Möglichkeiten mit Mockito kennenzulernen:

```
public class Greeting
{
    public String greet(final String name)
    {
        return "Hello " + name;
    }
}
```

Spezifische Rückgaben und Exceptions auslösen

Nachfolgend wollen wir uns anschauen, wie man für verschiedene Parameter spezifische Rückgabewerte bzw. gewünschtes Verhalten, insbesondere auch das Auslösen einer Exception, beschreiben kann. Dazu dienen folgende Methoden:

- `when()`, `anyString()` und `thenReturn()` – Verhalten für alle Eingabewerte beschreiben
- `when()`, Parameterwert und `thenReturn()` – Verhalten für spezifische Eingabewerte festlegen
- `when()` und `thenThrow()` – Exception auslösen

Nutzen wir diese Methoden in folgendem Beispiel. Für beliebige Eingaben[16] wird der Text »`Welcome to Mockito`« zurückgegeben, einzig für den Eingabetext `Mike` soll »`Mister Mike`« geliefert werden. Schließlich soll bei Eingabe von `ERROR` eine Exception ausgelöst werden. Für die beiden letzten Eingaben sehen wir, wie man die zuvor mit `anyString()` allgemein spezifizierte Rückgabe für einige Eingaben redefinieren kann:

```
@Test
public void testGreetingReturnValueAndThrowExcption()
{
    // Arrange
    final Greeting greeting = Mockito.mock(Greeting.class);

    // Achtung: Reihenfolge der when()-Aufrufe wichtig
    when(greeting.greet(anyString())).
                  thenReturn("Welcome to Mockito");
    when(greeting.greet("Mike")).
                  thenReturn(("Mister Mike");
    when(greeting.greet("ERROR")).
                  thenThrow((new IllegalArgumentException());

    // Act
    final Application app = new Application(greeting);
    final String result1 = app.generateMsg("Mike");
    assertEquals("Mister Mike", result1);

    final String result2 = app.generateMsg("ABC");
    assertEquals("Welcome to Mockito", result2);

    assertThrows(IllegalArgumentException.class,
                 () -> greeting.greet("ERROR"));   // => Exception
}
```

Listing 20.15 *Ausführbar als* **'MOCKITOTHROWINGEXAMPLE'**

Ein Aufruf von `generateMsg()` führt zu einem Aufruf der Methode `greet()` für die im Konstruktor der `Application`-Klasse übergebenen `greeting`-Instanz. Mithilfe von Mockito erstellen wir einen Stellvertreter und können uns so in die Aufrufkette für alle oder einige Eingaben einklinken und Modifikationen vornehmen.

Auch in diesem Beispiel kommt noch zustandsbasiertes Testen zum Einsatz. Insbesondere sehen wir, wie einfach sich Verhalten von Stubs bzw. deren simulierter Zustand vorgeben lässt.

Mehrere Rückgabewerte

Mitunter möchte man für sukzessive Aufrufe unterschiedliche Rückgabewerte bereitstellen. Mit Mockito lassen sich mit `thenReturn()` problemlos auch mehrere Rückgaben vorgeben – erfolgen Aufrufe häufiger als Werte in `thenReturn()` angegeben wurden, so wird immer wieder der letzte Wert geliefert.

[16] Diese lassen sich über `anyXyz()` ausdrücken. Das `Xyz` steht etwa für beliebige Typen, z. B. `anyInt()`, `anyLong()` usw.

In der folgenden Methode rufen wir dreimal `greet()` auf. Weil wir mit Mockito zwei Rückgabewerte vorgeben, sollte der dritte Aufruf das gleiche Ergebnis wie der zweite liefern:

```
@Test
void testMultipleReturnValues()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet(anyString())).
                  thenReturn("Hello Mockito1", "Hello Mockito2");

    // Act
    final Application app = new Application(greeting);
    final String result1 = app.generateMsg("One");
    final String result2 = app.generateMsg("Two");
    final String result3 = app.generateMsg("Three");

    // Assert
    assertEquals("Hello Mockito1", result1);
    assertEquals("Hello Mockito2", result2);
    assertEquals("Hello Mockito2", result3);
}
```

Listing 20.16 *Ausführbar als* '**MOCKITOMULTIPLERETURNVALUESEXAMPLE**'

Dies war fürs Erste der letzte zustandsbasierte Test. Nun wollen wir endlich verhaltensbasiert mit Mockito testen.

Prüfen von Aufrufen

Wir nutzen die gleichen Aufrufe von Funktionalitäten wie im vorherigen Beispiel. Anstatt aber zustandsbasiert deren Resultate zu prüfen, testen wir nun verhaltensbasiert, also, ob die erwartete Abfolge von Methodenaufrufen ausgeführt wird. Dazu dient in Mockito die Methode `verify()`, der man den zu prüfenden Mock übergibt. Danach spezifiziert man dann die erwartete Methode. Das Ganze sieht damit wie folgt aus:

```
@Test
public void testVerifyCallsAndParams()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet(anyString())).
                  thenReturn("Hello Mockito1", "Hello Mockito2");

    // Act
    final Application app = new Application(greeting);
    final String result1 = app.generateMsg("One");
    final String result2 = app.generateMsg("Two");
    final String result3 = app.generateMsg("Three");

    // Assert
    verify(greeting).greet("One");
    verify(greeting).greet("Two");
    verify(greeting).greet(anyString());
}
```

Listing 20.17 *Ausführbar als* '**MOCKITOVERIFYCALLSANDPARAMSEXAMPLE**'

Im Beispiel lernen wir zusätzlich zum Prüfen der Aufrufreihenfolge die Angabe von `anyString()` im Kontext von `verify()` kennen – zuvor hatten wir es nur bei der Vorgabe von Verhalten unserer Test-Doubles eingesetzt. Nachdem wir für zwei Werte den exakten Parameterwert spezifiziert haben, drücken wir im dritten Fall damit aus, dass wir zwar einen Aufruf der Methode `greet()` erwarten, uns der konkrete Parameterwert jedoch nicht interessiert.

Häufigkeit von Aufrufen

Bei der Beschreibung von Verhalten möchte man für Aufrufe teilweise die erwartete Anzahl an Aufrufen spezifizieren. Mit Mockito lässt sich mithilfe der Methoden `times()`, `atLeast()` und `atMost()` festlegen, wie häufig ein Aufruf erwartet wird, nämlich exakt, mindestens und höchstens die angegebene Anzahl. Diese Methoden werden im folgenden Programm genutzt, um ihre Funktionsweise zu verdeutlichen. Dazu rufen wir dreimal `greet()` auf und formulieren mithilfe von `verify()` verschiedene Erwartungen:

```
@Test
public void testGreetingCallCount()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet(anyString())).
                  thenReturn("Hello Mockito1", "Hello Mockito2");

    // Act
    final Application app = new Application(greeting);
    final String result1 = app.generateMsg("Tim");
    final String result2 = app.generateMsg("Mike");
    final String result3 = app.generateMsg("Tim");

    // Assert
    verify(greeting, atLeast(1)).greet("Tim");
    verify(greeting, atMost(2)).greet("Tim");
    verify(greeting, times(3)).greet(anyString());
}
```

Listing 20.18 *Ausführbar als* **'MOCKITOCALLTIMESEXAMPLE'**

Unerwartete Interaktionen

Im Normalfall prüft Mockito – anders als einige andere Mocking-Frameworks – nicht strikt, d. h., es sind abgesehen von den erwarteten auch weitere nicht spezifizierte Interaktionen (Methodenaufrufe) erlaubt. Das ist häufig eine praktische Einstellung, um die Tests nicht zu fragil und anfällig gegenüber Änderungen in den Applikationsklassen zu machen, sondern wirklich nur die wesentlichen Interaktionen zu prüfen. Möchte man aber beispielsweise verbieten, dass nach erwarteten Aufrufen noch weitere unerwartete erfolgen, so kann man dies über einen Aufruf von `verifyNoMoreInteractions()` ausdrücken.

Damit wir das nachvollziehen können, fügen wir eine Methode `noReturnJustPrint()` in unsere `Greeting`-Klasse ein:

```
public class Greeting
{
    // ...

    public void noReturnJustPrint(final String ignored)
    {
        System.out.println("some void method!");
    }
}
```

Zudem ergänzen wir die `Application`-Klasse um eine Methode `specialMsg()`, die die obige Methode zweimal aufruft:

```
public class Application
{
    // ...

    public void specialMsg(final String info)
    {
        greeting.noReturnJustPrint("preInfo");
        greeting.greet(info);
        greeting.noReturnJustPrint("postInfo");
    }
}
```

Im folgenden Beispiel wird mit Mockito weiterhin nur spezifiziert, dass `greet()` aufgerufen wird. Allerdings führt der Aufruf von `specialMsg()` im Test nun auch noch zu zwei zusätzlichen Methodenaufrufen, hier die zwei von `noReturnJustPrint()`. Dies wird von Mockito akzeptiert und verursacht keinen Testfehler.

```
@Test
public void testGreetingCallCount()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet(anyString())).thenReturn("The One and Only");

    // Act
    final Application app = new Application(greeting);
    app.specialMsg("HELLO");

    // Assert
    verify(greeting, times(1)).greet(anyString());
    // verifyNoMoreInteractions(greeting);
}
```

Listing 20.19 *Ausführbar als* '**MOCKITOINTERACTIONSEXAMPLE**'

Bei Bedarf kann man jedoch andere Methodenaufrufe als die über `verify()`-Aufrufe beschriebene Aufruffolge als Testfehler werten. Dazu ist ein Aufruf von `verifyNoMoreInteractions()` nötig, wie es als Kommentar angegebenen ist.

Durch diese Ergänzung schlägt die Testausführung wie folgt fehl:

```
org.mockito.exceptions.verification.NoInteractionsWanted:
No interactions wanted here:
```

Auf eine Sache möchte ich abschließend noch mal kurz hinweisen, wobei wir davon ausgehen, dass ein Aufruf von `doSomethingExpected()` erwartet wird:

```
mock.doSomethingUnexpected_Before();
mock.doSomethingExpected();
mock.doSomethingUnexpected_After();
```

Die beiden anderen Aufrufe verursachen einzeln oder in Kombination Testfehler. Leicht wird dies vergessen und man versucht mit `verifyNoMoreInteractions()` zu vermeiden, dass nach dem spezifizierten Ablauf weitere Aufrufe erfolgen. Wie erwähnt, werden aber nicht nur Aktionen nach dem spezifizierten Ablauf, sondern auch Interaktionen moniert, die davor stattfinden. Um der Sachverhalt »keine anderen Interaktionen auf den Mocks als die spezifizierten« klarer auszudrücken, hätte der Methodenname besser `verifyNoUnspecifiedInteractions()` lauten sollen.

Spezialfall: Mocking von `void`-Methoden

Wenn man `void`-Methoden ausführt, kann man deren Auswirkungen nur über die ausgelösten Seiteneffekte prüfen oder aber mit Mockito deren Aufruf testen. Aber es gibt doch ein paar Besonderheiten: Der bisherige Ansatz zum Festlegen von Verhalten und dem Muster

```
when(greeting.noReturnJustPrint("ABC"))
```

führt für `void`-Methoden zur Fehlermeldung »`The method when(T) in the type Mockito is not applicable for the arguments (void)`«.
Tatsächlich muss man deshalb für `void`-Methoden eine andere Syntax nutzen. Dabei hat man unter anderem folgende Optionen:

1. `doAnswer()` – Mithilfe eines `org.mockito.stubbing.Answer`-Objekts lässt sich eine Aktion innerhalb eines Test-Doubles ausführen.
2. `doThrow()` – Es ist auch möglich, eine Exception auszulösen, etwa wenn ein Übergabeparameter einen speziellen Wert besitzt.

Zu beiden Optionen schauen wir uns nun ein Beispiel an:

```
public class MockitoVoidMethodExample
{
    @Test
    void testHandlingVoidMethods()
    {
        // Arrange
        final Greeting greeting = Mockito.mock(Greeting.class);

        final Answer<String> answer = (invocation) ->
                                    { System.out.println("EXECUTING " +
                                      invocation.getMethod().getName() +
                                      Arrays.toString(invocation.getArguments()));
                                      return null;
                                    };

        doAnswer(answer).when(greeting).noReturnJustPrint(anyString());
```

```
        doThrow(IllegalStateException.class).when(greeting).noReturnJustPrint("
              ERROR");

        // Act
        greeting.additional("Mike");
        assertThrows(IllegalStateException.class,
              () -> greeting.additional("ERROR"));   // => Exception
    }
}
```

Listing 20.20 *Ausführbar als* **'MOCKITOVOIDMETHODEXAMPLE'**

Führt man den obigen Test als MOCKITOVOIDMETHODEXAMPLE aus, so produziert die Testmethode folgende Ausgaben und ein positives Testergebnis:

```
EXECUTING noReturnJustPrint[Mike]
```

Daran sieht man, dass man für `void`-Methoden mit `doAnswer()` eine Aktion ausführen und mit `doThrow()` eine Exception auslösen kann – hier für den Eingabetext `ERROR`.

Spezialfall: Auslesen von Parameterwerten

Nehmen wir an, wir wollten die Eingabeparameter sämtlicher Aufrufe einer speziellen Methode ermitteln, etwa um im Test nicht nur die Aufrufe, sondern auch die Übergabewerte prüfen zu können. Dazu dient ein `org.mockito.ArgumentCaptor`. Ein solcher sammelt alle Parameter aller Aufrufe und stellt diese dann über die Methode `getAllValues()` bereit.

Nachfolgendes Beispiel zeigt, wie man einen `ArgumentCaptor` einsetzen kann:

```
public class MockitoArgCaptorExample
{
    @Test
    public void testGreetingCallCount()
    {
        // Arrange
        final Greeting greeting = mock(Greeting.class);
        when(greeting.greet(anyString())).thenReturn("value");

        // Act
        final Application app = new Application(greeting);
        app.generateMsg("One");
        app.generateMsg("Two");
        app.generateMsg("Three");

        // Assert
        final ArgumentCaptor<String> argumentCaptor =
                                    ArgumentCaptor.forClass(String.class);
        verify(greeting, times(3)).greet(argumentCaptor.capture());

        final List<String> passedValues = argumentCaptor.getAllValues();
        assertEquals("One", passedValues.get(0));
        assertEquals("Two", passedValues.get(1));
        assertEquals("Three", passedValues.get(2));
    }
}
```

Listing 20.21 *Ausführbar als* **'MOCKITOARGCAPTOREXAMPLE'**

20.5.5 Abhängigkeiten mit Mockito auflösen

Nach dem Einstieg in die Mocking-Grundlagen wollen wir Mockito zum Testen und insbesondere zum Auflösen von Abhängigkeiten nutzen.

Weil Klassen oft im Kontext anderer Klassen ausgeführt werden, besteht beim Unit-Testen vielfach die Herausforderung, dass eine Klasse von einer oder mehreren anderen abhängt. Es wäre extrem aufwendig, möglicherweise sogar fast unmöglich und auch nicht wünschenswert, diese Kollaborateure oder gar ein ganzes Objektgeflecht geeignet zu parametrieren oder zu initialisieren. Vielmehr bietet sich der Einsatz eines Mocking-Frameworks an, um Tests ausführen zu können, ohne Auswirkungen oder Seiteneffekte auf echte Daten befürchten zu müssen. Klassische Beispiele, wo man dies nutzt, sind der Data Access Layer oder ein SMS-Service. In beiden Fällen möchte man die Unit Tests sicher ohne die externe Abhängigkeit ausführen können. Etwas Ähnliches haben wir mit dem `PizzaService` schon thematisiert und greifen es nun unter dem Gesichtspunkt des verhaltensbasierten Testens wieder auf.

Ausgangsbasis

Für das zustandsbasierte Testen wurde in Abschnitt 20.5.3 die Klasse `PizzaService`, die bei Bestellungen eine SMS-Benachrichtigung versendet, mithilfe der Technik EXTRACT AND OVERRIDE testbar gemacht. Dazu mussten wir aber sowohl in der Klasse selbst etwas ändern als auch einen Stub erzeugen.

Lernen wir nun, wie wir in diesem Fall die Abhängigkeiten mit Mockito und verhaltensbasiertem Testen einfacher auflösen können. Zunächst schauen wir uns noch mal die zu testende Klasse sowie die Klasse zum SMS-Versand an:

```
public class PizzaService
{
    private final SmsNotificationService notificationService;

    public PizzaService()
    {
        notificationService = new SmsNotificationService();
    }

    public void orderPizza(final String name)
    {
        notificationService.send("Pizza " + name + " wird in Kürze geliefert.");
    }
}

public class SmsNotificationService
{
    public void send(final String msg)
    {
        // SMS versenden ... hier durch einen Dialog repräsentiert
        JOptionPane.showConfirmDialog(null, msg);
    }
}
```

Wir haben zuvor folgende Schwachstellen ausgemacht:

1. Sicherlich soll nicht bei jedem Testlauf eine SMS versendet werden.
2. Das Darstellen eines Dialogs behindert die Automatisierung von Tests.

Die Wurzel allen Übels liegt aber in der direkten Abhängigkeit zu der zum SMS-Versand benötigten Klasse `SmsNotificationService`, die auch noch im Konstruktor erzeugt wird. Wir folgen nun den Ideen des Herauslösens von Abhängigkeiten, wie ich es exemplarisch in Abschnitt 20.5.2 gezeigt habe.

Umbau auf Constructor Injection

Gemäß den dort beschriebenen Aktionen führen wir einen Parameter im Konstruktor ein und injizieren die benötigte Instanz. Das erlaubt es uns, ein Test-Double im Test zu verwenden. Zudem haben wir als weitere Verbesserung die Hilfsmethode zum Erstellen des Nachrichtentextes bereits in die folgende Implementierung einfließen lassen:

```
public class PizzaService
{
    private final SmsNotificationService notificationService;

    public PizzaService(final SmsNotificationService notificationService)
    {
        this.notificationService = notificationService;
    }

    public void orderPizza(final String name)
    {
        notificationService.send(createNotificationMsg(name));
    }

    /* private */ static String createNotificationMsg(final String name)
    {
        return "Pizza " + name + " wird in Kürze geliefert.";
    }
}
```

Realisierung eines Unit Tests

Um Unit Tests auch ohne die Abhängigkeit zum SMS-Versand formulieren zu können, muss der Verweis auf die Klasse `SmsNotificationService` geeignet mit Mockito aufgelöst werden. Mit dem bislang gesammelten Wissen fällt uns die Implementierung eines Tests der Klasse `PizzaService` nicht besonders schwer:

```
@Test
public void testOrderPizza() throws Exception
{
    // Arrange
    final SmsNotificationService notificationService =
                                mock(SmsNotificationService.class);

    final PizzaService pizzaService = new PizzaService(notificationService);
```

```
    // Act
    pizzaService.orderPizza("Diavolo");
    pizzaService.orderPizza("Surprise");

    // Assert
    final ArgumentCaptor<String> argCaptor =
                                ArgumentCaptor.forClass(String.class);
    verify(notificationService, times(2)).send(argCaptor.capture());

    var expected = List.of(PizzaService.createNotificationMsg("Diavolo"),
                           PizzaService.createNotificationMsg("Surprise"));

    assertEquals(expected, argCaptor.getAllValues());
}
```

An diesem Beispiel sehen wir, wie leicht es fällt, Abhängigkeiten mithilfe von Mockito zu ersetzen. Das gilt jedoch nur dann so uneingeschränkt positiv, wenn es nur wenige Abhängigkeiten gibt und nur wenige Interaktionen getestet werden sollen. Je komplexer die Interaktion, desto schwieriger gestaltet sich die Formulierung der Testfälle. Hier spielt wieder Testbarkeit und gutes Design zusammen: Ein gelungenes Design zeichnet sich in der Regel dadurch aus, dass Klassen eine überschaubare Menge an Abhängigkeiten besitzen, wodurch wiederum die Testbarkeit erleichtert wird, da nur wenige Mocks und Stubs zum Einsatz kommen müssen.

Fazit

Hier wurde lediglich ein Einstieg in das Thema Testen mit Mocks und Stubs gegeben. Eine weiterführende Betrachtung von Mocks liefert das Buch »Practical Unit Testing with JUnit and Mockito« [44] von Tomasz Kaczanowski.

20.5.6 Unit Tests von privaten Methoden

Manchmal möchte man auch die privaten Bestandteile von Klassen testen, weil diese komplexere Berechnungen enthalten. Allerdings hat man darauf außerhalb der Klassen keinen Zugriff. Also, wie testet man private Methoden? Dazu existieren unter anderem folgende Möglichkeiten:

1. Manchmal ist das Klassendesign noch nicht ausgereift und möglicherweise bietet es sich an, ein paar der privaten Methoden in eine eigene Klasse auszulagern und dort mit höherer Sichtbarkeit bereitzustellen.
2. Am einfachsten ist es, die Sichtbarkeit einer zu testenden Methode auf Package-private zu ändern. Soll die Testbarkeit als Grund dafür explizit sichtbar sein, so kann man einen Kommentar `/* Test */` o. Ä. oder die Annotation `@VisibleForTesting` aus der Bibliothek Google Guava nutzen.
3. Mithilfe von Reflection (vgl. Abschnitt 12.1) kann man Zugriff auf private, zu testende Methoden erhalten. Allerdings muss dazu der Sichtbarkeitschutzmechanismus vor dem eigentlichen Methodenaufruf mit `setAccessible(true)`

außer Kraft gesetzt werden. Erschwerend kommt hinzu, dass im Kontext der Modularisierung seit Java 9 Zugriffe per Reflection nicht so einfach möglich sind. Für Details verweise ich auf mein Buch »Java – die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41].

Zunächst sollte man prüfen, ob gewisse Funktionalität an den aus Designsicht richtigen Stellen angeboten wird. Falls dies nicht der Fall sein sollte, kann man etwa Funktionalität verlagern und als Hilfsmethoden in Utility-Klassen bereitstellen. Verbleiben nach diesem Umbau noch private Methoden, die es sich zu testen lohnt, so ist die zweite Möglichkeit akzeptabel, da die Veränderung der Sichtbarkeit die Kapselung nur minimal beeinflusst, das Testen aber enorm erleichtert. ***Diese beiden Schritte beschreiben den von mir bevorzugten Weg***.

Die dritte Möglichkeit per Reflection hat dagegen diverse Nachteile: Zum einen müssen die Namen der aufzurufenden Methoden als Strings in den Sourcecode aufgenommen werden, was die Gefahr von Problemen durch Refactorings enorm erhöht: Werden Methoden umbenannt oder Signaturen verändert, so schlagen als Folge Tests fehl, die zuvor funktionierten, weil die gewünschten Methoden nicht mehr per Reflection gefunden werden. Für öffentliche Methoden, deren API sich eher selten und wenn dann eher geringfügig ändert, ist diese Einschränkung weniger problematisch – für private Methoden, an denen für gewöhnlich Änderungen (Parameterliste, Namensgebung usw.) doch eher häufig erfolgen, ist dies jedoch inakzeptabel. Außerdem ist der Zugriff per Reflection umständlicher, was den Testcode aufbläht und dessen Lesbarkeit erschwert.

Hinweis: Öffentliche vs. Package-private Methoden testen

Bezüglich der durch Unit Tests zu überprüfenden Methoden gibt es unterschiedliche Auffassungen. Manche Entwickler propagieren den Test der nach außen angebotenen Schnittstelle, also der öffentlichen Methoden. Andere bevorzugen dagegen den Test der kleineren Bausteine (`private`- und `protected`-Methoden). Beides hat seine Berechtigung: Werden ausschließlich öffentliche Methoden geprüft, so erfolgt eher ein Test auf Verhaltensebene der Business-Methoden, der zum Teil in Richtung Integrationstest geht. Interne Probleme oder Berechnungsfehler kann man besser durch Tests von Methoden mit geringerer Sichtbarkeit aufdecken. Finden jedoch nur solche Tests statt, so wird das Zusammenspiel der Methoden vernachlässigt. Demnach sollte man möglichst beide Arten von Tests durchführen. ***Unabhängig von der Sichtbarkeit empfiehlt es sich, kompliziertere Programmelemente zu überprüfen, da diese eher Fehler enthalten als verständliche und strukturell einfachere Programmteile***. Oft findet sich Komplexität nicht in öffentlichen Methoden, sondern vermehrt in denen der Sichtbarkeit `protected` und eingeschränkter. Allerdings beschreiben öffentliche Methoden das Verhalten der Klasse, das sicher auch getestet werden sollte. Damit sind wir wieder bei der obigen Aussage: ***Teste die wichtigen und / oder komplexeren Dinge***.

Abschließend noch ein Hinweis: Falls sich Fehler in privaten Methoden leicht am Verhalten von öffentlichen Methoden ablesen lassen, so bietet es sich an, die privaten Methoden bevorzugt indirekt durch Tests der öffentlichen Methoden zu prüfen. Ist das nicht gegeben, sollten auch Package-private Bestandteile getestet werden.

20.6 Test Smells

Gute und verständliche Tests zu schreiben ist gar nicht so einfach, wie es vielleicht scheinen mag. Nachfolgend lernen wir deshalb einige Fallstricke kennen, über die man bei ersten Gehversuchen mit Unit Tests stolpern kann. Wenn Sie diese Fallstricke vermeiden, werden Ihre Tests viel wertvoller zur Beurteilung der Softwarequalität.

Ich möchte Ihnen einige Tipps und Tricks zum Unit-Testen mit auf den Weg geben und dabei mit einigen Aussagen von James Coplien starten, die seinem Artikel mit dem provokanten Titel »Why Most Unit Testing is Waste«[17] [13] entstammen:

- »Design a test with more care than you design the code.«
- »Tests don't improve quality: developers do.«

Dr. Venkat Subramaniam, Autor verschiedener lesenswerter Bücher sowohl zu Groovy als auch zu Java, schreibt in seinem Buch »Programming Groovy 2: Dynamic Productivity for the Java Developer« [71] über Unit Tests, dass diese dem Motto FAIR folgen sollten, was so viel bedeutet wie FAST AUTOMATIC ISOLATED REPEATABLE. Diesem Motto würde ich gern noch die Anforderung Zuverlässigkeit hinzufügen.

20.6.1 Test Smell: Unangebrachtes `assertTrue()` und `assertFalse()`

Zur Absicherung von erwarteten Bedingungen dient der Assert-Teil eines Testfalls. Betrachten wir ein paar scheinbar ganz einfache Prüfungen von Erwartungen:

```
@Test
void testSmellAssertTrueAndFalse()
{
    // ARRANGE
    final DatabaseConnector db = createDbConnector(10);

    // ACT
    db.process();
    final Tasks tasks = db.getTasks();

    // ASSERT
    assertTrue(db.writeCount == 10);
    assertTrue(tasks.totalProcessed == 10);
    assertTrue(tasks.errorCount == 0);
}
```

[17] http://www.rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf

Was ist an diesen Zustandsprüfungen problematisch? Eine ganze Menge: Hier werden boolesche Bedingungen mithilfe der Methoden `assertTrue()` geprüft. Allerdings erhält man damit lediglich eine Aussage, ob die Bedingung erfüllt ist oder nicht, jedoch keine Aussage darüber, was der erwartete und der beobachtete Wert ist. Wenn eine der Prüfungen fehlschlägt, weiß man somit nicht direkt, welche das war. Das wird noch dadurch verschärft, dass recht viele Prüfungen erfolgen.

Wie macht man es besser? Verwendet man `assertEquals()` statt `assertTrue()` oder `assertFalse()`, so erhält man beim Fehlschlagen zumindest Hinweise auf Abweichungen vom erwarteten Wert. Den Testfall korrigieren wir wie folgt:

```
@Test
void testSmellAssertTrueAndFalse_Solution()
{
    final DatabaseConnector db = createDbConnector(10);

    db.process();
    final Tasks tasks = db.getTasks();

    assertEquals(10, db.writeCount);
    assertEquals(10, tasks.totalProcessed);
    assertEquals(0, tasks.errorCount);
}
```

Nach diesen Umformungen in eine Prüfung mit `assertEquals()` erzeugt das JUnit-Framework beim Fehlschlagen einer Bedingung automatisch einen Hinweistext, der beschreibt, welcher Wert erwartet und welcher geliefert wurde, in etwa so:

```
java.lang.AssertionError: expected:<10> but was:<12>
```

Das ist schon eine Verbesserung – jedoch wird durch die drei Aufrufe die Zuordnung zu dem tatsächlich fehlschlagenden `assertEquals()` erschwert. Das liegt schlicht daran, dass hier im Beispiel noch die zwei Test Smells »Zu viele Asserts im Testfall« und »Asserts ohne Hinweis« vorliegen. Mögliche Abhilfen lernen wir nun bei der Besprechung der nächsten Test Smells kennen.

20.6.2 Test Smell: Zu viele Asserts im Testfall

Mit einem Testfall soll ein möglichst eng umrissenes Verhalten überprüft werden. Im besten Fall lässt sich dies mit nur einem Assert prüfen. Falls ab und zu ein paar mehr Asserts zur Zustandsprüfung benötigt werden, ist dies akzeptabel, sofern diese semantisch zusammengehörend (kohärent) sind, also nicht mehrere Aspekte prüfen.

In folgendem Beispiel findet man sechs Asserts, die zumindest zwei Sachverhalte prüfen, hier durch die Objekte `db` und `tasks` repräsentiert:

```
@Test
void testSmellToManyAsserts()
{
    final DatabaseConnector db = createDbConnector(10);

    db.process();
    final Tasks tasks = db.getTasks();

    assertEquals(10, db.readCount);
    assertEquals(10, db.writeCount);
    assertEquals(10, db.commitCount);

    assertEquals(10, tasks.totalProcessed);
    assertEquals(0, tasks.errorCount);
    assertEquals(SUCCESS, tasks.status);
}
```

Wenn ein Testfall mehrere Asserts enthält, kann es mehrere Gründe geben, warum der Test nicht bestanden wird. Damit fällt es schwer, einen Verstoß direkt einem Assert zuzuordnen und somit ist auch die Ursache schwieriger zu ermitteln.

Wie macht man es besser? Für das Beispiel bietet es sich an, die Prüfungen in zwei Testfälle aufzuteilen, einen, der sich um das `db`-Objekt kümmert, und einen anderen, der das `task`-Objekt prüft:

```
@Test
void testSmellToManyAsserts_Solution_Db_related()
{
    final DatabaseConnector db = createDbConnector(10);

    db.process();

    assertEquals(10, db.readCount);
    assertEquals(10, db.writeCount);
    assertEquals(10, db.commitCount);
}

@Test
void testSmellToManyAsserts_Solution_Task_related()
{
    final DatabaseConnector db = createDbConnector(10);

    db.process();
    final Tasks tasks = db.getTasks();

    assertAll(() -> assertEquals(10, tasks.totalProcessed),
              () -> assertEquals(0, tasks.errorCount),
              () -> assertEquals(SUCCESS, tasks.status));
}
```

Man sieht sehr schön, wie die Testfälle auf das zu prüfende Objekt ausgerichtet sind. Auch in der Praxis kann man beim Entwirren eines komplexeren Testfalls oftmals mehrere Testfälle extrahieren, die für sich dann deutlich verständlicher und kürzer sind.

Zwar haben wir schon die Menge der Asserts auf ein akzeptables Maß pro Testfall reduzieren können. Allerdings gibt es ein paar Dinge, die man bei mehreren Asserts in einem Testfall bedenken sollte: Zum einen bietet sich die Verwendung von

`assertAll()` an, wie es im zweiten Testfall angedeutet wurde. Dadurch werden immer alle Asserts abgearbeitet und man hangelt sich bei Fehlern nicht einzeln durch diese. Zum anderen empfiehlt sich in der gezeigten Kombination mit mehreren Prüfungen die Verwendung von Hinweistexten in der Assertion. Ohne Angabe ist es mitunter recht schwer, die Fehlerursache einzugrenzen, zumindest wenn sehr ähnliche Prüfungen und erwartete Werte genutzt werden. Schauen wir uns nun also die Möglichkeit an, Hinweistexte bereitzustellen.

20.6.3 Test Smell: Asserts ohne Hinweis

Wenn es in einem Testfall mehr als ein Assert gibt, so ist es oftmals hilfreich, für die einzelnen Asserts einen Hinweistext bereitzustellen. Das gilt insbesondere für Prüfungen mit `assertTrue()` bzw. `assertFalse()`.

Betrachten wir folgendes Assert mit dem Vergleich zweier Laufzeiten:

```
@Test
void testSmellNoHint()
{
    final long runtime1 = measure(() -> complexCalculation1());
    final long runtime2 = measure(() -> complexCalculation2());

    assertTrue(runtime2 > runtime1);
}
```

Diesen Testfall habe ich sehr ähnlich in der Praxis gefunden und er schlug des Öfteren fehl. Diese krude Implementierung des Tests erschwert das Nachvollziehen.

Wie macht man es besser? Es bietet sich folgende Korrektur durch Angabe eines Hinweistextes an:

```
@Test
void testSmellNoHint_Solution()
{
    final long runtime1 = measure(() -> complexCalculation1());
    final long runtime2 = measure(() -> complexCalculation2());

    assertTrue(runtime2 > runtime1,
               "runtime2 " + runtime2 + " ms should be > " + runtime1 + " ms");
}
```

Damit lässt sich das Problem durch die erzeugte Fehlermeldung

```
java.lang.AssertionError: runtime2 550 ms should be > 770 ms
```

eingrenzen und kann so überhaupt erst adressiert werden.

Neben Asserts ohne Hinweis ist es oftmals keine gute Idee, fixe Laufzeitannahmen in Unit Tests zu machen. Das wird als separat als »Test Smell: Unit Tests zur Prüfung von Laufzeiten« kurz behandelt.

20.6.4 Test Smell: Einsatz von `toString()` in `assertEquals()`

Eine Problematik beim Testen besteht in der sicheren Wiederholbarkeit. Manchmal ist diese nicht gegeben, weil zu feingranular und damit fragil geprüft wird, etwa wenn eine Vielzahl einzelner Werte in Kombination miteinander verglichen werden. So etwas sieht man beispielsweise im Zusammenspiel mit der Methode `toString()`. Lasse Koskela spricht in seinem Buch »Effective Unit Testing: A Guide for Java Developers« [47] dann von ***Hyperassertions***.

Nachfolgend sehen wir eine Prüfung einer Datenbankkonfiguration. Dieser Test basiert auf einer Stringrepräsentation, hier stark gekürzt:

```
@Test
void testSmellToStringInAssertEquals()
{
    final DatabaseConnector dbConnection = createMongoDbConnector();

    assertEquals("mongodb.writeConcern.timeout=10000," +
                 "mongodb.writeConcern.writes=1, " +
                 "mongodb.port=27017, " +
                 "mongodb.password=ksdjsa2455aAYdsj," +
                 "mongodb.user=ABCD",
                 dbConnection.toString());
}
```

Wie schon in Kapitel 16 über Bad Smells erwähnt, sind Vergleiche, basierend auf den von `toString()` produzierten Ausgaben, fragil und sollten vermieden werden. Beachtet man diesen Hinweis nicht, so würde jede noch so kleine Änderung entweder an der Konfiguration oder der Aufbereitung der Stringrepräsentation den Vergleich fehlschlagen lassen. Im Beispiel würde schon eine minimale Rekonfiguration in den Datenbankverbindungsdaten (z. B. der Portnummer oder eines Timeouts) Probleme bereiten. Eine solche Modifikation sollte (abgesehen von raren Ausnahmen) weder zu einem Fehlschlag des Testfalls führen noch eine Anpassung im Testfall notwendig machen.

Wie macht man es besser? Wenn man nur einzelne relevante Bestandteile der als String vorliegenden Datenbankkonfiguration mithilfe von `contains()` abfragen würde, so wäre die Prüfung weniger fragil – dieses Vorgehen entspricht dann allerdings eher einem Trostpflaster. Deutlich sinnvoller ist es, die wichtigen Informationen mithilfe von Zugriffsmethoden zu ermitteln und geeignet durch Asserts zu überprüfen.

Als Grundregel gilt: ***Prüfe in Unit Tests lediglich diejenigen Werte, die wirklich relevant sind und benötigt werden, um die Korrektheit zu garantieren.*** Je mehr Zustand drumherum geprüft wird, der nicht (oder kaum) mit dem eigentlichen Test verbunden ist, desto größer ist die Fragilität des Tests. Damit beginnt ein Wartungsalbtraum für Tests: Diese müssen dann ständig angepasst werden, weil sie sonst bei vielen eigentlich eher kosmetischen oder unbedeutenderen Programmänderungen fehlschlagen.

Also, wie könnte man es konkret besser machen: Sicherlich sollte das Passwort nicht Bestandteil des Tests sein. Wir nehmen zudem noch an, das die Timeout-Zeit

irrelevant sei. Die restlichen drei Konfigurationswerte würde man dann wieder mit `assertAll()` geeignet mithilfe einzelner Abfragen prüfen.

```
@Test
void testSmellToStringInAssertEquals_Solution()
{
    final DatabaseConnector dbConnection = createMongoDbConnector();

    assertAll(() -> assertEquals(1, dbConnection.mongodb.writeConcern_writes),
              () -> assertEquals(27017, dbConnection.mongodb.port),
              () -> assertEquals("ABCD", dbConnection.mongodb.user));
}
```

Dadurch erhält man bei Fehlern konkrete Hinweise, welche Werte unerwartet sind und zudem würde eine (leicht) anders aufbereitete Stringrepräsentation keine Auswirkungen auf das Testergebnis haben, da nun die Attribute (hier vereinfachend ohne Datenkapselung und stattdessen mit direkten Zugriff) auf Übereinstimmung geprüft werden. Selbst eine JSON-Aufbereitung in `toString()` wäre nun kein Problem mehr.

20.6.5 Test Smell: Unit Tests zur Prüfung von Laufzeiten

Manchmal sollen die Laufzeiten von Programmteilen gegen ein erwartetes Laufzeitverhalten geprüft werden. Leider erfolgt dies zum Teil durch den Einsatz von Unit Tests. Allerdings stellen diese eigentlich kein probates Mittel dar, insbesondere dann nicht, wenn man keine Varianzen in den Laufzeiten berücksichtigt. Schauen wir uns ein Beispiel eines problematischen Sourcecodes an:

```
@Test
void testSmellTestingRunningTimes()
{
    final int count = 1000;

    // Laufzeitmessung für Algorithmus 1
    long runtime1 = 0;

    for (int i = 0; i < count; i++)
    {
        final long start = System.currentTimeMillis();
        performCalculation_V1();
        final long end = System.currentTimeMillis() - start;
        runtime1 += end;
    }

    // Laufzeitmessung für Algorithmus 2
    // for-loop mit performCalculation_V1() hier exemplarisch 100 ms
    long runtime2 = 100;

    // Fragile Annahme über die Laufzeiten
    assertTrue(runtime2 > runtime1,
               "calc V2 runtime should be > calc V1 runtime");
}
```

In diesem (bis auf Kleinigkeiten) so in der Praxis gefundenen Sourcecode wurden gleich mehrere Fehler gemacht. Eine solche Häufung von Fehlern ist nicht ungewöhnlich. Man findet diese oftmals in den aus anderen Gründen schon besonders fragilen Teilen (in die-

sem Fall im Sourcecode des Tests). Das Ganze bestärkt die Aussage von James Coplien, dass einige Tests, zumindest in der Art, wie sie erstellt wurden, überflüssig sind bzw. Tests mit ebensolcher Sorgfalt wie der eigentliche Sourcecode erstellt werden müssen. Dadurch vermeidet man potenziell aufwendige Überarbeitungen der Testfälle. Noch unerfahrene Entwickler sollten sich einen erfahrenen Kollegen suchen, der sie beim Erstellen der Testfälle unterstützt oder zumindest mit ihnen ein Codereview durchführt.

Wie macht man es besser? Führen wir ein Codereview einmal virtuell durch: Bei der ersten Variante der Laufzeitberechnung ist zunächst einmal ungünstig, dass die Zeitmessung mithilfe von `currentTimeMillis()` erfolgt. Wieso? Die damit gemessenen Werte sind recht ungenau. Zudem ist es ungeschickt, die einzelnen Laufzeiten aufzuaddieren, statt eine Gesamtlaufzeit zu berechnen. So verstärken sich Fehler bzw. die Abweichungen. Dadurch wird meistens eine deutlich geringere oder größere Laufzeit als die tatsächliche gemessen, sodass die Gesamtlaufzeit kürzer oder länger scheint, als sie ist. Auch der Variablenname `end` ist irreführend, weil hier die Laufzeit einer Einzelberechnung ermittelt wird und nicht der Endzeitpunkt. Wie man es besser machen kann, ist im Listing für den zweiten Messlauf gezeigt:

```
@Test
void testSmellTestingRunningTimes_Improved()
{
    final int count = 1000;

    // Verbesserte Laufzeitmessung für Algorithmus 1
    // Verbesserung 1: nanoTime()
    final long start = System.nanoTime();
    for (int i = 0; i < count; i++)
    {
        performCalculation_V1();
    }
    // Verbesserung 2: Gesamtlaufzeit
    final long runtime1 = System.nanoTime() - start;

    final long start2 = System.nanoTime();
    // for-loop
    final long runtime2 = System.nanoTime() - start2;

    // Immer noch fragile Annahme über die Laufzeiten
    assertTrue(runtime2 > runtime1,
               "calc V2 runtime should be > calc V1 runtime");
}
```

Beachten Sie bitte, dass die Prüfung der Laufzeit hochgradig von äußeren Einflüssen abhängig und somit sicher nicht zuverlässig wiederholbar ist. Zwar erhöhen die jeweils 1000 Durchläufe vermutlich die Messgenauigkeit, allerdings verstößt dieser Test dadurch aber ziemlich sicher auch gegen die Forderung, schnell ausführbar zu sein. Insgesamt sollte man vorsichtig sein, wenn man in Unit Tests mit Annahmen zum Timing arbeitet, da sich Effekte von der momentanen Systemlast möglicherweise stark auf den Testablauf auswirken können und zu fragilen Tests führen: Von Lauf zu Lauf oder bei variierender Systemlast bekommt man unterschiedliche Testresultate. Das ist zu vermeiden, weil die Tests bedeutend an Aussagekraft verlieren. Das verringert wie-

derum die Motivation für das Erstellen von Unit Tests, worunter oftmals längerfristig die Qualität leidet.

Zudem sollte man bedenken, dass sich Test- und Produktivumgebungen oft in der Hardwareausstattung unterscheiden und eine (deutlich) potentere Produktivumgebung möglicherweise zu abweichenden Testresultaten führen würde.

20.7 Nützliche Tools für Unit Tests

In diesem Abschnitt werden mit Hamcrest, AssertJ, MoreUnit, Infinitest und JaCoCo sowie EclEmma nützliche Tools zum Unit-Testen vorgestellt. Hamcrest und AssertJ helfen beim Formulieren von Testbedingungen. MoreUnit erleichtert das Erstellen und Ausführen von Unit Tests. Infinitest sorgt für die Durchführung von Unit Tests als Regressionstests. Die Tools JaCoCo und EclEmma ermitteln die sogenannte Testabdeckung. Dadurch kann man vor allem die Programmteile identifizieren, die nicht (oder zumindest nicht ausreichend) von Unit Tests geprüft werden.

20.7.1 Hamcrest

Hamcrest ist ein Tool, dessen Einsatz die Lesbarkeit und die Verständlichkeit von Unit Tests erhöhen kann. Dazu erlaubt es Hamcrest, Vergleichsoperationen, sogenannte ***Matcher***, zu schreiben. Diese lassen sich oftmals gewinnbringend kombinieren. Außerdem kann man einige Bedingungen nahezu umgangssprachlich deklarativ formulieren. Für Unit Tests ist dies nützlich, um die Lesbarkeit zu erhöhen. Zudem fügt sich Hamcrest nahtlos in verschiedene Test-Frameworks ein, etwa JUnit oder TestNG. Die ohne Hamcrest erstellten Tests bleiben weiterhin gültig, wodurch ein schrittweiser Wechsel erleichtert wird und man die neue Schreibweise sukzessive einführen kann.

Um Hamcrest nutzen zu können, müssen wir folgende Abhängigkeit in unsere Gradle-Build-Datei `build.gradle` hinzufügen:

```
compile group: 'org.hamcrest', name: 'hamcrest-core', version: '2.2'
compile group: 'org.hamcrest', name: 'hamcrest-library', version: '2.2'
```

Hamcrest-Integration in JUnit

Der Einsatz von Hamcrest kann die Lesbarkeit von Assertions erhöhen. Dabei dient die Methode `assertThat()` zum Aufstellen einer Testbehauptung sowie verschiedene Methoden zum Formulieren einzelner Bedingungen mithilfe sogenannter Matcher. Zum Start sind diese Imports sinnvoll:

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;
import static org.hamcrest.Matchers.is;
```

Schauen wir uns an, wie sich folgende mit JUnit recht technisch formulierte Testbehauptung vereinfachen lässt:

```
assertEquals(1234, calculatedPrice());
```

Nutzen wir dagegen Hamcrest ist das Ganze (für meinen Geschmack) besser lesbar:

```
assertThat(calculatedPrice(), is(equalTo(1234)));
```

Tatsächlich ist die zweite Variante zwar nahezu umgangssprachlich lesbar, es gibt jedoch auch Entwickler, die die obere Form intuitiv besser und schneller erfassen. Damit ergibt sich folgender Tipp: Implementierungsnahe Tests können eher Variante 1 nutzen und Variante 2 bietet sich für High-Level-Tests an. Erinnern Sie sich: JUnit erlaubt es auch, Integrations- oder kleinere Systemtests zu erstellen.

Wie schon beschrieben, erleichtert die Angabe eines Hinweistextes die Identifikation der Fehlerursache, speziell wenn mehrere Asserts oder die Varianten `assertTrue()` bzw. `assertFalse()` genutzt werden, da letztere keine verständliche Meldung erzeugen. Zwar wird die produzierte Fehlermeldung durch den Hinweistext informativer, dafür reduziert sich die Lesbarkeit des Sourcecodes aber ein wenig:

```
assertEquals(1234, calculatedPrice(), "price");
assertThat("price", calculatedPrice(), is(1234));
```

Dabei wird eine kleine Unschönheit deutlich: Während in JUnit 4 der Hinweistext noch als erster Parameter anzugeben war, ist dies nun der letzte. Bei Hamcrest ist es aber immer noch der erste Parameter.

Hamcrest in Aktion: Der erste Test

Um Hamcrest kennenzulernen, schreiben wir einen kleinen Unit Test. Statt der bekannten `assertEquals()`-Methode zum Aufstellen einer Testbehauptung nutzen wir hierbei die Methode `assertThat()`.

In folgendem Beispiel soll geprüft werden, ob der Parameter `tim` inhaltlich mit dem Wert von `tim2` übereinstimmt. Zum Aufstellen der Testbehauptung kommt der Matcher `equalTo(T)` zum Einsatz. Aus Gründen der besseren Lesbarkeit werden die Hamcrest-Matcher und Hilfsmethoden statisch importiert und damit der Vergleich definiert: Mit `equalTo(T)` erfolgt dieser über die `equals(Object)`-Methode des im ersten Parameter angegebenen Objekts. In den zwei weiteren Zeilen werden statt komplexerer Business-Objekte nur einzelne Attribute davon zum Aufstellen einer Behauptung genutzt.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;

public class SimplePersonTest
{
   @Test
   public void testEquals()
   {
      final Person tim = new Person("Tim", LocalDate.of(1971, 3, 27), "Kiel");
      final Person tim2 = new Person("Tim", LocalDate.of(1971, 3, 27), "Kiel");

      assertThat(tim, equalTo(tim2));
      assertThat("Geburtsjahr", tim.getBirthday().getYear(), equalTo(1971));
      assertThat("Stadt", tim.getCity(), equalTo("Kiel"));
   }
}
```

Listing 20.22 *Ausführbar als* '**SIMPLEPERSONTEST**'

Der Test wird erfolgreich durchlaufen, sofern die Klasse `Person` die `equals(Object)`-Methode korrekt implementiert.

Lesbarkeit weiter erhöhen Wir haben schon gesehen, dass der `is`-Matcher zwar zur Lesbarkeit beiträgt, allerdings besitzt er keine eigene Funktionalität, sondern ruft nur den `equalTo`-Matcher auf. Folgende Zusicherungen drücken alle das Gleiche aus, wodurch man je nach persönlichem Geschmack die für einen selbst am besten lesbare Variante wählen kann.

```
assertThat(tim, equalTo(tim2));
assertThat(tim, is(equalTo(tim2)));
assertThat(tim, is(tim2));
```

Übersicht über die wichtigsten Matcher

Hamcrest bietet diverse Matcher zur Beschreibung von Bedingungen, wodurch sich Unit Tests besser lesbar gestalten lassen. Dazu sollten die Matcher mit statischen Imports aus dem Package `org.hamcrest` eingebunden werden.

Object Matcher Referenz- und inhaltliche Gleichheit prüft man mit den Matchern `sameInstance` bzw. `equalTo`. Nützlich ist auch der typprüfende Matcher `instanceOf`. Zum Prüfen auf `null` dienen die Matcher `notNullValue` und `nullValue`.

Logische Matcher Die Matcher `allOf`, `anyOf` und `not` realisieren logische Operatoren entsprechend '`&&`', '`||`' bzw. '`!`' in Java.

String Matcher Für den Vergleich von Strings existieren diverse Matcher mit sprechenden Namen. `equalToIgnoringCase` testet die case-insensitive Gleichheit von

Strings. Mit `equalToIgnoringWhiteSpace` testet man die Gleichheit, ohne auf Leerzeichen zu achten. Die Matcher `containsString`, `startsWith` und `endsWith` prüfen auf Übereinstimmung mit Teilstrings.

Number Matcher Beim Vergleich von Gleitkommazahlen sollte man prüfen, ob diese innerhalb eines Toleranzbereichs liegen (vgl. Abschnitt 5.1.2). Der Matcher `closeTo` berücksichtigt dies. Die Matcher `greaterThan`, `greaterThanOrEqualTo`, `lessThan`, `lessThanOrEqualTo` prüfen entsprechend ihrem Namen auf größer, größer gleich, kleiner und kleiner gleich.

Collections Matcher Auch im Bereich von Collections existieren eine Menge an hilfreichen Matchern, etwa folgende: `hasItem`, `hasItems`, `hasEntry`, `hasKey` und `hasValue`. Alle prüfen entsprechend ihrem Namen, ob eine Collection ein Element oder eine Menge von Elementen enthält bzw. für Maps, ob ein Eintrag, ein Schlüssel oder ein Wert vorhanden sind.

Lesbare Suchen in Collections Zur Suche nach einen Element in einer Collection würde man sich mit JUnit eine eigene Hilfsmethode, hier als Beispiel `hasItem()`, schreiben müssen:

```
boolean found = hasItem(customers, desiredPerson);
assertTrue(found);

boolean hasItem(final Collection<Person> customers, final Person desiredPerson)
{
    return customers.stream().anyMatch(person -> person.equals(desiredPerson));
}
```

In Hamcrest lässt sich dies deutlich besser lesbar mit `hasItem()` schreiben:

```
assertThat(customers, hasItem(desiredPerson));
```

20.7.2 AssertJ

In den letzten Monaten habe ich mich als echte Alternative zu Hamcrest immer stärker für AssertJ begeistert, das das Schreiben verständlicher und auf einem Fluent API basierender, gut lesbarer Assertions unterstützt, und möchte es hier kurz vorstellen.

AssertJ-Integration

Um AssertJ nutzen zu können, müssen wir folgende Abhängigkeit in unsere Gradle-Build-Datei `build.gradle` hinzufügen:

```
// Assert J
testCompile group: 'org.assertj', name: 'assertj-core', version: '3.17.2'
```

Die nachfolgend gezeigten Testmethoden benötigen folgenden Import:

```
import static org.assertj.core.api.Assertions.*;
```

Einführung in AssertJ

Genau wie Hamcrest nutzt auch AssertJ `assertThat()` zum Aufstellen von Testbehauptungen. Abhängig vom als Parameter übergebenen Objekt erhalten wir passende typspezifische `XyzAssert`-Klassen als Rückgabe. Diese bieten dann eine Vielzahl an praktischen Prüfmethoden. Wir schauen uns ein paar Testmethoden an, die exemplarisch einige AssertJ-Funktionalitäten und insbesondere die gute Lesbarkeit und intuitive Verständlichkeit zeigen.

Stringprüfungen Während JUnit 5 lediglich die Prüfung auf (Nicht-)Übereinstimmung bietet, ist es gerade bei Strings in der Praxis wünschenswert, lesbar prüfen zu können, ob ein String nicht leer ist oder gewünschte Zeichen enthält. Bei JUnit 5 müsste man dazu auf eine unschöne Prüfung mit `assertFalse()` und `assertTrue()` zurückfallen:

```
@Test
void someStringAsserts()
{
    // JUnit style
    assertFalse("ABC".isEmpty());
    assertTrue("ONE TWO THREE".contains("TWO"));

    // AssertJ
    assertThat("ABC").isNotEmpty();
    assertThat("ONE TWO THREE").contains("TWO");
}
```

Zahlenprüfungen Auch beim Prüfen von Zahlen bietet AssertJ einiges an Komfort. Folgendes Listing zeigt zunächst den Test auf exakte Übereinstimmung, danach mit einer Abweichung und schließlich die Prüfung auf einen Wertebereich:

```
@Test
void someNumberAsserts()
{
    // JUnit style
    assertEquals(42, 42);
    assertEquals(3.415, 3, 0.15d);

    IntPredicate greaterThan =  n -> n >= 27;
    assertTrue(greaterThan.test(42));
    assertTrue(isBetween(7, 0, 10));

    // AssertJ
    assertThat(42).isEqualTo(42);
    assertThat(3.1415).isEqualTo(3, withPrecision(0.15d));
    assertThat(3.1415).isCloseTo(3, withPrecision(0.15d));
    assertThat(42).isGreaterThanOrEqualTo(27);
    assertThat(7).isBetween(0, 10);
}

boolean isBetween(int n, int lowerBound, int upperBound)
{
    return n >= lowerBound && n < upperBound;
}
```

Bewusst habe ich hier noch einige Alternativen mit JUnit 5 angegeben, um die Vorteile von AssertJ ganz deutlich zu machen. Für die folgenden Prüfungen zeige ich dies nicht mehr, da das Prinzip jeweils analog zu den bisher gezeigten Aufrufen ist: Man muss sich mit JUnit 5 Hilfsmethoden oder Prädikate erstellen und dann geeignet aufrufen.

Listenprüfungen Praktischerweise setzt sich der Komfort für Listen und Collections nahtlos fort. Besser noch, hier gibt es eine Vielzahl an praxisrelevanten Prüfmethoden, etwa `startsWith()` oder `endsWith()`, die man sonst nur aus der Klasse `String` kennt. Das Schreiben von Tests wird auf einmal zur Freude und es entstehen verständliche, gut lesbare Test. Man möchte gar nicht überlegen, wie kompliziert dies rein mit JUnit-5-Bordmitteln wäre:

```
@Test
void someListAsserts()
{
    List<String> list = List.of("2", "3", "5", "7", "11", "13");

    // JUnit style
    assertNotNull(list);
    assertFalse(list.isEmpty());

    // AssertJ
    assertThat(list).isNotNull();
    assertThat(list).isNotEmpty();
    assertThat(list).startsWith("2");
    assertThat(list).contains("7");
    assertThat(list).endsWith("11", "13");
}
```

Aber es geht noch besser! Durch das Fluent API können wir die Aufrufe der Prüfmethoden verketten:

```
@Test
void someChainedListAsserts()
{
    List<String> list = List.of("2", "3", "5", "7", "11", "13");

    assertThat(list).isNotNull().isNotEmpty().startsWith("2").contains("7");
}
```

Zudem gibt es ausgefeiltere Prüfungen, etwa um auf Nichtexistenz eines Elements oder aber auf das Auftreten einer Abfolge von Werten zu prüfen:

```
@Test
void someMoreListAsserts()
{
    List<String> list = List.of("2", "3", "5", "7", "11", "13");

    assertThat(list).doesNotContain("42").containsSequence("3", "5", "7");
}
```

Map-Prüfungen Für Maps setzt sich das zuvor vorgestellte Konzept nahtlos fort:

```
@Test
void someMapAsserts()
{
    Map<String, Integer> nameAgeMap = Map.of("Tom", 47, "Tim", 49, "Micha", 49);

    assertThat(nameAgeMap).isNotEmpty().doesNotContainKeys("John").
                           containsKey("Tim").contains(Map.entry("Micha", 49));
}
```

Throwable Assertions

Mitunter sollen in einem Testfall auch Exceptions abgeprüft werden. Bekanntermaßen bietet JUnit 5 dazu die Methode `assertThrows()`. Mit AssertJ wird auch diese Aufgabe wieder ein wenig einfacher. Betrachten Sie zum Verständnis folgendes Listing. Dort ist das Überprüfen des Typs der Exception sowie des Hinweistexts gezeigt:

```
@Test
void testException()
{
    ThrowingCallable action = () -> { throw new Exception("PENG!"); };

   assertThatThrownBy(action).isInstanceOf(Exception.class).
                              hasMessageContaining("PENG");
}
```

Das ist aber noch längst nicht alles. Man kann das Ganze auch noch deutlich ausgefeilter wie folgt gestalten:

```
@Test
void testException_2()
{
    ThrowingCallable action = () -> { throw new IOException("PENG!"); };

    assertThatExceptionOfType(IOException.class).isThrownBy(action)
                                                 .withMessage("%s!", "PENG")
                                                 .withMessageContaining("NG")
                                                 .withNoCause();
}
```

Hinweistexte für Assertions

JUnit 5 erlaubt es, die ausgegebenen Fehlermeldungen mit Informationen in Form von Hinweistexten anzureichern. Mit AssertJ ist das auch möglich, dazu dient die `as()`-Methode. Für Personen könnten wir etwa das Geburtsjahr und den Heimatort folgendermaßen prüfen – hier bewusst mit zwei Abweichungen, um die Texte in Aktion erleben zu können:

```
@Test
public void descriptionTest()
{
    final Person tim = new Person("Tim", LocalDate.of(1971, 3, 27), "Kiel");

    assertThat(tim.getBirthday().getYear()).as("prüfe Geburtsjahr").
                                            isEqualTo(1977);
    assertThat(tim.getCity()).as("prüfe Stadt für %s", tim.getName()).
                              isEqualTo(("Köln"));
}
```

Wie gesehen, kann man im Hinweistext auch Platzhalter nutzen, die dann über die als Parameter übergebenen Werte gefüllt werden.

Extraktion von Attributen

Zum Teil möchte man auf einzelne Attribute von Objekten in einer Collection zugreifen, nachfolgend auf den Namen von `Person`-Objekten. Mit `extracting()` lassen sich Attribute aus Objekten per Reflection auslesen, ähnlich wie es die Methode `map()` bei Streams ermöglicht:

```
final Person tim = new Person("Tim", LocalDate.of(1971, 3, 27), "Kiel");
final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Bremen");

var persons = List.of(tim, mike);

assertThat(persons).extracting("name")
                   .contains("Tim", "Mike")
                   .doesNotContain("Peter", "Franz");
```

Besonderheit beim Objektvergleich in AssertJ

Mitunter besitzt ein Objekt keine spezifische `equals()`-Methode. Dies kann für das Schreiben von Tests problematisch sein, weil alle relevanten Attribute per Hand abgefragt und verglichen werden müssen, nicht so bei AssertJ!

Bekanntermaßen können wir die Gleichheit mit der folgenden Behauptung prüfen:

```
assertThat(firstObject).isEqualTo(secondObject);
```

Selbstverständlich wird eine solche Prüfung immer dann fehlschlagen, wenn der verglichene Typ keine spezifische `equals()`-Methode anbietet und dadurch ein Referenzvergleich erfolgt. Schauen wir uns dies für die Klasse `SimplePerson` an:

```
public class AssertJSpecialComparisionsTest
{
    public static class SimplePerson
    {
        private final String name;
        private final float weight;
        private final int age;

        public SimplePerson(String name, float weight, int age)
        {
            this.name = name;
            this.weight = weight;
            this.age = age;
        }
    }

    // ...
}
```

Aufgrund des Fehlens einer eigenen `equals()`-Methode wird ein Vergleich mit `isEqualTo()` für unterschiedliche Objektreferenzen immer fehlschlagen. Wenn man aber Attribut für Attribut vergleicht, dann können wir bei Wertgleichheit eine Übereinstimmung ermitteln. Dabei hilft die Methode `usingRecursiveComparison()`:

```
@Test
void testAllAttributesEqual()
{
    SimplePerson tim = new SimplePerson("tim", 111.11f, 49);
    SimplePerson tim2 = new SimplePerson("tim", 111.11f, 49);

    assertThat(tim).usingRecursiveComparison().isEqualTo(tim2);
}
```

Fast noch ein wenig cooler ist die Möglichkeit, beim Vergleich von Objekten mit `isEqualToComparingOnlyGivenFields()` nur diejenigen Attribute anzugeben, die in einem speziellen Anwendungsfall von Bedeutung oder Interesse sind, nachfolgend Gewicht und Alter. So können dann zwei unterschiedliche Personen trotzdem als gleich gewertet werden, sofern die relevanten Attribute übereinstimmen:

```
@Test
void testGivenAttributesEqual()
{
    SimplePerson tim = new SimplePerson("tim", 111.11f, 49);
    SimplePerson tom = new SimplePerson("tom", 111.11f, 49);

    assertThat(tim).isEqualToComparingOnlyGivenFields(tom, "weight", "age");
}
```

Letztere Methode ist mittlerweile deprecated und man sollte einen Ausschluss von Attributen in Kombination mit der ersten Variante wie folgt nutzen:

```
@Test
void testGivenAttributesEqual_Recommended()
{
    SimplePerson tim = new SimplePerson("tim", 111.11f, 49);
    SimplePerson tom = new SimplePerson("tom", 111.11f, 49);

    assertThat(tim).usingRecursiveComparison().
                    ignoringFields("name").isEqualTo(tom);
}
```

Fazit

Wir haben einiges zu AssertJ gelernt. Das war ein schneller Einstieg, der Ihnen hoffentlich Lust auf weitere Experimente gemacht hat. Diese gelingen mit AssertJ fast wie von selbst, da durch das Fluent API immer alle Möglichkeiten angezeigt werden. Die tollen Erweiterungen in JUnit 5 werden mit AssertJ noch sinnvoll komplettiert.

20.7.3 MoreUnit

JUnit ist gut in Eclipse integriert und Testfälle lassen sich ausführen und sogar debuggen. Allerdings gibt es noch Raum für Verbesserungen: Beispielsweise sind die Tastaturkürzel zum Ausführen von Unit Tests (ALT+SHIFT+X,T) recht unhandlich. Um dieses und weitere Probleme kümmert sich das Eclipse-Plugin MoreUnit. Es steht frei im Eclipse Marketplace zur Installation bereit und bietet folgende Features:

- MoreUnit bietet Tastaturkürzel zum Ausführen (CTRL+R) und zum Hin-und-her-Wechseln zwischen Implementierung und Unit Test zu einer Klasse (CTRL+J). Existiert kein Test, so wird mit CTRL+J ein Dialog zum Anlegen eines korrespondierenden Unit Tests angeboten.
- Sofern eine Klasse Abhängigkeiten besitzt, kann man sich dafür automatisch Mock-Objekte (z. B. für Mockito, EasyMock, jMock 2) erstellen lassen.
- Man erhält eine Icon-Dekoration, sodass man direkt im Package Explorer durch einen kleinen grünen Punkt sieht, ob zu einer Klasse ein Test existiert.
- Beim Refaktorisieren werden Klassen und korrespondierende Testklassen automatisch synchron zueinander verschoben oder umbenannt.

Abbildung 20-20 *MoreUnit-Icon-Dekorator in Eclipse*

20.7.4 Infinitest

Infinitest ist ein Tool zur Unterstützung von kontinuierlichen Testläufen von Unit Tests.[18] Infinitest adressiert eine Unzulänglichkeit von normalen Unit Tests: Diese müssen nach Änderungen im Sourcecode in der Regel von Hand ausgeführt werden. Manchmal wird dies in der Hektik des Alltags vergessen oder ein relevanter Test übersehen, und so werden durch Änderungen neu eingeführte Fehler eventuell nicht aufgedeckt. Durch Infinitest findet man Fehler in der Regel schneller und sicherer. Es führt Unit Tests automatisch als Regressionstests direkt in der IDE aus. Die entsprechenden Tests werden intelligent ausgewählt, d. h., es werden nur die Tests ausgeführt, die von den Änderungen im Sourcecode betroffen sind. Bei auftretenden Fehlern erfolgt eine automatische Rückmeldung in der Statuszeile.

Dieses direkte Ausführen von Unit Tests noch innerhalb der IDE beim Entwickeln ist aus zweierlei Hinsicht erstrebenswert:

1. Man erhält eine sofortige Rückmeldung, ob man mit einer Änderung etwas kaputt gemacht hat.
2. Durch die instantane Rückmeldung erleichtert man nochmals die Arbeit mit Continuous Integration, da Fehler nicht erst durch den Build-Lauf und das Ausführen von Unit Tests auf dem Build-Server gefunden werden.

[18] Das Eclipse-Plugin kann unter `http://infinitest.github.io` bezogen werden.

20.7.5 JaCoCo

In diesem Kapitel habe ich bereits motiviert, dass eine durch Unit Tests abgesicherte Implementierung für qualitativ bessere Software sorgen kann. ***Die erzielbare Qualität hängt jedoch maßgeblich von der Güte der Implementierung selbst, aber auch von der der Tests sowie von der Anzahl der Testfälle ab***. Erinnern Sie sich an den Satz von James Coplien: »Tests don´t improve quality: developers do.« Gute Tests erhöhen allerdings nur die Qualität der durch diese tatsächlich überprüften, abgedeckten Programmteile. Man spricht in diesem Zusammenhang auch von ***Testabdeckung***. Diese von Hand zu bestimmen, ist nahezu unmöglich. Mithilfe von Tools kann man die Testabdeckung bis auf Ebene einzelner Anweisungen und Verzweigungen und ebenfalls aggregiert auf Methoden und Klassen ermitteln und so die Programmteile aufzeigen, die vollständig, nur teilweise oder auch gar nicht von Tests überprüft werden.

Das Tool JaCoCo hilft bei der Bestimmung der Testabdeckung. Zur Aufbereitung der Informationen klinkt sich JaCoCo automatisch und dynamisch in den erzeugten Bytecode des zu untersuchenden Programms ein, wodurch vom Entwickler selbst keine Änderungen am Programm vorzunehmen sind. Die ermittelten Informationen werden übersichtlich als Reports in HTML aufbereitet. Werfen wir für ein Beispiel einen Blick auf Abbildung 20-21.

quelltext > ch20_unittests.services Source Files Sessions

ch20_unittests.services

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
TaskUtilsV1		79%		50%	3	5	6	13	1	3	0	1
TaskUtilsV2		85%		100%	2	6	4	14	2	4	0	1
TaskCompareResults		86%		75%	3	7	0	8	2	5	0	1
TaskConverter		90%		n/a	1	2	2	4	1	2	0	1
Total	35 of 225	84%	3 of 12	75%	9	20	12	39	6	14	0	4

Created with JaCoCo 0.7.7.201606060606

Abbildung 20-21 *Von JaCoCo generierter Report*

Bewertung der Messergebnisse und Ermittlung fehlender Testfälle

Basierend auf den erzeugten Reports kann man ungetestete Bereiche der Software identifizieren und dafür bei Bedarf neue Tests erstellen. Ziel ist es, weitere Fehler durch Tests aufzudecken und die Qualität zu verbessern, nicht aber nur willkürlich die Testabdeckung zu erhöhen. Abgesehen von der gemessenen Testabdeckung ist es viel wichtiger, sinnvolle Testfälle zu erstellen. Was ist damit gemeint? Wenn das Ziel lediglich darin besteht, die Testabdeckung zu erhöhen, so werden oftmals Tests für einfache oder weniger kritische Teile erstellt. Das ist aber kontraproduktiv, weil dadurch Zeit verloren geht und vermutlich einige für ein zuverlässiges Verhalten der Klasse essenzielle Tests nicht erstellt werden, weil diese (kaum) zur Testabdeckung beitragen. ***Eine hohe Testabdeckung ist also kein Selbstzweck!***

Mit diesem Hinweis im Hinterkopf kann uns die Testabdeckung aber eine gute Hilfestellung sein, um diejenigen Programmstellen zu erkennen, die weiterer Tests bedürfen. Werden diese ergänzt, so steigt auch die Testabdeckung. Durch gut gewählte,

relevante Unit Tests verbessert sich außerdem die innere Qualität: Das ist dadurch begründet, dass ungetesteter Sourcecode potenziell einige versteckte Fehler enthält. Je weniger Testfälle existieren, desto weniger von diesen »schlummernden Zeitbomben« können gefunden werden. Das Erstellen von Tests sollte man nicht unüberlegt fortführen, sondern nur so lange, wie sich sinnvolle Testfälle ergänzen lassen.

Mit gewissem Aufwand lassen sich für wichtige Klassen etwa 50 – 70 % Testabdeckung erreichen. Gerade für komplexere Berechnungen sind höhere Werte wünschenswert. Eine Testabdeckung von 100 % ist allerdings kaum zu erzielen, da es Programmzeilen, Methoden oder im Extremfall sogar ganze Klassen gibt, die sich schwierig testen lassen. Ein Beispiel dafür sind unter anderem `catch`-Blöcke, die nicht angesprochen werden, oder Methoden, die leer implementiert werden, nur um ein Interface zu erfüllen. Abbildung 20-22 zeigt die `enum`-Aufzählung `TaskCompareResults`, die eine statische Prüfmethode `isOkValue(TaskCompareResults)` anbietet. Für diese sieht man, dass nicht alle Kombinationen der Oder-Verknüpfung durch Tests geprüft werden und diese Zeile dadurch nicht vollständig getestet worden ist. Daraus ergeben sich die dargestellten Werte für »Missed Instructions« und »Missed Branches«.

quelltext > ch20_unittests.services > TaskCompareResults — Sessions

TaskCompareResults

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
valueOf(String)		0%		n/a	1	1	1	1	1	1
values()		0%		n/a	1	1	1	1	1	1
static {...}		100%		n/a	0	1	0	4	0	1
isOkValue(TaskCompareResults)		100%		75%	1	3	0	1	0	1
TaskCompareResults(String, int, String)		100%		n/a	0	1	0	3	0	1
Total	9 of 64	86%	1 of 4	75%	3	7	0	8	2	5

Created with JaCoCo 0.7.7.201606060606

```
public enum TaskCompareResults
{
    OK("OK"),
    OK_EXISTING_TASK("OK_EXISTING_TASK"),
    FAILURE_DIFFERENT_TASK_CONTENT("FAILURE_DIFFERENT_TASK_CONTENT");

    final String text;

    private TaskCompareResults(final String text)
    {
        this.text = text;
    }

    public static boolean isOkValue(final TaskCompareResults value)
    {
        return value == OK || value == OK_EXISTING_TASK;
    }
}
```

Abbildung 20-22 *JaCoCo-Report für eine* `enum`*-Aufzählung*

Auswirkungen auf die Qualität

Wie eingangs erwähnt, wird die erzielbare Qualität der Software maßgeblich von der ***Güte*** der Testfälle und im geringeren Maße von deren *Anzahl* bestimmt. Wie fast immer gilt »***Qualität vor Quantität***«. Mit Tools wie JaCoCo bestimmt man aber lediglich die Testabdeckung, die normalerweise durch die Anzahl der Testfälle steigt. Man erhält jedoch keine Aussage über die Güte jedes einzelnen Tests. Werden nur »harmlose«

Testmethoden geschrieben, um eine möglichst hohe Testabdeckung zu erreichen, ist dies nicht hilfreich, sondern manchmal sogar kontraproduktiv. Entscheidend ist, dass die Testmethoden Fehlerfälle provozieren und somit helfen, Fehler aufzudecken. ***Eine geringere Testabdeckung mit guten Tests für komplexe und fehleranfällige Teile des Programms ist einer höheren Testabdeckung vorzuziehen, wenn diese nur dadurch erzielt wird, dass triviale `get()`- und `set()`-Methoden geprüft werden***.

Trotz 100 % Testabdeckung nicht fehlerfrei Ein letzter Punkt ist noch zu beachten: Eine hohe Testabdeckung besagt nicht, dass die Unit Tests *erfolgreich* durchlaufen werden, sondern lediglich, dass die Zeilen durch den Unit Test abgearbeitet werden. Das Testergebnis kann jedoch negativ ausfallen. Betrachten wir dies für folgende Methode `coverage100ButNPE()`, die auf Anregungen aus einem interessanten Artikel[19] basiert.

```
public class Coverage
{
    public String coverage100ButNPE(final boolean condition)
    {
        String value = null;
        if (condition)
        {
            value = String.valueOf(condition);
        }
        return value.trim();
    }
    // ...
```

Für diese Methode erreicht man mit zwei Testfällen 100 % Abdeckung:

```
public class CoverageTest
{
    final Coverage coverage = new Coverage(4711, "4711");

    @Test
    public void testCoverage100ButNPE1()
    {
        coverage.coverage100ButNPE(true);
    }

    @Test
    public void testCoverage100ButNPE2()
    {
        // Löst eine NullPointerException aus
        coverage.coverage100ButNPE(false);
    }
}
```

Dass eine Abdeckung von 100 % keine Aussage über Korrektheit oder Fehlerfreiheit erlaubt, soll hier verdeutlicht werden. Das erkennen wir, wenn wir den Test ausführen. Für den Übergabewert `false` wird eine `NullPointerException` ausgelöst. Das Programm enthält also trotz einer vermeintlich perfekten Testabdeckung von 100 %

[19] http://www.ibm.com/developerworks/java/library/j-cq01316/

dennoch einen eklatanten Fehler. Als Fazit lässt sich ziehen, dass eine Testabdeckung von 25 % durch Tests geschäftskritischer Funktionalität viel mehr Wert sein kann als 70 %, wenn dieser Wert nur durch Tests einfacher Methoden erreicht wird.

Ausführen von JaCoCo im Build

Um die Testabdeckung mit JaCoCo bestimmen zu können, binden wir es in unseren Build-Prozess ein. Dazu muss das Plugin samt einer kleinen Konfiguration in die Build-Datei `build.gradle` aufgenommen werden:

```
apply plugin: 'jacoco'

jacoco {
    toolVersion = "0.8.6"
    reportsDir = file("$buildDir/reports/jacoco/")
}

test.finalizedBy(project.tasks.jacocoTestReport)

task showJacocoTestReport << {
    File file = file("$buildDir/reports/jacoco/test/html/index.html")
    java.awt.Desktop.desktop.browse file.toURI()
}
```

Durch diese Erweiterungen wird JaCoCo in den Build-Lauf integriert. Die Testabdeckung wird durch Aufruf von `gradle clean test` berechnet. Dadurch werden die Klassen instrumentiert und die Unit Tests ausgeführt. Die als HTML aufbereiteten Reports werden im Verzeichnis `build/reports/jacoco/test/html/index.html` erzeugt. Als Erleichterung habe ich einen Task zur Anzeige geschrieben, den Sie wie folgt aufrufen können:

```
gradle showJaCoCoTestReport
```

Dadurch wird ein Webbrowser mit der Startseite der Ergebnisse geöffnet, wie dies exemplarisch Abbildung 20-23 zeigt.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ch06_collections.basics		0%		0%	158	158	465	465	111	111	41	41
ch99_swing_gui.painting_in_components.java2d		0%		0%	70	70	328	328	54	54	15	15
ch13_i18n.basics		0%		0%	74	74	282	282	48	48	20	20
ch06_collections.searchsortfilter		0%		0%	112	112	277	277	84	84	35	35
ch08_applikationsbausteine.configuration		0%		0%	108	108	291	291	69	69	25	25
ch04_javagrundlagen.fileio		0%		0%	112	112	312	312	73	73	21	21
ch09_multithreading.communication		0%		0%	81	81	375	375	58	58	28	28
ch22_optimierungen.memorybound		0%		0%	63	63	293	293	33	33	7	7
ch18_entwurfsmuster.behavioral		0%		0%	113	113	215	215	73	73	22	22
ch13_i18n.utils		0%		0%	69	69	258	258	55	55	15	15
ch10_advancedjava.reflection		0%		0%	60	60	234	234	37	37	11	11
ch11_dateandtime		0%		0%	52	52	274	274	51	51	22	22
ch99_swing_gui.advanced.jlist		0%		0%	72	72	199	199	62	62	22	22
ch99_swing_gui.intro		0%		0%	50	50	206	206	43	43	18	18
ch99_swing_gui.painting_in_components		0%		0%	47	47	207	207	32	32	14	14
ch10_advancedjava.serializable		0%		0%	92	92	202	202	58	58	12	12
ch99_swing_gui.advanced.jtable.modifications		0%		0%	61	61	204	204	50	50	14	14
ch18_entwurfsmuster.creational		0%		0%	61	61	183	183	51	51	18	18
ch07_bulk_operations		0%		0%	62	62	147	147	54	54	13	13

Abbildung 20-23 *JaCoCo-Report im Browser*

Hinweis: Java-14/15-Kompatibilität

JaCoCo ist seit Version 0.8.6 nun auch offiziell kompatibel mit Java 14. Zudem wird ein experimenteller Support für Java 15 und 16 geboten. EclEmma 3.1.3 basiert noch auf JaCoCo 0.8.5 und kommt daher mit Java 15 noch nicht zurecht.

20.7.6 EclEmma

Ähnlich wie JaCoCo ist EclEmma ein freies Tool zur Bestimmung der Testabdeckung und basiert auf JaCoCo. Der Vorteil von EclEmma gegenüber JaCoCo ist, dass die IDE nicht verlassen werden muss und die Testabdeckung direkt in der IDE beim Ausführen von Unit Tests berechnet und passend dargestellt wird. Einzige Voraussetzung ist die Installation des EclEmma-Plugins über den Eclipse Marketplace.[20] Danach bietet EclEmma folgende Eigenschaften:

- Es besitzt eine gute und nahtlose Integration in Eclipse.
- Die initialen Konfigurationsarbeiten sind kinderleicht: Tatsächlich muss nichts am Projekt verändert oder konfiguriert werden.
- Die Prüfung lässt sich einfach starten – analog zum Ausführen von Tests mit JUnit gibt es ein Menü RUN –> COVERAGE AS –> JUNIT TEST bzw. als Kontextmenü COVERAGE AS –> JUNIT TEST auf dem Projekt.
- Es werden diverse Metriken ermittelt und übersichtlich in einem speziellen View dargestellt. Darüber hinaus werden sowohl die von Tests abgedeckten als auch alle nicht abgedeckten Zeilen im Sourcecode-Editor farblich markiert.

Die Ergebnisse der berechneten Testabdeckung, ähnlich wie wir es schon für andere Klassen mit JaCoCo ermittelt haben, sehen wir in Abbildung 20-24 für EclEmma.

Hinweis: Java-14/15-Kompatibilität

Weil das aktuelle EclEmma 3.1.3 auf JaCoCo 0.8.5 basiert und dieses schon kompatibel mit Java 14 ist (allerdings laut Dokumentation nur experimentell), lassen sich auch Projekte mit Java-14-Features prüfen. Das aktuelle Java 15 wird derzeit von EclEmma nicht unterstützt. Das kommt vermutlich in der nächsten EclEmma-Version, die dann wohl JaCoCo in Version 0.8.6 einbindet und dadurch Java-15-Support bietet.

[20] Details finden Sie im Internet unter `http://www.eclemma.org/index.html`. Ende September 2020 ist Version 3.1.3 aktuell.

Controller.java | ParametrizedTes | ParametrizedTes | DiscountCalcula | NumberUtilsV3Nu | NumberUtilsV5Si | Nu

```
12 public final class NumberUtilsV4CombinedConditions
13 {
14     public static boolean isNumber(final String value)
15     {
16         Preconditions.checkNotNull(value, "parameter 'value' must not be null");
17
18         if (value.isEmpty())
19         {
20             return false;
21         }
22         for (int i = 0; i < value.length(); i++)
23         {
24             if (!(Character.isDigit(value.charAt(i))))
25             {
26                 return false;
27             }
28         }
29         return true;
30     }
31
32     private NumberUtilsV4CombinedConditions()
33     {}
34 }
```

Coverage

NEW-JP4-quelltext (14.01.2017 11:03:28)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
src/main/java	2.2 %	1'148	50'737	51'885
ch20_unittests.cobertura	100.0 %	38	0	38
ch17_refactorings	87.6 %	169	24	193
NumberUtilsV5Sign.java	93.3 %	42	3	45
NumberUtilsV4CombinedConditions.java	89.7 %	26	3	29
NumberUtilsV4CombinedConditions	89.7 %	26	3	29
isNumber(String)	100.0 %	26	0	26
NumberUtilsV4CombinedConditi...	0.0 %	0	3	3
NumberUtilsV3NullInputCorrected.java	87.2 %	34	5	39
NumberUtilsV2EmptyInputCorrected.java	85.7 %	30	5	35
NumberUtilsV1.java	83.3 %	25	5	30
NumberUtilsV6WithJdk.java	80.0 %	12	3	15
ch20_unittests.basics	81.9 %	118	26	144

Abbildung 20-24 *Von EclEmma generierter Report*

Vergleich EclEmma und JaCoCo

Insgesamt ist EclEmma eine sinnvolle Ergänzung zu JaCoCo. Ersteres hilft bei der alltäglichen Arbeit direkt in der IDE. Mit EclEmma erhält man schnell Rückmeldung und sieht schon während der Entwicklung, ob man auf dem richtigen Weg ist, seine Testabdeckung zu verbessern, und insbesondere auch, wo Bedarf für weitere Tests besteht.

JaCoCo kann dagegen im Build-Prozess eingesetzt werden. Die dabei produzierten HTML-Reports lassen sich historisieren und so Trends ermitteln.

20.8 Umstieg von JUnit 4 auf JUnit 5

In den nachfolgenden Abschnitten lernen wir einiges Wissenswertes zum Umstieg von JUnit 4 auf JUnit 5 kennen. Zuvor vielleicht so viel: Für uns als Nutzer hat sich gar nicht so viel getan, sodass Sie viele Dinge recht einfach übertragen können – unter der Motorhaube findet man jedoch eine Vielzahl an Neuerungen, etwa eine bessere Modularisierung, leichtere Erweiterbarkeit und eine gute Rückwärtskompatibilität für eine schrittweise Migration.

20.8.1 Erweiterung im Gradle-Build für JUnit-5-Tests

Ohne weitere Aufwände wird JUnit 5 von Eclipse und IntelliJ unterstützt. Das gilt prinzipiell auch für die Build-Tools Maven und Gradle. Für beide müssen ein paar Abhängigkeiten ergänzt werden. Für Gradle sind dies folgende, um sowohl die Neuerungen aus JUnit 5 als auch eine Ausführung von in JUnit 4 geschriebenen Tests für eine schrittweise Migration zu ermöglichen:

```
// JUnit 4 Support
testImplementation "junit:junit:4.13"
testRuntimeOnly("org.junit.vintage:junit-vintage-engine:5.7.0")

// JUnit 5
testImplementation "org.junit.jupiter:junit-jupiter-engine:5.7.0"
testRuntimeOnly "org.junit.platform:junit-platform-launcher:1.7.0"

// Migration Support to Enable Rules in JUnit 5
testImplementation "org.junit.jupiter:junit-jupiter-migrationsupport:5.7.0"
```

Wir erkennen die Bibliotheken für das Kompilieren und für das Ausführen. Bei Letzterem dient die Vintage-Engine zur Ausführung von JUnit-4-Tests.

20.8.2 Veränderungen in den Annotations

Wie schon vorab erwähnt, befinden sich die Annotations von JUnit 4 und JUnit 5 in unterschiedlichen Packages, wodurch ein Parallelbetrieb und eine Migration erleichtert wird.

In JUnit 4 stammen die Annotations aus folgenden Packages:

```
import static org.junit.Assert.*;
import org.junit.Test;
```

Mit JUnit 5 benötigt es folgende Imports:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
```

Durch die Beheimatung in verschiedenen Packages ist eine Koexistenz beider APIs problemlos möglich. Die modulare Architektur von JUnit 5 erlaubt es, abhängig vom

jeweiligen Test den richtigen Ausführungskontext auszuwählen, nämlich denjenigen für JUnit 4 oder für JUnit 5.

Die nachfolgende Tabelle stellt die Annotations aus JUnit 4 denen aus JUnit 5 gegenüber.

Tabelle 20-2 *Abweichende Annotations in JUnit 4 und JUnit 5*

JUnit 4	JUnit 5
`@Ignore`	`@Disabled`
`@Before`	`@BeforeEach`
`@After`	`@AfterEach`
`@BeforeClass`	`@BeforeAll`
`@AfterClass`	`@AfterAll`
`@Category`	`@Tag`
`@RunWith`	`@ExtendWith`
`@Rule`	`@ExtendWith` oder andere Konstrukte

Abweichende Parameterreihenfolge bei Assert

Der Umstieg von JUnit 4 auf JUnit 5 bereitet normalerweise wenig Probleme. Jedoch sollte man ein paar Dinge wissen: Zum einen besitzen die Prüfmethoden nun eine abweichende Reihenfolge, wenn man einen Hinweistext mitgeben möchte:

```
// JUnit 4
assertNotNull("Unexpected null value", null);

// JUnit 5
assertNotNull(null, "Unexpected null value");
```

Die in JUnit 5 verwendete Variante ist konsistenter als die von JUnit 4: Möchte man nachträglich einen Hinweistext mitgeben, so fügt man diesen einfach am Ende ein.

Abweichende Prüfung auf Exeptions

Zum anderen hat sich die Abfrage und Behandlung von Exceptions geändert (dies gilt auch für Timeouts und allgemein für JUnit Rules):

```
// JUnit 4
@Test(expected = java.lang.NumberFormatException.class)
public void testFailWithExceptionJUnit4()
{
    // Hier wird bewusst ein Fehler provoziert
    final int value = Integer.parseInt("Fehler simulieren!");
}

// JUnit 5
@Test
public void testFailWithExceptionJUnit5()
{
    assertThrows(java.lang.NumberFormatException.class, () ->
    {
        // Hier wird bewusst ein Fehler provoziert
        final int value = Integer.parseInt("Fehler simulieren!");
    });
}
```

20.8.3 Alternativen für JUnit Rules

Die nachfolgenden Abschnitte beschreiben mit JUnit Rules einige Erweiterungen zu JUnit, die in Version 4.7 hinzugefügt wurden und wie diese sich nun mit JUnit-5-Bordmitteln abbilden lassen.

Die Rule `TestName`

Ich beginne die Darstellung der JUnit Rules mit der unspektakulärsten und einfachsten, nämlich der JUnit Rule `TestName`, die wenig überraschend den Namen des momentan ausgeführten Testfalls, also der ausgeführten Methode, zurückliefert.

Äquivalent in JUnit 5 Etwas Ähnliches kann man in JUnit 5 mithilfe der Annotation `@DisplayName` oder durch Auslesen von Informationen, die mithilfe des Interface `TestInfo` injiziert wurden, erreichen:

```
public class NameRuleReplacementInJUnit5Test
{
    @Test
    @DisplayName("displayname unicode \u1234")
    public void injectParams(final TestInfo testInfo)
    {
        assertEquals("displayname unicode \u1234", testInfo.getDisplayName());
        assertEquals("injectParams", testInfo.getTestMethod().get().getName());
    }
}
```

Listing 20.23 *Ausführbar als* '**NAMERULEREPLACEMENTINJUNIT5TEST**'

Die Rule `Timeout`

Zum Teil soll die Ausführungsdauer von Testfällen begrenzt werden. In JUnit 4 kann man für jeden Testfall einzeln in der Annotation `@Test` über den Parameter `timeout` explizit einen Wert für dessen maximale Ausführungsdauer setzen. Alternativ lässt sich mit der JUnit Rule namens `Timeout` eine Zeitdauer festlegen, sodass alle Testfälle beim Überschreiten der angegebenen Timeout-Zeit abgebrochen und als Fehlschlag gewertet werden.

```
@Rule
public Timeout timeout = new Timeout(500, TimeUnit.MILLISECONDS);
```

Alle Testmethoden werden unabhängig von ihrer tatsächlichen Ausführungszeit nach dem Überschreiten des Timeouts abgebrochen und als Fehlschlag gewertet.

Äquivalent in JUnit 5 Mit JUnit 5 lassen sich Testfälle mit Timeouts durch die Methoden `assertTimeout()` und `assertTimeoutPreemptively()` ausführen. Bitte denken Sie an Folgendes: Während in JUnit 4 der Testfall abgebrochen wurde, falls dort eine Endlosschleife auftrat, geschieht dies in JUnit 5 nur, wenn Sie die Methode `assertTimeoutPreemptively()` nutzen.

```
public class TimeoutReplacementInJUnit5Test
{
    @Test
    void testAssertTimeout()
    {
        assertTimeout(Duration.of(500, ChronoUnit.MILLIS),
                      () -> longRunningAction());
    }

    @Test
    void testAssertTimeoutPreemptively()
    {
        assertTimeoutPreemptively(Duration.of(2, ChronoUnit.SECONDS),
                                  () -> loopForever());
    }


    private void longRunningAction() throws InterruptedException
    {
        for (int i = 0; i < 20; i++)
        {
            TimeUnit.MILLISECONDS.sleep(250);
        }
    }

    private void loopForever() throws InterruptedException
    {
        for (;;)
        {
            TimeUnit.SECONDS.sleep(1);
        }
    }
}
```

Listing 20.24 *Ausführbar als* '**TIMEOUTREPLACEMENTINJUNIT5TEST**'

Die Rule `ExpectedException`

Ab JUnit 4.7 gibt es die JUnit Rule `ExpectedException`, die sich mit der Verarbeitung von Exceptions in Unit Tests beschäftigt:

```
import org.junit.rules.ExpectedException;

public class ExpectedExceptionTest
{
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void illegalStateExceptionWithMessageTextExpected()
    {
        // würde die Exception bereits vor der eigentlichen Aktion
        // ausgelöst, so käme es zu einem Testfehler
        // throw new IllegalStateException("XYZ is not initialized");
        thrown.expect(IllegalStateException.class);
        thrown.expectMessage("XYZ is not initialized");

        throw new IllegalStateException("XYZ is not initialized");
    }
}
```

Listing 20.25 *Ausführbar als* '**EXPECTEDEXCEPTIONTEST**'

Im Listing sehen wir, dass standardmäßig die Annahme gilt, dass keine Exception ausgelöst wird, was man durch `ExpectedException.none()` festlegt. Das ist für die meisten Testfälle sinnvoll als Vorbelegung. Erwartet man in einem Testfall dagegen, dass eine Exception ausgelöst wird, so ist dies explizit durch Aufruf von `expect()` zu spezifizieren. Neben dem Typ von Exception lässt sich per `expectMessage()` auch der dort enthaltene Hinweistext prüfen.

Äquivalent in JUnit 5 Mit JUnit 5 wurde die Verarbeitung von Exceptions komplett umgestaltet und wird nun durch die Methode `assertThrows()` realisiert, die ein entsprechendes `Throwable`-Objekt zurückliefert, dessen Attributwerte normal über `assertEquals()` geprüft werden können – hier für den Hinweistext genutzt:

```
public class ExpectedExceptionJUnit5ReplacementTest
{
    @Test
    void illegalStateExceptionWithMessageTextExpected()
    {
        // würde die Exception bereits vor der eigentlichen Aktion ausgelöst,
        // so käme es zu einem Testfehler
        // throw new IllegalStateException("XYZ is not initialized");
        IllegalStateException thrown = assertThrows(IllegalStateException.class,
           () -> { throw new IllegalStateException("XYZ is not initialized"); })
                ;

        assertEquals("XYZ is not initialized", thrown.getMessage());
    }
}
```

Listing 20.26 *Ausführbar als* '**EXPECTEDEXCEPTIONJUNIT5REPLACEMENTTEST**'

Die Rule `TemporaryFolder`

Mithilfe der Rule `TemporaryFolder` erlaubt JUnit 4 die Ausführung von Unit Tests unter Bereitstellung eines jeweils eigenen temporären Verzeichnisses für einen Testfall. Als Beispiel legen wir nun ein Unterverzeichnis sowie eine Datei in einem temporären Verzeichnis an und prüfen dessen Inhalt sowie die Existenz der Datei:

```
public class TemporaryFolderTest
{
   @Rule
   public TemporaryFolder folder = new TemporaryFolder();

   @Test
   public void testUsingTempFolder() throws IOException
   {
      final File subFolder = folder.newFolder("subfolder");
      final File createdFile = folder.newFile("abc.txt");

      // Ermittle den Inhalt des Temp-Folders, normales File-API
      final String[] dirContents = folder.getRoot().list();

      // Prüfe, ob File existiert und Verzeichnisinhalt übereinstimmt
      // assertEquals() auf Set damit die Reihenfolge keinen Einfluss hat
      assertTrue(createdFile.exists());
      assertEquals(Set.of("subfolder", "abc.txt"), Set.of(dirContents));
   }
}
```

Listing 20.27 *Ausführbar als* '**TEMPORARYFOLDERTEST**'

Äquivalent in JUnit 5 Auch die Interaktion mit dem Dateisystem ist in JUnit 5 anders als in JUnit 4 gelöst. Hierzu gibt es nun die Annotation `@TempDir`:

```
public class TemporaryFolderJUnit5ReplacementTest
{
   @Test
   public void testUsingTempFolder(@TempDir File folder) throws IOException
   {
      final File subFolder = new File(folder, "subfolder");
      subFolder.mkdir();
      final File createdFile = new File(folder, "abc.txt");
      createdFile.createNewFile();

      // Ermittle den Inhalt des Temp-Folders, normales File-API
      final String[] dirContents = folder.list();

      // Prüfe, ob File existiert und Verzeichnisinhalt übereinstimmt
      // assertEquals() auf Set damit die Reihenfolge keinen Einfluss hat
      assertTrue(createdFile.exists());
      assertEquals(Set.of("subfolder", "abc.txt"), Set.of(dirContents));
   }
}
```

Listing 20.28 *Ausführbar als* '**TEMPORARYFOLDERJUNIT5REPLACEMENTTEST**'

20.8.4 Veränderungen bei parametrierten Tests

Kommen wir nun zu parametrierten Tests, die dabei helfen, eine Vielzahl an Wertebelegungen zu testen. Durch den Einsatz parametrierter Tests kann man vermeiden, für jede Wertebelegung eine eigene Testmethode erstellen zu müssen und so die Testklasse aufzublähen sowie unübersichtlich und schwierig wartbar zu machen.

Parametrierte Tests in JUnit 4

Um in JUnit 4 einen parametrierten Test zu nutzen, sind folgende Schritte durchzuführen:

1. Die Testklassen mit `@RunWith(Parameterized.class)` annotieren.
2. Einen Konstruktor erstellen, der die Testdaten entgegennimmt. Als Parameter benötigt dieser Konstruktor jeweils Eingabewert und erwartetes Ergebnis.
3. Attribute für jeden der Übergabeparameter des Konstruktors definieren.
4. Eine spezielle Datenlieferanten-Methode erstellen. Diese muss als `public` und `static` definiert sein und als Rückgabe ein `Iterable<Object[]>` besitzen.
5. Eine Testmethode, die dann mit den Werten »gefüttert« wird, erstellen.

Wenn man den Test wie gerade beschrieben implementiert, sieht man, dass doch recht viel Sourcecode entsteht und dies zulasten der Einfachheit und teilweise der Verständlichkeit geht – allerdings erfolgt hier eine gute Zuordnung zwischen Eingabe- und Ergebniswert. Betrachten wir zur Verdeutlichung die Konvertierung von Hexadezimalzahlen:

```
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class ParameterizedTestExample
{
    private char hexDigit;
    private int expected;

    // Jeder Methodenparameter und auch der erwartete Wert
    // werden als Parameter an den Konstruktor übergeben
    public ParameterizedTestExample(final char hexDigit,
                                    final int expectedResult)
    {
        this.hexDigit = hexDigit;
        this.expected = expectedResult;
    }

    @Parameterized.Parameters(name= "{index}: hexInput (0) => value {1}")
    public static Iterable<Object[]> hexInputs()
    {
       return Arrays.asList(new Object[][]
       {
         {'0', 0 }, {'1', 1 }, {'2', 2 }, {'3', 3 }, {'4', 4},
         {'5', 5 }, {'6', 6 }, {'7', 7 }, {'8', 8 }, {'9', 9},
         {'A', 10}, {'B', 11}, {'C', 12}, {'D', 13}, {'E', 14}, {'F', 15}
      });
    }
```

```
    @Test
    public void testHexCodedByteValueToInt_With_AllValidInputs()
    {
        final int result = Controller.hexCodedByteValueToInt((byte) hexDigit);

        assertEquals(expected, result);
    }
}
```

Listing 20.29 *Ausführbar als* '**PARAMETRIZEDTESTEXAMPLE**'

Zwischenfazit Zwar sind parametrierte Tests grundsätzlich eine gute Idee, jedoch sind sie in JUnit 4 eher unbefriedigend umgesetzt, weil sich dadurch die Komplexität der Testklasse erhöht und sich Tests nicht sonderlich elegant formulieren lassen. Schlimmer noch: Man kann in einer solchen Testklasse keine Testmethoden ergänzen, die ohne Parametrierung ausgeführt werden sollen. Darüber hinaus ist es nicht möglich, für verschiedene Methoden unterschiedliche Parametrierungen zu verwenden.

Äquivalent in JUnit 5

JUnit 5 adressiert viele Schwachstellen parametrierter Tests in JUnit 4. Dafür wurde die Annotation `@ParametrizedTest` eingeführt:

```
public class ParametrizedTestJUnit5ReplacementTest
{
    @ParameterizedTest(name = "{index}: hex {0} => dec {1}")
    @CsvSource({ "0, 0", "1, 1", "2, 2", "3, 3", "4, 4",
                 "5, 5", "6, 6", "7, 7", "8, 8", "9, 9",
                 "A, 10", "B, 11", "C, 14", "D, 13", "E, 12", "F, 15" })
    void testHexToDec(final char hexDigit, final int expected)
    {
        final int result = Controller.hexCodedByteValueToInt((byte) hexDigit);

        assertEquals(expected, result);
    }
}
```

Listing 20.30 *Ausführbar als* '**PARAMETRIZEDTESTJUNIT5REPLACEMENTTEST**'

20.8.5 Alternative zur Hamcrest-Integration in JUnit 4

Weil der Einsatz von Hamcrest die Lesbarkeit von Assertions deutlich erhöht, wurde es teilweise in JUnit 4 integriert. Das umfasst insbesondere die Methode `assertThat()` zum Aufstellen einer Testbehauptung sowie verschiedene Methoden zum Formulieren einzelner Bedingungen.

In JUnit 5 ist Hamcrest nicht mehr integriert und muss als externe Bibliothek eingebunden werden, wie es in Abschnitt 20.7.1 beschrieben wurde.

Alternativ bietet es sich an, AssertJ für die lesbare Gestaltung von Testbehauptungen zu verwenden. Dies wurde in Abschnitt 20.7.2 bereits thematisiert.

20.9 Fazit

Wir haben in diesem Kapitel gelernt, wie sich Unit Tests Schritt für Schritt entwickeln lassen und sich daraus Verbesserungen im Applikationscode ergeben können. Bei Änderungen im Sourcecode empfiehlt es sich, für wichtige Teile begleitend Unit Tests zur Absicherung der Änderungen zu schreiben, um sich vor (leichten) Fehlern, insbesondere Flüchtigkeits- und Denkfehlern, zu schützen.

Zum Abschluss möchte ich noch einmal wiederholen, dass der Einsatz von Unit Tests folgende positive Auswirkungen hat: Unit Tests ...

- geben mehr Sicherheit, wenn neue Funktionalität benötigt wird oder Änderungen erfolgen müssen;
- lassen sich sehr gut schrittweise erweitern;
- helfen dabei, ein besseres Verständnis für den Applikationscode zu entwickeln;
- ermöglichen es, sinnvolle Kommentare einzufügen und Funktionalität durch Testfälle zu dokumentieren;
- helfen dabei, Abhängigkeiten von anderen Packages und Klassen zu reduzieren.

20.10 Weiterführende Literatur

Dieses Kapitel hat einen Einblick in das Thema Unit-Testen gegeben. Weiterführende Informationen finden Sie in den folgenden Büchern:

- »**Unit-Tests mit JUnit**« von Andrew Hunt und David Thomas [35]
  Ein empfehlenswertes Buch, das Unit-Testen motiviert und vom Einstieg bis zur Gestaltung komplexerer Testfälle mit guten Tipps und Ratschlägen weiterhilft.
- »**Unit Tests mit Java**« von Johannes Link [49]
  Dieses Buch beschreibt den Test-First-Ansatz beginnend mit den Grundlagen von Unit Tests. Es geht auch auf Stub- und Mock-Objekte ein und behandelt zudem das Testen von Datenbankabfragen sowie den Test nebenläufiger Programme.
- »**Next Generation Java Testing: TestNG and Advanced Concepts**« von Cédric Beust und Hani Suleiman [4]
  In diesem Buch wird das Thema Testen detailliert behandelt. Es beginnt mit JUnit und TestNG und geht auch auf komplexere Testszenarien mit Applikationsservern, Datenbanken usw. ein.
- »**Practical Unit Testing with JUnit and Mockito**« von Tomasz Kaczanowski [44]
  Dieses Buch von Tomasz Kaczanowski führt zunächst schrittweise und gründlich in die Thematik Unit-Testen ein. Anschließend geht es ausführlicher auf TDD, aber auch auf Beschränkungen davon ein. Der Hauptteil widmet sich dann Test-Doubles, im Speziellen Mocks und Mockito als Framework.
- »**Effective Unit Testing**« von Lasse Koskela [47]
  Lasse Koskela zeigt in seinem Buch, wie man lesbare, wartbare und vertrauenswürdige Tests schreibt. Abgerundet wird das Ganze mit einigen zum Teil kontroversen Gedanken zum Design von Tests.

21 Codereviews

In diesem Kapitel stelle ich sogenannte Codereview-Meetings, kurz Codereviews, vor und beschreibe zunächst in Abschnitt 21.1, was man darunter versteht. Weil die Durchführung von Codereviews nicht immer ganz unproblematisch ist, nenne ich in Abschnitt 21.2 einige typische Probleme und zeige Lösungsmöglichkeiten auf. In Abschnitt 21.3 motiviere ich, warum Codereviews einen wichtigen Teil der Qualitätssicherung bilden. Abschließend wird in Abschnitt 21.4 eine Checkliste vorgestellt, die bei der Durchführung eigener Codereviews als Leitlinie dienen kann.

21.1 Definition

Unter ***Codereviews*** versteht man Meetings von mehreren Entwicklern, in denen Sourcecode-Abschnitte mit dem Ziel betrachtet werden, mögliche Softwaredefekte aufzuspüren. Nach dem Motto »Viele Augen sehen mehr als die eigenen zwei« nutzt man das Potenzial und Wissen anderer Personen, um Probleme im vorliegenden Sourcecode aufzudecken. Die Teilnahme einiger projektfremder Entwickler ist durchaus wünschenswert; häufig kann man von ihrem Wissen profitieren. Vielleicht werden bislang nicht bedachte Themen wie Synchronisation beim Einsatz von Multithreading oder die Nutzung anderer Datenstrukturen angesprochen.

Durchführung

Für ein Codereview in seiner einfachsten Form reicht es bereits, dass beim Pair Programming der jeweils zuschauende Entwickler dem anderen kritisch beim Tippen über die Schulter schaut und Kommentare abgibt. Das Codereview findet dann aber nur nebenbei statt. Alternativ kann man auch ganz gezielt mit einer Gruppe von zwei oder drei Entwicklern einige Klassen oder Methoden direkt am Rechner durchgehen. Häufig ist es für eine konzentrierte Analyse sinnvoller, die zu untersuchenden Teile des Sourcecodes auszudrucken oder diese am Beamer zu begutachten und sich dazu in einem ruhigen Besprechungsraum zusammenzusetzen. Diese Form des Codereviews kann man auf eine größere Anzahl von Entwicklern ausweiten – wobei ich durchaus Meetings mit bis zu 20 beteiligten Personen erlebt habe. Sinnvoll und produktiver sind allerdings die zuvor beschriebenen Meetings in kleinerer Runde mit zwei bis fünf Teilnehmern. Bei einer größeren Teilnehmerzahl nimmt in der Regel die Konzentration ab und so-

ziale Aspekte treten mehr in den Vordergrund: Aus einem zwanglosen und informellen Gespräch wird ein eher schwergewichtiger Vorgang. Dieser Effekt wird verstärkt, wenn das gesamte Vorgehen formaler wird, etwa mit diversen zu erstellenden Protokollen und speziellen Formularen. Dies ist meiner Meinung nach oftmals eher hinderlich.

Vorteile

Um Softwaredefekte aufzudecken, zu beheben und somit die Qualität des Sourcecodes auf einem hohen und möglichst einheitlichen Niveau zu halten, bieten sich regelmäßig stattfindende Codereviews an. Ergänzend zu den täglichen (Mini-)Reviews beim Pair Programming können separat stattfindende Codereviews Themengebiete gründlicher beleuchten und mehr in Richtung Konzeptreview gehen. Oftmals ist es dazu hilfreich, den zu begutachtenden Sourcecode zusätzlich auch konzeptuell vorzustellen. Verweise auf eingesetzte Entwurfsmuster und ein Systemschaubild können viele erklärende Worte ersetzen und gleichzeitig als Diskussionsbasis dienen. So entsteht im Laufe der Zeit ein gemeinsames Verständnis von Softwarequalität, wodurch man Problemen vorbeugen kann. Wichtig ist es aber auch, die erarbeiteten Ergebnisse zeitnah als Fehlerbehebungen in den Sourcecode einfließen zu lassen und nicht nur den präventiven Effekt zu nutzen.

Tipp: Zeitpunkt der Codereviews

Codereviews können entweder als sogenanntes ***Pre-Commit-Review*** oder als sogenanntes ***Post-Commit-Review*** durchgeführt werden. Bei Ersterem erfolgt die Sourcecode-Überprüfung vor dem Einspielen in ein Repository, beim Letzterem nachträglich. In der Praxis finden größere Codereview-Meetings allerdings viel seltener als Commits statt. Häufig sieht man daher eine Mischform: Vor einem Commit kann ein spontanes Codereview in kleinerem Rahmen und Umfang stattfinden, um gemäß dem Vieraugenprinzip Flüchtigkeitsfehler zu vermeiden. Größere oder regelmäßige Meetings folgen dagegen in der Regel nicht speziellen Commit-Zyklen, sondern dienen als »Nachsorge«-Untersuchung, einerseits, um Defekte aufzudecken, und andererseits, um einen Coding-Standard sicherzustellen.

Psychologische Aspekte und Probleme

Wie bereits angedeutet, besitzen Codereviews auch eine soziale und psychologische Komponente. Der Sourcecode eines Entwicklers wird untersucht. Die Schwierigkeit besteht darin, notwendige Kritik zu äußern, ohne dies vorwurfsvoll oder belehrend zu tun. Ansonsten kann sich der reviewte Entwickler schnell in eine Verteidigungsposition gedrängt fühlen. Auch die Anzahl der Teilnehmer an einem solchen Meeting kann dies auslösen: Je mehr Leute teilnehmen, desto offizieller wird der Charakter des Reviews und es entsteht schnell eine Konfrontationssituation vom betroffenen Entwickler zu den Kritikern. Die positiven Effekte des Codereviews kommen dann nicht zum Tragen und ein solches Meeting schadet eher, denn es wird kaum mehr gelernt. Zudem besteht die

Gefahr, dass sich bei mehreren ähnlich ablaufenden Meetings eine Kontra-Codereview-Stimmung entwickelt und niemand mehr freiwillig seinen Sourcecode vorstellen möchte. Es ist Aufgabe des Veranstalters, als Moderator zu fungieren und dafür zu sorgen, dass Reviews in einer angenehmen Atmosphäre ablaufen. Der folgende Abschnitt geht auf mögliche Probleme und deren Lösung bei der Durchführung von Codereviews ein.

21.2 Probleme und Tipps zur Durchführung

Codereviews zu veranstalten ist mitunter eine undankbare Aufgabe, denn Qualitätssicherung (QS) muss häufig anderen Dingen weichen, insbesondere dann, wenn es im Projektzeitplan eng wird. Das liegt auch daran, dass Projektleiter nicht immer leicht von den Vorteilen von QS-Maßnahmen zu überzeugen sind. Das gilt im Speziellen, wenn dies den Projektplan zunächst etwas verzögert. Diese Einstellung ist jedoch eher kurzsichtig und kontraproduktiv: QS-Maßnahmen wie Unit Tests und Codereviews können in hektischen Projektphasen nämlich sogar besonders nützlich sein, weil sie das Verständnis für realisierte Konzepte vertiefen und für weniger Aufregung bei Änderungen trotz knapper Zeit sorgen können. Wie bereits in Kapitel 20 erwähnt, führt zudem eine gute Testabdeckung durch Unit Tests zu mehr Vertrauen und Sicherheit, da mögliche Probleme in Erweiterungen schnell sichtbar werden. Meint man, auf QS-Maßnahmen verzichten zu können, so erhöht sich in der Regel der nachträgliche Testaufwand für Integrations- und Applikationstests beträchtlich. Auch die Fehlerrate beim Einsatz der Applikation und die daraus resultierenden Wartungsaufwände steigen. Ähnlich zu einem Darlehen steht anfangs nur vermeintlich mehr Zeit für Implementierungsaufgaben zur Verfügung. Tatsächlich wird aber nur kurzfristig Zeit eingespart, die später in Form von umfangreicheren Fehlerkorrekturen »zurückgezahlt« werden muss. Bei nicht vollständig umgesetzter Funktionalität spricht man auch von technischen Schulden oder ***Technical Debt***.

Es gibt aber auch andere Gründe für eine Ablehnung von Codereviews: Von einigen Kollegen werden nahezu alle Meetings als lästiges Übel angesehen – da macht dann auch das Codereview keine Ausnahme. Und so sprechen gegen die Teilnahme immer einige gute und weniger gute Gründe, wie etwa Termindruck oder generelles Desinteresse. Selbst bei einer anfänglichen Befürwortung kann die Motivation zur Teilnahme an Codereviews im Laufe der Zeit abnehmen. Zudem ist es häufig so, dass genau die Leute am meisten von Codereviews profitieren würden, die sich dagegen aussprechen.

Werfen wir kurz einen Blick auf mögliche Ursachen der genannten Probleme und erarbeiten einige Vorschläge, wie man diesen gegensteuern kann.

1. **Sourcecode-Qualität**
   Problem: Die vorgestellten Programmteile haben häufig keine besonders gute Qualität und man diskutiert immer wieder bereits hinlänglich bekannte Fehler (z. B. schlechte Struktur, fehlende Parameterprüfungen öffentlicher Methoden, keine Abfrage auf `null`, wo nötig, (viel) zu lange Methoden, fehlende Dokumentation).

Abhilfe: Lästige Routinearbeiten – wie das Prüfen auf Einhaltung von Coding Conventions – sollten im Vorfeld durch Tools, etwa die bereits in Abschnitt 19.4 vorgestellten Checkstyle, PMD oder SonarQube bzw. SonarLint, durchgeführt werden. Damit reduziert sich der rein formale und syntaktische Aspekt der Sourcecode-Qualität und die Teilnehmer können sich vermehrt mit Fragen zu Design und Konzepten beschäftigen, ohne sich in Nebensächlichkeiten zu verlieren.

2. **Sourcecode-Umfang**
   Problem: Die vorgestellten Abschnitte sind meistens deutlich zu lang.
   Abhilfe: Der Umfang sollte auf einige 100 Zeilen oder ausgedruckt auf wenige, d. h. etwa eine bis drei DIN-A4-Seiten beschränkt werden. Ansonsten lassen sich die vorgestellten Programmstellen und die möglicherweise darin enthaltenen Probleme nicht gründlich analysieren. Dann ist der Gegenwert im Verhältnis zum Zeitaufwand zu gering.
3. **Themenmix**
   Problem: Der Lerneffekt bleibt aus, da zu selten spannende Konzepte vorgestellt und stattdessen immer wieder lange »Sourcecode-Wüsten« durchforstet werden.
   Abhilfe: Es sollte vorwiegend thematisch ausgewählter, interessanter Sourcecode untersucht werden, der zudem den zuvor genannten Umfang nicht überschreitet. Aber nicht nur der Sourcecode an sich, sondern auch die realisierten Konzepte, also nicht nur das »Wie«, sondern auch das »Warum so und nicht anders«, sind von Interesse. Konzept-, Design- und Architekturreviews mit einer Vorstellung von Entwurfsmustern sowie von Tipps und Tricks können die Meetings bereichern. Hier ist man als Veranstalter gefragt, die richtige Mischung zu finden.
4. **Veranstaltungsdauer**
   Problem: Das Meeting zieht sich über mehr als eine Stunde hin; im Extremfall über mehr als zwei Stunden.
   Abhilfe: Bei Codereview-Meetings geht es um konzentriertes Analysieren. Mehr als eine Stunde ist das kaum möglich. Daher sollte die Meetingdauer auf maximal eine Stunde beschränkt oder dann zumindest eine Pause eingelegt werden. Können nicht alle Probleme diskutiert werden, so sollten diese in einem Folgemeeting wieder aufgegriffen werden. Eine Zeitbegrenzung bietet außerdem den Vorteil, dass selbst bei Projektstress eine Teilnahme möglich und auch bei schwierigeren Themen noch ausreichend Konzentration vorhanden ist.
5. **Gegenwert**
   Problem: Es kommt vor, dass mit großer Mannschaft und viel Aufwand Codereviews durchgeführt werden, ohne jedoch im Nachhinein die gefundenen Defekte zu beheben. Werden Resultate aber nicht verwendet, stellt sich die Frage nach dem Nutzen des Meetings. Es entsteht schnell der Eindruck, dass geäußerte Kritik keine Konsequenzen auf die Sourcecode-Qualität hat. Dies erzeugt unter Umständen das Gefühl, zu viel Zeit für zu wenig Gegenwert aufzubringen. Der Eindruck verstärkt sich vor allem in »heißen« Projektphasen.

Abhilfe: Codereviews dienen der aktiven Verbesserung von Softwarequalität. Daher sollten festgestellte Defekte möglichst schnell aus dem untersuchten Sourcecode entfernt werden. Falls es die Änderungen und Anmerkungen sinnvoll erscheinen lassen, kann der überarbeitete Sourcecode später einem erneuten Review unterzogen werden. Manchmal findet man bei einem solchen iterativen Vorgehen noch einige weitere Verbesserungsmöglichkeiten.

6. **Teilnahme**
   Problem: Die Erfahrung zeigt, dass bei einer freiwilligen Teilnahme immer wieder anderen Arbeiten mehr Priorität eingeräumt wird.
   Abhilfe: Codereviews sollten, wie andere Meetings auch, Pflichttermine sein. Einerseits verhindert man dadurch Ausreden von Entwicklern, die eine Teilnahme vermeiden wollen. Andererseits erschwert dies Projektleitern, einzelne Entwickler von der Teilnahme abzuhalten.

21.3 Vorteile von Codereviews

Dieser Abschnitt zeigt exemplarisch die positiven Auswirkungen von Codereviews auf die Softwarequalität anhand von Beispielen aus der Praxis. Basierend auf den Erkenntnissen des vorangegangenen Abschnitts sorgen wir durch den Einsatz von Analysetools bereits dafür, dass ein Mindeststandard bezüglich der Sourcecode-Qualität erreicht wird, bevor wir mit den Codereviews beginnen. Formale Aspekte wie Namenskonventionen, Methodenlänge usw. lassen sich mit Tools gut überprüfen, semantische Aspekte und die Korrektheit der eingesetzten Algorithmen sind dagegen durch Tools nur schwer oder gar nicht überprüfbar. Durch Codereviews ist beides jedoch leicht möglich. Betrachten wird dazu einige Beispiele, um dies zu verdeutlichen.

Beispiel 1 Beginnen wir mit der Klasse `DepartureLineTimeComparator`, die zwei Abfahrten vom Typ `Departure` nach Liniennummer und Abfahrtszeit vergleicht und wie folgt definiert ist:

```
public final class DepartureLineTimeComparator implements Comparator<Departure>
{
    private final Comparator<Departure> lineComparator = new LineComparator();
    private final Comparator<Departure> timeComparator = new TimeComparator();

    public int compare(final Departure departure1, final Departure departure2)
    {
        int result = lineComparator.compare(departure1, departure2);
        if (result == 0)
        {
            timeComparator.compare(departure1, departure2);
        }
        return result;
    }
}
```

Diese Klasse ist kurz und übersichtlich, enthält aber dennoch einen versteckten Fehler. Es ist intuitiv klar, dass eine Hintereinanderausführung der beiden `Comparator<Departure>`-Instanzen durchgeführt werden soll. Ein aufmerksamer Blick zeigt schnell den Fehler: Beim Aufruf von `compare(Departure, Departure)` der `TimeComparator`-Instanz fehlt eine Zuweisung an die Variable `result` bzw. alternativ eine `return`-Anweisung, wodurch lediglich die Linieninformationen verglichen wird. Bei einer Prüfung in einem Codereview wäre dieser Fehler höchstwahrscheinlich nicht übersehen worden. Eine Korrektur fällt nicht schwer:

```
public int compare(final Departure departure1, final Departure departure2)
{
    int result = lineComparator.compare(departure1, departure2);
    if (result == 0)
    {
        // KORREKTUR //
        result = timeComparator.compare(departure1, departure2);
    }
    return result;
}
```

Beispiel 2 Ein anderes Beispiel ist folgende Zuweisung:

```
boolean onlyTimeChanged = hasSameLine() && hasSameEndPos() && hasSameEndPos();
```

Bei einem Codereview wäre bestimmt aufgefallen, dass man nicht zweimal die Endposition durch Aufruf der Methode `hasSameEndPos()` vergleichen möchte, sondern eigentlich die Start- und die Endposition, wodurch sich folgende Korrektur ergibt:

```
boolean onlyTimeChanged = hasSameLine() && hasSameStartPos() && hasSameEndPos();
```

Beispiel 3 Neben solchen Flüchtigkeitsfehlern kann man mit Codereviews auch Fehler im Algorithmus aufdecken, wie in diesem Beispiel:

```
private static volatile boolean shouldTerminate = false;

private static void work()
{
    final Telegram telegram = waitForTelegram();
    if (telegram.getType().equals(TELEGRAM_TYPE_XYZ))
    {
        telegram.performTelegramHandling();
    }
    if (telegram.getType().equals(TELEGRAM_TYPE_QUIT))
    {
        terminate();    // Seiteneffekt, setzt shouldTerminate-Flag
    }

    if (!shouldTerminate())
    {
        work();
    }
}
```

Das Beispiel war in der Realität noch deutlich komplizierter, sodass ich einige Zeit gebraucht habe, die Intention zu erkennen: Es wird hier lediglich eine `while`-Schleife durch Rekursion nachgebaut. Allerdings ist die Anzahl der Schleifendurchläufe durch die Größe des Stacks bestimmt. Nach einiger Zeit kommt es zu einem `java.lang.StackOverflowError`. Bis dahin kann es aber durchaus längere Zeit dauern, da immer nur eine Methode auf den Stack gelegt wird. Auf meinem Rechner können etwa 12.000 Aufrufe der Methode erfolgen. Wenn man von Methodenlaufzeiten von einigen Sekunden ausgeht, so läuft das Programm durchaus einige Stunden scheinbar fehlerfrei, bis es dann auf einmal abstürzt.

Eine Korrektur fällt einfach, besitzt zum einen keine Laufzeitbeschränkungen mehr und ist zum anderen intuitiv verständlich:

```
private static volatile boolean shouldTerminate = false;

private static void workCorrected()
{
    while (!shouldTerminate())
    {
        final Telegram telegram = waitForTelegram();
        if (telegram.getType().equals(TELEGRAM_TYPE_XYZ))
        {
            telegram.performTelegramHandling();
        }
        if (telegram.getType().equals(TELEGRAM_TYPE_QUIT))
        {
            terminate();    // Seiteneffekt, setzt shouldTerminate-Flag
        }
    }
}
```

Fazit

Kleine Fehler dieser Art »verstecken« sich leider häufiger im Sourcecode. Idealerweise werden sie schon beim Testen erkannt, denn ansonsten machen sie später beim Einsatz des Programms Probleme. ***Codereviews sind hervorragend dazu geeignet, solche »schlummernden Zeitbomben« aufzudecken und zu beheben, bevor sie sich als Fehler (beim Kunden) manifestieren.***

Obwohl Codereviews viel zur Qualitätssicherung beitragen können, so ist es jedoch aufgrund des Umfangs an Sourcecode häufig nicht möglich, jede Zeile zu reviewen. Damit werden logischerweise auch nicht alle Fehler entdeckt. Allerdings sollte es das Ziel sein, möglichst viel vom tatsächlich relevanten Sourcecode zu reviewen. Dies wird erleichtert, wenn kein Commit ohne ein vorheriges Codereview stattfinden darf. Selbst wenn man dies nicht ganz so strikt umsetzt, baut sich durch Codereviews mit der Zeit ein gemeinsames Qualitätsverständnis auf, was wiederum die Wahrscheinlichkeit für Fehler in neu entwickeltem Sourcecode reduziert. Auch Pair Programming kann zur Verbesserung der Sourcecode-Qualität beitragen. Darüber hinaus helfen Tools wie etwa die bereits in Abschnitt 19.4 vorgestellten Tools PMD oder SonarQube und SonarLint, entwicklungsbegleitend Fehler aufzudecken und zeitnah beheben zu können.

21.4 Codereview-Checkliste

Bei der Durchführung eigener Codereviews kann folgende Checkliste helfen, einige wichtige Aspekte beim Analysieren von Sourcecode im Blick zu behalten. Es bietet sich an, diese Liste zu kopieren und vor einem Codereview an alle Teilnehmer zu verteilen.

1. **Variablen und Namensgebung**
   Sind Namen sinnvoll gewählt und konsistent? Werden Namensrichtlinien beachtet? Vermeide den Gebrauch einer Variablen für verschiedene Zwecke!
2. **Variableninitialisierung**
   Sind alle Variablen möglichst lokal definiert und korrekt initialisiert? Sind Konstanten `final` definiert?
3. **Typsicherheit**
   Vermeide die Verwendung von Raw Types, benutze bevorzugt Generics.
4. **Magic Numbers**
   Sind Konstanten anstelle von nichtssagenden Magic Numbers verwendet? Sind zusammengehörende Konstanten nicht einzeln, sondern besser als `enum` definiert?
5. **Struktur**
   Ist der Sourcecode optisch klar strukturiert und nachvollziehbar?
6. **Kommentare**
   Sind Kommentare verständlich, ausreichend und erforderlich? Veraltete und überflüssige Kommentare sollten entfernt werden!
7. **Redundanzen**
   Ist kein Sourcecode-Fragment doppelt, überflüssig oder unbenutzt?
8. **Korrektheit und Tests**
   Arbeitet das Programm bereits augenscheinlich so wie gewünscht? Gibt es Unit Tests? Sind diese sinnvoll und erzielen eine hohe Testabdeckung (z. B. über 70 %)?
9. **Seiteneffekte**
   Existieren Seiteneffekte? Entferne diese wenn möglich, ansonsten dokumentiere sie ausreichend!
10. **Interface-Design**
    Ist das Interface einer Klasse nach außen möglichst klein? Andere Methoden sollten bevorzugt `private` oder `protected` definiert sein.
11. **Fehlerbehandlung**
    Werden Rückgabewerte (korrekt) ausgewertet und Fehlersituationen behandelt? Werden Exceptions, sofern sinnvoll möglich, behandelt?
12. **Casting**
    Werden Casts nur in Ausnahmefällen verwendet? Gibt es stattdessen eine objektorientierte Lösungsmöglichkeit mit gemeinsamer Basisklasse?
13. **Achtung »==« != »`equals()`«**
    Werden `==` und `equals()` korrekt eingesetzt?

22 Optimierungen

In diesem Kapitel werden Strategien und Techniken zum Optimieren von Programmen vorgestellt. Unter Optimierung wird meistens verstanden, dass sich die Ausführungsgeschwindigkeit eines Programms verbessert oder sich dessen Speicherbedarf reduziert.

Abschnitt 22.1 gibt einen Überblick über den Themenkomplex der Performance-Analyse. Unter anderem werden verschiedene, die Performance beeinflussende Faktoren sowie Techniken und Tools vorgestellt, die dabei helfen, mögliche Ursachen erkennen und kritische Stellen finden zu können. In Abschnitt 22.2 werden wir den Einfluss des Einsatzes unterschiedlicher Datenstrukturen auf die Laufzeit und den Speicherverbrauch diskutieren. Neben den gewählten Algorithmen und Datenstrukturen kann der zur Laufzeit benötigte Speicher die Ausführungsgeschwindigkeit beeinflussen – vor allem wenn einer JVM wenig Speicher zur Verfügung steht und es dadurch vermehrt zu Garbage Collections kommt. Als eine mögliche Abhilfe wird in Abschnitt 22.3 die Technik Lazy Initialization zur Vermeidung einer speicher- und zeitaufwendigen Konstruktion komplexer Objekte vorgestellt.

Negative Auswirkungen auf die Programmlaufzeit können durch unterschiedliche Abhilfemaßnahmen gemildert oder behoben werden. Dabei hängt der Erfolg einer Optimierung entscheidend von der problemangepassten Ebene ab: Bevor wir diese detailliert betrachten, stellt Abschnitt 22.4 konkret an einem Beispiel den Ablauf und verschiedene Techniken beim Optimieren eines Programmteils vor. Dadurch lassen sich die in den nachfolgenden Abschnitten beschriebenen Techniken leichter verstehen und einordnen. Einige Varianten zur Verbesserung der Ein- und Ausgabeverarbeitung (I/O) werden in Abschnitt 22.5 besprochen. Abschnitt 22.6 diskutiert Auswirkungen des zur Verfügung stehenden Speichers und stellt diesbezüglich adäquate Optimierungstechniken vor. Abschließend betrachtet Abschnitt 22.7 Techniken, die zu einer schnelleren Ausführung beitragen, indem CPU-intensive Berechnungen optimiert werden.

Generell sollten Optimierungen nicht allzu häufig notwendig sein. Wenn allerdings ein Performance-Problem vorliegt, so ist dessen Beseitigung elementar wichtig, beispielsweise um eine flüssige Abarbeitung in einem GUI oder einen guten Datendurchsatz in einer komplexen Berechnung zu erzielen. Manche Performance-Probleme sind während der Entwicklung jedoch nicht oder nur schwierig erkennbar, da sie sich erst beim Kunden durch ein viel höheres Datenvolumen oder andere Hardwarevoraussetzungen als bei entwicklungsbegleitenden Tests manifestieren. Empfehlenswert ist daher, die Zielumgebung, vor allem deren Beschränkungen und Datenraten, zu bedenken und gegebenenfalls zu simulieren.

22.1 Grundlagen

Optimierungen sind auf verschiedenen Ebenen möglich. Dabei gilt folgende Faustregel: ***Je höher die Abstraktionsebene ist, auf der wir Änderungen vornehmen, desto größer sind die erzielbaren Performance-Gewinne***. Die Wahl geeigneter Kommunikationsformen mit externen Systemen sowie der Einsatz passender Algorithmen und Datenstrukturen für ein zu lösendes Problem bewirken häufig viel mehr als feingranulare Änderungen auf der Ebene des Sourcecodes. Letztere machen ein Programm sogar teilweise lediglich schlechter lesbar und damit auch schwieriger wartbar. Sinnvoll ist es daher, zunächst immer nahe an der zu realisierenden Problemstellung und möglichst verständlich zu programmieren, bevor man bei Bedarf mit Optimierungen beginnt. Befolgt man diese Vorgehensweise nicht, werden dadurch in der Regel das Design und der Sourcecode komplizierter. Dies erschwert spätere Refactorings. Außerdem leiden Testbarkeit und Verständlichkeit. ***Deshalb sollte man immer im Hinterkopf behalten, dass mögliche Einbußen an Lesbarkeit und Erweiterbarkeit (für die Zukunft) schwerwiegender sein können als kurzfristig erzielbare Performance-Gewinne.*** Dieses Kapitel stellt daher bevorzugt solche Techniken vor, die zwar gute Performance-Gewinne ermöglichen, jedoch die Lesbarkeit und Verständlichkeit erhalten oder nur unwesentlich verringern. Abstriche sollte man nur dann in Kauf nehmen, wenn tatsächlich schwerwiegende Performance-Probleme durch Messungen nachgewiesen wurden.

Donald E. Knuth schrieb schon 1974 (meistens reduziert auf das zur Hervorhebung in Fettschrift angegebene Zitat): »Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. **We should forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil**« [46]. Frei übersetzt: »Programmierer verschwenden enorme Zeit damit, über die Laufzeit unkritischer Bestandteile ihrer Programme nachzudenken oder sich darüber zu sorgen. Diese Versuche der Performance-Steigerung wirken sich jedoch sehr negativ auf das Debugging und die Wartbarkeit aus. **In 97% aller Fälle sollten wir alle kleineren Verbesserungen nicht betrachten: Vorzeitige Optimierung ist die Quelle allen Übels**.«

Liegt tatsächlich ein Performance-Problem vor, so sollte man zunächst umfangreiche Analysen und Messungen vornehmen, um die verursachenden Stellen zu identifizieren. Ändert man den Sourcecode stattdessen vorschnell oder unüberlegt an diversen Stellen, die lediglich auf Vermutungen beruhen, kann dies zu weiteren Problemen führen. Der Grund dafür ist so einleuchtend wie einfach: Wenn wir mit unseren Vermutungen falsch liegen, so vergeuden wir wertvolle Zeit damit, irrelevante Programmteile zu modifizieren und zu optimieren. Wahrscheinlich machen wir die Programmteile zusätzlich noch unleserlich und bauen vielleicht sogar ungewollt Fehler ein, ohne einen Vorteil bezüglich der Performance zu erreichen. ***Optimierungen sind daher mit Vorsicht zu genießen, und keinesfalls sollte man nur basierend auf Vermutungen optimieren***. Jon Bentley betont: »Das wichtigste Prinzip beim Code-Tuning ist: Tun Sie es selten« [3].

Tipp: Einflüsse von Nebenläufigkeit und Parallelverarbeitung

Durch die Integration mehrerer Prozessorkerne in CPUs kommt heutzutage eine weitere Dimension ins Spiel: Das Aufteilen von Programmen auf mehrere Threads (oder gar Prozesse), ohne dass sich diese gegenseitig zu sehr durch Synchronisation ausbremsen. Wenn massive Parallelität und Nebenläufigkeit benötigt werden, kann dieser Effekt durch den Einsatz der Concurrent Collections (vgl. Abschnitt 11.1) gemildert werden. Zudem erlaubt das Fork-Join-Framework die Parallelisierung von Aufgaben (vgl. Abschnitt 11.3). Mit Java 8 sind diverse Erweiterungen zur Parallelverarbeitung in das JDK integriert worden. Ein paar davon werden in Abschnitt 11.4 vorgestellt. Java 9 bringt mit Ergänzungen in der Klasse `CompletableFuture` und der Unterstützung von Reactive Streams wichtige Erweiterungen, um Concurrency auf einer höheren Stufe der Abstraktion beschreiben zu können. Beides wird im Kapitel 11 thematisiert.

22.1.1 Optimierungsebenen und Einflussfaktoren

Wie bereits eingangs erwähnt, existieren verschiedene Einflussfaktoren, die sich auf die Performance auswirken können. Diese schauen wir uns nun an.

Hängt die Ausführungsgeschwindigkeit maßgeblich von den zu berechnenden Aufgaben ab, dann bezeichnet man ein Programm als ***CPU-bound***. Mit anderen Worten bedeutet dies: Verwendet man einen schnelleren Prozessor (CPU = Central Processing Unit), so steigt die Ausführungsgeschwindigkeit proportional zur schnelleren CPU. Für ***Memory-bound***-Programme gilt, dass Zugriffe auf den Hauptspeicher den kritischen Pfad darstellen und starke Auswirkungen auf die Ausführungsgeschwindigkeit haben. Zwei Faktoren spielen dabei eine Rolle: Einerseits wirkt sich ein zu klein dimensionierter Speicher negativ auf die Ausführungsgeschwindigkeit aus. Andererseits können auch häufige, eventuell vermeidbare Speicherzugriffe die Performance verschlechtern. Im einfachsten Fall kann man einer JVM mehr Speicher zur Verfügung stellen, um das Problem zu adressieren. Unter ***I/O-bound*** versteht man, dass die Ausführungsgeschwindigkeit eines Programms maßgeblich durch Zugriffe auf externe Systeme (Festplatte, Datenbank usw.) bestimmt wird. Beispielsweise können manchmal enorme Geschwindigkeitsgewinne erreicht werden, indem man Daten gepuffert verarbeitet.

Die zuvor vorgestellte Klassifizierung ergibt sich aus den unterschiedlichen Zugriffszeiten auf die entsprechenden Komponenten: Der Zugriff auf Register eines Prozessors ist schneller als der Zugriff auf nachgelagerte Prozessor-Caches. Beide sind jedoch extrem schnell. Hauptspeicherzugriffe sind im Gegensatz dazu bereits etwas langsamer. Nochmals um Größenordnungen langsamer sind Datei- und Netzwerkzugriffe.

Waren früher noch Berechnungen durch die CPU und später dann Hauptspeicherzugriffe ein Flaschenhals bezüglich der Performance, so gilt dies mittlerweile nur noch selten: Performance-Probleme sind inzwischen vielfach eine Kombination aus Memory-bound und I/O-bound. Das liegt daran, dass CPU und Speicher heutzutage extrem schnell sind und außerdem Programme oftmals mit anderen Komponenten, et-

wa einer Datenbank, oder mit anderen Programmen über Remote Calls interagieren. Ziel von Optimierungen muss es daher sein, zunächst die Kommunikation mit externen Systemen genauer zu betrachten, weil dort in der Regel das größte Optimierungspotenzial steckt. Nachdem auf dieser Ebene alle Möglichkeiten ausgeschöpft sind, kann man versuchen, Speicherzugriffe sowie eingesetzte Algorithmen und Datenstrukturen zu optimieren. Abgesehen von der Optimierung der genutzten Algorithmen sollte man von CPU-bound-Optimierungen auf der Ebene einzelner Anweisungen möglichst Abstand nehmen, weil diese meistens zwei Nachteile mit sich bringen: Zum einen muss man schon Experte sein und genau wissen, was man tut, um überhaupt Vorteile erzielen zu können und das Ganze besser zu machen als der Hotspot-Optimierer der JVM. Dieser optimiert unter anderem Speicherzugriffe und ordnet Anweisungen um oder fasst diese zusammen. Zum anderen ist derart per Hand optimierter Sourcecode oftmals unleserlich – nach einiger Zeit auch für einen selbst!

22.1.2 Optimierungstechniken

Obwohl die genannten Ebenen sehr unterschiedlich sind, basieren die Strategien und Techniken zur Optimierung immer wieder auf ähnlichen Ideen:

- **Wahl passender Strategien** – Die Wahl passender Strategien zum Zugriff oder zur Berechnung kann massive Auswirkungen auf die Performance haben. Erfolgt ein Zugriff auf Dateien beispielsweise Byte für Byte, ist das der Aufgabe nicht angemessen, da es nicht an die Funktionsweise der Hardware angepasst ist. Festplatten sind darauf optimiert, größere Datenmengen an einem Stück zu verarbeiten. Eine Pufferung sowie ein blockweises Lesen und Schreiben kann gegenüber mehrfachen Einzelzugriffen daher eine enorme Geschwindigkeitssteigerung bewirken, ohne jedoch konzeptionelle Änderungen zu bedingen. Ähnliche positive Effekte beobachtet man beim Zugriff auf Datenstrukturen: Ein indizierter Zugriff auf ein Array oder eine `ArrayList<E>` ist bei sehr umfangreichen Datenmengen deutlich schneller als bei einer `LinkedList<E>`. Der Einfluss verschiedener Datenstrukturen wird ausführlich in Abschnitt 22.2 besprochen. Ein weiteres Beispiel ist das Zusammenfassen von Aktionen: Eine zu erledigende Arbeit wird nicht in vielen kleinen, wiederholten Schritten, sondern gebündelt als eine größere Aktion ausgeführt.
- **Caching und Pooling** – Das Zwischenspeichern von Ergebnissen und das Wiederverwenden bereits erzeugter Objekte oder berechneter Ergebnisse fällt in die Kategorie von Caching und Pooling. Wie bereits erwähnt, gibt es eine Hierarchie bezüglich der Zugriffszeiten auf verschiedene Datenspeicher oder Medien. Caches können auf unterschiedlichen Ebenen eingesetzt werden. Ziel dabei ist es, Zugriffe auf das jeweils teurere Medium oder wiederholte aufwendige Berechnungen zu reduzieren, indem ein möglichst passender Satz an benötigten Daten im Cache gehalten wird. Dabei macht man sich die sogenannte ***Lokalität*** zunutze. Darunter versteht man, dass man gewisse ***Muster beim Datenzugriff*** findet: Oftmals werden

Daten während einer Verarbeitung innerhalb kürzerer Zeit mehrfach gelesen und zudem auch benachbarte Daten (z. B. Daten auf der Festplatte, Artikel einer Einkaufsliste oder ähnliche Top-Seller). Aufgrund dieser Beobachtungen lohnen sich Caches häufig in tieferen Schichten, beispielsweise bei Zugriffen auf Datenbanken oder das Dateisystem. Innerhalb einer Applikation profitiert man von Caches jedoch nur, wenn die Zugriffszeit auf Daten einen signifikanten Einfluss auf die Performance besitzt.

- **Vermeidung unnötiger Aktionen** – Je weniger unnütze Arbeit verrichtet werden muss, desto besser. Dies gilt beispielsweise für Zugriffe auf das Dateisystem, etwa bei Log-Ausgaben. Weiterhin versteht man darunter Strategien zur Vermeidung der Konstruktion optionaler Programmteile oder im Speziellen von Teilkomponenten schwergewichtiger Objekte. Eine nützliche Technik ist die Lazy Initialization, die in Abschnitt 22.3 genauer betrachtet wird.

Tabelle 22-1 zeigt Beispiele für die beschriebenen Techniken auf den drei Optimierungsebenen.

Tabelle 22-1 *Optimierungstechniken und Beispiele*

	Ebene		
Technik	**I/O**	**Memory**	**CPU**
Wahl passender Strategien	Gepufferte Zugriffe, Anpassungen der Serialisierung	Algorithmen und Datenstrukturen	Algorithmen und Datenstrukturen, Peephole-Optimierungen
Caching und Pooling	Daten-Caches, Data Transfer Objects	Daten-Caches, Objekt-Pools	Vorausberechnung und Zwischenspeicherung von Werten
Vermeidung unnötiger Aktionen	Log-Level-Prüfung, Lazy Initialization	Lazy Initialization	Vermeidung von Auto-Boxing/-Unboxing

Die zuvor erwähnten Beispiele machen deutlich, dass die Abgrenzung der Techniken nicht immer messerscharf ist: Beispielsweise dient ein Cache bzw. ein Objekt-Pool dazu, unnötige Aktionen (Speicherzugriffe) zu vermeiden, und ist damit thematisch nahe der Technik »Vermeidung unnötiger Aktionen« angesiedelt. Zudem erkennen wir, dass die Technik Lazy Initialization sowie die Wahl passender Algorithmen und Datenstrukturen[1] auf unterschiedlichen Ebenen angewendet werden können.

[1] In diesem Buch betrachte ich eher Datenstrukturen, da es für die Wahl passender Algorithmen diverse Literatur gibt. Bei Bedarf gebe ich Tipps, um Algorithmusprobleme zu lösen.

22.1.3 CPU-bound-Optimierungsebenen am Beispiel

Performance-Optimierung wird von vielen häufig immer noch als eine Verbesserung auf der Ebene einfacher Anweisungen im Sourcecode und maximal des eingesetzten Algorithmus betrachtet. Zuvor habe ich bereits angedeutet, dass diese Art der Optimierung heutzutage nur in seltenen Fällen eine entscheidende Verbesserung bezüglich der Ausführungsgeschwindigkeit eines Programms bewirken wird. Um dies etwas zu präzisieren und ein Gefühl für mögliche Umsetzungen und Auswirkungen zu vermitteln, möchte ich im Folgenden einige Optimierungsmöglichkeiten auf den Ebenen Design, Sourcecode und Bytecode bzw. Kompilierung beschreiben.

- **Design** – Auf der Ebene des Designs gilt es, die eingesetzten Algorithmen und Datenstrukturen optimal auf die Anforderungen anzupassen. Ein sauber strukturiertes Design kann dabei helfen, spätere Korrekturen leichter vornehmen zu können, Manchmal reicht es bereits, einen Teil eines Algorithmus in eine Klasse oder Methode auszulagern und dadurch austauschbar zu machen (vgl. Abschnitt 18.3.4 zum STRATEGIE-Muster). Betrachten wir dies für folgendes Interface zur Summenberechnung und eine naive Realisierung der dort deklarierten `sumOfN(int)`-Methode, die die Summation der Zahlen von 0 bis n mit einer Schleife löst:

```
public interface SumCalculator
{
    public long sumOfN(final int n);
}

public class NaiveSum implements SumCalculator
{
    public long sumOfN(final int n)
    {
        long sum = 0;
        for (int i = 0; i <= n; i++)
        {
            sum += i;
        }
        return sum;
    }
}
```

Weil hier gegen das Interface `SumCalculator` programmiert wird, kann die derzeit genutzte Implementierung bei Performance-Problemen im Nachhinein ausgetauscht werden, ohne Änderungen in aufrufenden Klienten erforderlich zu machen. Man kann beispielsweise eine Klasse `OptimizedSum` wie folgt implementieren:

```
public class OptimizedSum implements SumCalculator
{
    public long sumOfN(final int n)
    {
        return n * (n + 1) / 2;
    }
}
```

Bei dieser Realisierung der Methode `sumOfN(int)` nutzt man, dass folgende mathematische Gleichung gilt:

$$\sum_{i=0}^{n} i = \frac{n * (n+1)}{2}$$

Ähnliche Ideen haben Steve Wilson und Jeff Kesselman in ihrem Buch »Java Platform Performance« [73] entwickelt, allerdings ohne Bezug zum STRATEGIE-Muster, dafür aber mit einer trickreicheren Summierungsmethode.

- **Sourcecode** – In seltenen Fällen können Optimierungen auf Sourcecode-Ebene durch Nutzung anderer Anweisungen die Ausführungsgeschwindigkeit wirklich erhöhen. Für viele Konstrukte gilt dies jedoch nicht oder nur extrem eingeschränkt. Betrachten wir dies für die (vermeintliche) Optimierung, eine Multiplikation durch einen Bit-Shift zu realisieren. Immer wieder mal liest man, dass Bit-Shifts für die Multiplikation deutlich performanter sein sollen. Ein Bit-Shift um eine Position nach links entspricht einer Multiplikation mit dem Faktor zwei. Im folgenden Beispiel wird eine Multiplikation mit der Zahl sechs durch zwei Bit-Shifts (einmal um 1 Bit und einmal um 2 Bit gefolgt von einer Addition) realisiert:

```
final long ONE_BILLION = 1_000_000_000L;
for (long i = 0; i < ONE_BILLION; i++)
{
   value = (i << 2) + (i << 1); // Multiplikation durch Bit-Shifts
}
```

Die derart »optimierte« Variante benötigt bei einer Milliarde (!) Durchläufen auf meinem iMac (Quad-Core Intel i7 mit 4 GHz) mit JDK 14.0.1 lediglich ca. 635 Millisekunden. Allerdings ist sie damit sogar minimal langsamer als die herkömmliche und gut lesbare Multiplikation, die rund 630 Millisekunden benötigt – vergleichen Sie es mit den Ausführungszeiten auf Ihrem Rechner durch Start des Programms PEEPHOLEOPTIMIZATIONEXAMPLE:

```
for (long i = 0; i < ONE_BILLION; i++)
{
   value = i * 6;
}
```

In älterer Literatur und auch von einigen Entwicklern werden Optimierungstechniken auf Sourcecode-Ebene in ihrer Wirksamkeit auf die Gesamtperformance stark überbewertet bzw. überschätzt. Seit JDK 12 kann man JMH (vgl. Abschnitt 14.4) nutzen, um derartige Aussagen zu überprüfen – hier gekürzt gezeigt:

```
Benchmark                                  (multCount) Mode Score    Units
PeepholeOptimizationJMH.bitShiftMultiplication  10000000 avgt 7075.989 us/op
PeepholeOptimizationJMH.plainMultiplication     10000000 avgt 6999.056 us/op
```

Wie die ermittelten Zahlen zeigen, muss man diese Art der Optimierung genau betrachten: Erst bei extrem Performance-kritischen Applikationen und nach Aus-

schöpfen aller anderen Möglichkeiten ist in ganz seltenen Fällen möglicherweise ein Performance-Gewinn zu erzielen. Also: ***Vorsicht vor solchen Optimierungen!***

- **Bytecode bzw. Kompilierung** – Bei der Kompilierung des Java-Sourcecodes in korrespondierenden Bytecode werden ***vom Compiler automatisch*** verschiedene Optimierungen ausgeführt. Zur Laufzeit erfolgt eine weitere Optimierung durch den sogenannten Hotspot-Optimierer (vgl. folgenden Praxistipp »Arbeitsweise und Einfluss des Hotspot-Optimierers«), der unter anderem die in der folgenden Aufzählung beschriebenen Techniken Inlining, Loop-Unrolling und Peephole-Optimierung vornimmt. Diese Techniken tragen dazu bei, unnötige Berechnungen zu vermeiden und Programmanweisungen gut auf die ausführende Hardware abzustimmen, beispielsweise durch die Nutzung von Caches und Registern.

  - **Inlining** – Kurze Methoden können vom Compiler direkt in den Methodenrumpf des Aufrufers eingefügt werden, statt angesprungen zu werden:

```
private static int processValues(final int x, final int y)
{
    return multiply(x, y);
}

private static int multiply(final int x, final int y)
{
    return x * y;
}
```

    Mit Inlining wird daraus Bytecode, der folgendem Sourcecode der Methode `processValuesInlined(int, int)` entspricht – hier ist die Integration des Methodenrumpfs der Methode `multiply(int, int)` gezeigt:

```
// Integration der Methode multiply()
private static int processValuesInlined(final int x, final int y)
{
    return x * y;
}
```

  - **Loop-Unrolling** – Für kurze Schleifen werden die Anweisungen des Schleifenrumpfs mehrfach hintereinander kopiert. Betrachten wir folgende Schleife:

```
private static int conventionalLoop()
{
    int x = 0;
    for (int i = 1; i < 4; i++)
    {
        x += i * i;
    }
    return x;
}
```

    Durch den Compiler optimiert wird daraus eine Folge von Anweisungen, die als Terme nur noch Konstanten enthalten. Zur Verdeutlichung der Technik sind hier die Berechnungen noch einzeln aufgeführt. In der im Anschluss beschriebenen Peephole-Optimierung werden diese weiter zusammengefasst.

```
private static int unrolledLoop()
{
    int x = 0;
    x += 1 * 1;
    x += 2 * 2;
    x += 3 * 3;
    return x;
}
```

- **Peephole-Optimierung** – Es werden beispielsweise Speicher- und Registerzugriffe optimiert und Berechnungen zur Kompilierzeit durchgeführt. Im nachfolgenden Beispiel wird eine Variable `x` mit dem Wert `0` initialisiert und danach eine Berechnung ausgeführt:

```
private static void assignment()
{
    int x = 0;
    x = 4 * 25;
    // ...
```

Zur Kompilierzeit wird das Ergebnis der Multiplikation der Zahlenliterale berechnet und dieser Wert, hier `100`, direkt an die Variable zugewiesen:

```
private static void assignmentOptimized()
{
    int x = 100; // Peephole: Zuweisung und Multiplikation
    // ...
```

Tipp: Arbeitsweise und Einfluss des Hotspot-Optimierers

Beim Start eines Java-Programms wird sämtlicher Bytecode interpretiert, d. h., Anweisung für Anweisung wird von einem Interpreter ausgeführt. Besonders häufig benutzte Programmteile, sogenannte »Hotspots«, werden von einem in die JVM integrierten Optimierer erkannt und kompiliert. Dieser sogenannte Hotspot-Optimierer erzeugt den Maschinencode zur Laufzeit parallel zur Abarbeitung des eigentlichen Programms. Man spricht daher von einem ***Just-in-Time-Compiler*** (JIT). Dieser Übersetzungsvorgang wirkt sich geringfügig verzögernd auf die eigentliche Programmlaufzeit aus. Die Auswirkungen sind jedoch minimal, da die Kompilierung nur für einige ausgewählte Methoden ausgeführt wird, die zuvor erkannten Hotspots.

Die JVM konnte bis Java 10 in zwei Optimierungsmodi laufen. Im Server-Modus (`-server`) werden die in der obigen Aufzählung genannten Optimierungen besonders stark durchgeführt. Die Kompilierung ist dadurch etwas langsamer, aber besser optimiert als im Client-Modus (`-client`), der weniger optimiert, um schnell zu kompilieren, und somit weniger Einfluss auf die Anwendungslaufzeit hat. Seit Java 11 gibt es nur noch den Server-Modus.

Diskussion

Je höher die Abstraktionsebene ist, auf der wir Optimierungen vornehmen, desto größer sind in der Regel die zu erzielenden Performance-Gewinne. Daher bietet es sich an, zunächst die Architektur und das Design zu verbessern: Manchmal gibt es für ein zu lösendes Problem geeignetere Algorithmen oder Datenstrukturen als die momentan gewählten. Beginnt man mit Optimierungen auf Designebene, so lassen sich als Folge oftmals die zuvor vorgestellten Optimierungen auf Sourcecode-Ebene vermeiden. Nur wenn auf der Ebene des Designs tatsächlich keine weiteren Verbesserungen mehr zu erzielen sind und weiterhin Optimierungsbedarf besteht, kann man Optimierungen auf Sourcecode-Ebene in Betracht ziehen. Somit wird nur in wirklich notwendigen Fällen die Lesbarkeit des Sourcecodes durch Optimierungen beeinträchtigt. Nichtsdestotrotz sieht man leider gerade auf der Ebene des Sourcecodes immer wieder vorschnell durchgeführte Optimierungen, die Lesbarkeit gegen vermutete Performance-Verbesserungen eintauschen, indem sie die Transformationen des Compilers auf Bytecode-Ebene in den Sourcecode aufnehmen. Dies sollte man vermeiden, da die Hotspot-Optimierung sehr effizient erfolgt und zudem automatisch geschieht – ohne Einfluss auf die Lesbarkeit.

Vor Beginn von Optimierungen ist es wichtig, dass der Programmteil korrekt funktioniert und ausreichend getestet ist. Zudem bietet sich eine Absicherung durch Unit Tests (vgl. Kapitel 20) an, um nachfolgende Änderungen auf Korrektheit und Kompatibilität zur bisherigen Umsetzung prüfen zu können. Außerdem lassen sich durch ein solches Vorgehen die Schritte Optimierung und Programmentwicklung sauber voneinander trennen. Änderungen sollten dann entweder zur Realisierung der Funktionalität oder zur Optimierung erfolgen. Man vermeidet so, beide Tätigkeiten miteinander zu vermischen und am Ende weder einen korrekten Algorithmus noch eine sinnvolle Optimierung realisiert zu haben.

22.1.4 Messungen – Erkennen kritischer Bereiche

Es ist es nahezu unmöglich, Performance-kritische Bereiche lediglich durch visuelle Sourcecode-Analyse zu erkennen. Dies gilt umso mehr, je komplizierter und umfangreicher Applikationen werden, etwa bei verteilten Applikationen oder beim Einsatz von Applikationsservern.

Für kleinere Applikationen lassen sich im einfachsten Fall über sogenannte Stoppuhr- bzw. Vorher-nachher-Messungen bereits Aussagen treffen. Dies setzt allerdings voraus, dass man bereits eine ungefähre Ahnung hat, welche Abschnitte des Sourcecodes Performance-Probleme verursachen könnten. In der Regel gilt: ***Alles, was nicht innerhalb von Schleifen oder anderweitig sehr häufig ausgeführt wird, ist höchstwahrscheinlich für die Performance unkritisch.***

Warum es elementar wichtig ist, nicht auf Verdacht, sondern nur durch Messungen begründet zu optimieren, wird durch die im Anschluss vorgestellte »80-zu-20-Regel« motiviert.

Die »80-zu-20-Regel«

Die sogenannte »80-zu-20-Regel« (auch Pareto-Prinzip genannt) lässt sich interessanterweise auf viele Situationen anwenden und wird in den unterschiedlichsten Bereichen der Softwareentwicklung verwendet. Eine Auslegung der Regel besagt, dass man die ersten 80 % des Sourcecodes in 20 % der Projektzeit schreibt. Für die restlichen 20 % des Sourcecodes benötigt man allerdings die verbleibenden 80 % der Zeit[2]. ***Konkret bedeutet dies, dass schnell prototypische Umsetzungen mit einem relativ großen Funktionsumfang entstehen können, dass es aber bis zu einem fertigen Produkt noch ein sehr langer Weg ist***.

An dieser Stelle ist jedoch ein weiteres Auftreten der Regel noch interessanter: Für die Ausführungszeit gilt häufig, dass 80 % des Sourcecodes nur 20 % der Rechenleistung benötigen und dass 20 % für 80 % Last verantwortlich sind. Im Normalfall kann man also davon ausgehen, dass nur einige wenige kritische Performance-Bremsen im Programm existieren. Die meisten davon lassen sich entsprechend der erstgenannten Auslegung der 80-zu-20-Regel schnell beheben: Man erzielt 80 % Verbesserung mit 20 % Zeiteinsatz. Oftmals sind diese ersten 80 % glücklicherweise bereits ausreichend, um eine ansprechende Performance zu erzielen. Eine umfassendere Optimierung, um das letzte Quäntchen Performance (die letzten 20 %) herauszukitzeln, ist dann in der Regel aber mühselig und unangemessen zeitaufwendig. Die wesentliche Erkenntnis ist jedoch: ***Optimiert man, ohne vorher zu messen und kritische Stellen erkannt zu haben, so liegt die Wahrscheinlichkeit bei ca. 80 %, dass man eine Stelle bearbeitet, die keinen signifikanten Einfluss auf die Performance besitzt. Man kann also nahezu endlos an unwichtigen Stellen feilen – ohne jedoch nennenswerte Verbesserungen zu erzielen***.

Messungen des Zeitverhaltens

Für Performance-Abschätzungen werden in den folgenden Abschnitten immer wieder Stoppuhr-Messungen durchgeführt. Man kann dazu die Methode `System.currentTimeMillis()` nutzen. Für exaktere Messungen und Analysen ist diese Methode aufgrund ihrer Ungenauigkeit[3] jedoch nur bedingt geeignet. Wird eine größere Genauigkeit benötigt, so sollte für diese Fälle besser die Methode `System.nanoTime()` genutzt werden, die immer den genauesten verfügbaren Zeitgeber des Betriebssystems wählt.

Zur Unterstützung unserer Zeitmessungen entwickeln wir eine Utility-Klasse `PerformanceUtils`, die die Methode `System.nanoTime()` einsetzt. Eine innere Klasse `TimingEntry` speichert einen Namen und gemessene Zeiten. Die Zeiten werden über die Hilfsmethoden `startMeasure(String)` und `stopMeasure(String)` der äußeren Klasse ermittelt.

---

[2] Die tatsächlichen Zahlen sind nicht fix, sondern schwanken etwas: 90 zu 10 oder 70 zu 30 passen auch sehr häufig.

[3] Die Auflösung der Zeitmessung liegt in Windows-Systemen nur etwa im Bereich von 10 bis 20 ms. Ein Zahlenwert von 0 ms ist damit durchaus mit einer Angabe von 15 ms vergleichbar.

Das folgende Listing zeigt die Realisierung der Utility-Klasse `PerformanceUtils`, die die genannten Zeitmessungen ermöglicht:

```
public final class PerformanceUtils
{
    private static final Logger log =
                                LogManager.getLogger(PerformanceUtils.class);

    private static final Map<String, TimingEntry> timingMap = new HashMap<>();

    public static void startMeasure(final String name)
    {
        timingMap.put(name, new TimingEntry(System.nanoTime()));
    }

    public static void stopMeasure(final String name)
    {
        final TimingEntry timingEntry = getTimingEntry(name);
        timingEntry.setStopTime(System.nanoTime());
    }

    public static void printTimingResult(final String name)
    {
        final TimingEntry timingEntry = getTimingEntry(name);
        printTimingResult(name, timingEntry.startTime, timingEntry.stopTime);
    }

    public static void printTimingResult(final String info,
                                         final long begin, final long end)
    {
        log.info(info + " took " + TimeUnit.NANOSECONDS.toMillis(end - begin) +
                 " ms");
    }

    public static void printTimingResultWithAverage(final String name,
                                                    final long count)
    {
        printTimingResult(name);

        final TimingEntry timingEntry = getTimingEntry(name);
        final double avg = TimeUnit.NANOSECONDS.toMillis(timingEntry.stopTime -
                           timingEntry.startTime) / (double) count;
        log.info(String.format(info + " avg %f ms", avg));
    }

    private static TimingEntry getTimingEntry(final String name)
    {
        final TimingEntry timingEntry = timingMap.get(name);
        if (timingEntry==null)
            throw new IllegalArgumentException("No data for '" + name + "'");

        return timingEntry;
    }

    private static final class TimingEntry
    {
        private final long   startTime;
        private long         stopTime;

        public TimingEntry(final long startTime)
        {
            this.startTime = startTime;
        }
```

```
        public void setStopTime(final long stopTime)
        {
            this.stopTime = stopTime;
        }

        @Override
        public int hashCode()
        {
           return Objects.hash(this.startTime);
        }

        @Override
        public boolean equals(final Object obj)
        {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;

            final TimingEntry other = (TimingEntry) obj;
            return (startTime == other.startTime && stopTime == other.stopTime);
        }
    }
}
```

Bei der Ausgabe der Messergebnisse erfolgt eine Umrechnung auf eine Genauigkeit in Millisekunden, da dies in der Regel für eine Einschätzung der Auswirkungen verschiedener Designentscheidungen ausreichend ist.

Mit einfachen Vorher-nachher-Messungen lassen sich nur relativ triviale Abläufe untersuchen. Müssen komplexere Aktionen analysiert werden oder ist das Verhalten über mehrere Durchläufe oder Stunden interessant, so helfen Profiling-Tools, kritische Stellen aufzuspüren. Die Stoppuhr-Messung hat trotzdem ihre Daseinsberechtigung, weil sie, im Gegensatz zu einer Messung mit einem Profiler, keinen Einfluss auf die Ausführungsdauer des gemessenen Programmabschnitts hat.

Messungen des Speicherbedarfs

Die obigen Aussagen zur Ausführungszeit gelten analog für den von einer Applikation verbrauchten Speicher, der ohne Profiling-Tools nur rudimentär mit den Methoden `freeMemory()` und `totalMemory()` der Klasse `java.lang.Runtime` ermittelt werden kann. Allerdings ist durch die automatisch, zu beliebigen Zeitpunkten stattfindende Speicherbereinigung durch Garbage Collections zur genaueren Messung ein vorheriger (gegebenenfalls mehrfacher) Aufruf von `System.gc()` ratsam. Jedoch erfolgt dadurch nicht zwangsläufig eine Garbage Collection. Aber in vielen Fällen wird diese durchgeführt, und es kommt zu einer verbesserten Genauigkeit der Speichermessung.

Der einer Applikation zur Verfügung stehende Hauptspeicher kann großen Einfluss auf die Performance haben. Um eine erste Einschätzung des Speicherbedarfs zu bekommen und einige Effekte zu analysieren, kann man sich vergleichbar zur obigen Klasse `PerformanceUtils` eine Klasse `MemoryInfo` schreiben, die erste, aber rudimentäre

Auswertungen erlaubt. Gerade bei der Untersuchung des Speicherverbrauchs besitzen Profiling-Tools besondere Vorzüge, da diese die Belegung des Speichers analysieren können und somit Angaben zu den genutzten Typen und der Anzahl ihrer Instanzen ermöglichen. Dadurch lässt sich z. B. die Anzahl von `Integer`-Instanzen ermitteln.

Professionelles Memory-Analyse-Tool Falls Sie ein Problem mit dem Speicherverbrauch haben, kann ich Ihnen das Eclipse Memory Analyzer Tool (MAT) wärmstens ans Herz legen, das unter `https://www.eclipse.org/mat/` frei verfügbar ist. Es hat mir schon ein paar Mal gute Dienste geleistet, nachdem ich weder mit VisualVM noch mit den im Anschluss beschriebenen Profiling-Tools den Speicherproblemen auf die Spur gekommen bin. Das MAT ermöglicht die Analyse von Heapdumps selbst von riesigen Datenbeständen. Insbesondere ist das Tool in der Lage, diese zu gruppieren und logischen Einheiten zuzuordnen, wodurch es leichter fällt, nachzuvollziehen, welches Objekt verhindert, dass die referenzierten Objekte freigegeben werden können.

Profiling-Tools

Um Performance-Probleme in eigenen Programmen aufdecken zu können, helfen Profiling-Tools. Teilweise sind die im JDK bereitgestellten Tools bereits ausreichend. Hilfreicher ist jedoch `VisualVM`, das ein GUI bietet, das den Einstieg in die Performance-Optimierung erleichtert. `VisualVM` wurde mit JDK 6 eingeführt und bündelt einige bisher einzeln vorhandene Performance-Auswertungsprogramme, wie `jmap` (Speicherübersicht), `jps` (Übersicht über alle laufenden JVMs eines Rechners), `jstack` (Info über alle Threads eines Programms mit Stacktrace) und `jstat` (Anzeige von Performance-Informationen), zu einer integrierten Performance-Messsuite, die zudem über verschiedene Plugins erweitert werden kann. Mittlerweile ist es als separater Download unter `https://visualvm.github.io/download.html` erhältlich. Mit `VisualVM` lässt sich das Verhalten von Anwendungen detailliert auswerten und grafisch visualisieren. In Abbildung 22-1 ist eine Profiling-Session gezeigt.

Das Tool `VisualGC` existiert als Plugin für `VisualVM`. Damit lassen sich Garbage-Collection-Vorgänge gut beobachten und auswerten. Dies ist in Abbildung 22-2 exemplarisch gezeigt. Man sieht auf einen Blick den Füllgrad der Speicherbereiche Eden, Survivor, Old und Perm sowie die »Wanderung« der Objekte durch diese Bereiche. Außerdem kann man die Häufigkeit und Dauer von Garbage Collections ablesen. Grundlagen zum Thema Garbage Collection sind in Abschnitt 12.4 beschrieben.

Abbildung 22-1 *VisualVM beim CPU-Profiling*

Abbildung 22-2 *VisualVM mit VisualGC-Plugin*

Kommerzielle Alternative Als Alternative zu den bereits recht hilfreichen, informativen und frei verfügbaren Tools existieren einige kommerzielle Programme, die noch ein wenig mehr Funktionalität bieten. Exemplarisch sei das Tool JProfiler empfohlen, das als Evaluationsversion unter `http://www.ej-technologies.com/` heruntergeladen werden kann.

Validierung von Änderungen

Nachdem die kritischen Stellen in einem untersuchten Programm gefunden, analysiert und modifiziert wurden, sollte man zum einen alle Tests ausführen und zum anderen die Profiling-Messungen wiederholen. Durch den Einsatz von Tests wird sichergestellt, dass durch die Optimierungen keine Fehler eingebaut wurden. Die Performance-Messungen werden erneut durchgeführt, um zu beurteilen und im Idealfall zu bestätigen, dass es tatsächlich zu den erwarteten Verbesserungen gekommen ist. Andernfalls verwerfen wir normalerweise die Änderungen. Nur in seltenen Fällen können Optimierungen für mehr Klarheit durch strukturelle Verbesserungen oder eine bessere Namensgebung gesorgt haben. Derartige Änderungen sollten natürlich beibehalten werden.

Skalierbarkeit

Es ist wichtig, Performance-Tests mit einem umfangreichen, praxisrelevanten Satz an Testdaten mit annähernd realen Datenvolumina nachzustellen, weil einige Performance-Probleme erst bei sehr großen Datenmengen auftreten. Man spricht von schlechter ***Skalierbarkeit***, wenn mit zunehmender Datenmenge die Ausführungsdauer signifikant, mitunter sogar überproportional, zunimmt. Einige Designentscheidungen können zu einer schlechten Skalierbarkeit führen. Dies betrifft vor allem die eingesetzten Algorithmen und Datenstrukturen. In der einführenden Diskussion haben wir ein Beispiel dafür mit der Summierung von Zahlen kennengelernt.

Auch Aspekte der Parallelisierung von Aufgaben spielen eine Rolle: Ob man voneinander unabhängige Anfragen (Aufrufe) sequenziell oder aber parallel durch mehrere Threads bearbeitet, kann sich massiv auf die Performance und das Antwortzeitverhalten auswirken. Bei einer großen Zahl solcher Anfragen kommt es dann bei sequenzieller Abarbeitung möglicherweise zu miserablen Antwortzeiten und bei großer Parallelität bestenfalls zu einem geringen Einfluss der Anzahl von Anfragen auf die Antwortzeit.

Beispiel Betrachten wir den Einfluss von Datenstrukturen exemplarisch an einer Software zur Kundenverwaltung. Beim Entwurf eines solchen Systems muss man sich für eine Datenstruktur zur Verwaltung der Kundendaten entscheiden. Geschieht dies in Form einer Liste, so ist zwar ein indizierter Zugriff schnell. In der Regel wird man auf Kunden aber nicht indiziert, sondern bevorzugt über deren Namen oder andere Auswahlkriterien zugreifen wollen. Eine lineare Suche über alle Datensätze ist für einige Hundert oder Tausend Einträge sicherlich kein Problem. Allerdings skaliert die Lösung schlecht. Je mehr Einträge vorhanden sind, desto länger dauert die Suche. Deren Ausführungszeit wächst in diesem Fall proportional zur Anzahl der gespeicherten Daten. Nutzt man eine `TreeMap<K,V>`, so liegen die gespeicherten Elemente sortiert vor, und eine Suche ist wesentlich performanter möglich. Die Ausführungsdauer wächst logarithmisch zum Datenvolumen.

Für eine formalere Betrachtung zur Einordnung der Komplexität wird im Folgenden die O-Notation vorgestellt.

22.1.5 Abschätzungen mit der O-Notation

Zur Abschätzung und Beschreibung der Komplexität von Algorithmen und damit zur Einordnung ihres Zeitverhaltens wäre es unpraktisch, immer Messungen vornehmen zu müssen. Außerdem spiegeln Messungen lediglich das Laufzeitverhalten unter gewissen Randbedingungen der Hardware (Prozessortakt, Speicher usw.) wider.

Um Folgen von Designentscheidungen unabhängig von solchen Details und auf einer abstrakteren Ebene einordnen zu können, verwendet man in der Informatik die sogenannte ***O-Notation***, die die obere Schranke für die Komplexität eines Algorithmus angibt. Man möchte folgende Frage beantworten können: »Wie verhält sich ein Programm, wenn statt 1.000 Eingabewerten beispielsweise 10.000 oder 100.000 Eingabewerte verarbeitet werden?« Zur Beantwortung dieser Frage müssen die einzelnen Schritte eines Algorithmus betrachtet und klassifiziert werden. Ziel ist es, die Berechnung der Komplexität zu formalisieren, um Auswirkungen von Veränderungen an der Anzahl der Eingabedaten auf die Programmlaufzeit abschätzen zu können.

Betrachten wir folgende `while`-Schleife als einführendes Beispiel:

```
int i = 0;                      // O(1)
while (i < n)                   // O(n)
{
    createPersonInDb(i);        // O(1)
    i++;                        // O(1)
}
```

Jede einzelne Anweisung wird mit der Komplexität $O(1)$ bewertet. Die Schleife selbst erhält aufgrund der n Ausführungen des Schleifenrumpfs die Komplexität $O(n)$ zugeordnet.[4] Rechnet man diese Werte zusammen, dann sind die Kosten für die Abarbeitung des Programms folglich: $O(1) + O(n) * (O(1) + O(1)) = O(1) + O(n) * 2$. Für eine Abschätzung der Komplexität spielen konstante Summanden und Faktoren keine Rolle. Nur die höchste Potenz von n ist von Interesse. Somit kommt man für das abgebildete Programmstück auf eine Komplexität von $O(n)$. Diese Vereinfachung ist zulässig, da für größere Werte von n der Einfluss von Faktoren und kleineren Komplexitätsklassen unbedeutend ist. Für das Verständnis der Betrachtungen in den folgenden Abschnitten sollte diese informelle Definition ausreichend sein.

Nachfolgend möchte ich noch zwei Sätze von Robert Sedgewick aus seinem Standardwerk »Algorithmen« [65] zitieren, die die O-Notation charakterisieren: »[...] ist die O-Schreibweise ein nützliches Hilfsmittel, um obere Schranken für die Laufzeit anzugeben, die von den Einzelheiten der Eingabedaten und der Implementation unabhängig sind.« Und weiter heißt es dort: »Die O-Schreibweise erweist sich als äußerst nützlich, indem sie Analytikern hilft, Algorithmen nach ihrer Leistungsfähigkeit zu klassifizieren, und indem sie Entwickler von Algorithmen bei der Suche nach den »besten« Algorithmen unterstützt.«

[4] Die Bedeutung der Notation wird auf der nächsten Seite mit der Vorstellung von Beispielen für weitere Komplexitätsklassen verständlicher. Eine weiterführende Darstellung finden Sie unter `http://www.linux-related.de/index.html?/coding/o-notation.htm`.

Komplexitätsklassen

Um das Laufzeitverhalten verschiedener Algorithmen miteinander vergleichen zu können, reichen in der Regel sieben unterschiedliche Komplexitätsklassen aus. Folgende Aufzählung nennt die jeweilige Komplexitätsklasse und einige Beispiele dazu:

- $O(1)$ – Die konstante Komplexität liefert eine von der Anzahl der Eingabedaten n unabhängige Laufzeit. Diese Komplexität repräsentiert häufig ***eine Anweisung*** oder eine einfache Berechnung, die aus einigen Berechnungsschritten besteht.
- $O(log(n))$ – Bei der logarithmischen Komplexität kommt es zu einer Verdopplung der Laufzeit, wenn die Eingabedatenmenge n quadriert wird. Ein bekanntes Beispiel für diese Komplexität ist die ***Binärsuche***.
- $O(n)$ – Bei der linearen Komplexität wächst die Laufzeit proportional zur Anzahl der Elemente n. Dies ist bei einfachen Schleifen und Iterationen der Fall, etwa bei einer sequenzielle ***Suche in einem Array*** oder einer Liste.
- $O(n \cdot log(n))$ – Diese Komplexität ist eine Kombination aus linearem und logarithmischem Wachstum. Einige der schnellsten ***Sortieralgorithmen*** (z. B. Mergesort) besitzen diese Komplexität.
- $O(n^2)$ – Die quadratische Komplexität führt bei einer Verdopplung der Menge der Eingabedaten n bereits zu einer Vervierfachung der Laufzeit. Bei einer Verzehnfachung der Eingabedaten kommt es schon zu einem Verhundertfachen der Laufzeit. In der Praxis findet man diese Komplexität bei ***zwei ineinander verschachtelten `for`- oder `while`-Schleifen***. Einfache Sortieralgorithmen haben normalerweise diese Komplexität.
- $O(n^3)$ – Bei der kubischen Komplexität kommt es bei einer Verdopplung von n bereits zu einer Verachtfachung der Laufzeit. Die naive ***Multiplikation von Matrizen*** ist ein Beispiel für diese Komplexitätsklasse.
- $O(2^n)$ – Die exponentielle Komplexität führt bei einer Verdopplung von n zu einer Quadrierung der Laufzeit. Das klingt zunächst wenig. Bei einer Verzehnfachung steigt die Laufzeit aber um den Faktor 20 Milliarden! Die exponentielle Komplexität tritt häufig bei ***Optimierungsproblemen*** auf, etwa dem sogenannten Problem des Handlungsreisenden (***Traveling-Salesman-Problem***), bei dem es darum geht, den kürzesten Weg zwischen verschiedenen Städten zu ermitteln und dabei alle Städte zu besuchen. Um dem Problem der exorbitanten Laufzeit Herr zu werden, verwendet man Heuristiken, die zwar nicht die optimale Lösung, sondern nur eine Annäherung daran ermitteln, dafür aber eine viel geringere Komplexität und deutlich kürzere Laufzeit besitzen.

Tabelle 22-2 zeigt eindrucksvoll, welche Auswirkungen die genannten Komplexitätsklassen für verschiedene Mengen von Eingabedaten n besitzen – bitte beachten Sie, dass die Zeitkomplexität $O(2^n)$ nicht dargestellt wird, weil deren Wachstum zu stark ist, um es sinnvoll ohne den Einsatz von 10er-Potenzen auszudrücken.

Tabelle 22-2 *Auswirkungen verschiedener Zeitkomplexitäten*

n	$O(log(n))$	$O(n)$	$O(n \cdot log(n))$	$O(n^2)$	$O(n^3)$
10	1	10	10	100	1.000
100	2	100	200	10.000	1.000.000
1.000	3	1.000	3.000	1.000.000	1.000.000.000
10.000	4	10.000	40.000	100.000.000	1.000.000.000.000
100.000	5	100.000	500.000	10.000.000.000	1.000.000.000.000.000
1.000.000	6	1.000.000	6.000.000	1.000.000.000.000	1.000.000.000.000.000.000

Anhand der dargestellten Werte bekommt man ein Gefühl für die Auswirkungen unterschiedlicher Komplexitäten: Bis etwa $O(n \cdot log(n))$ sind die Komplexitätsklassen günstig. Optimal und wünschenswert, wenn auch für viele Algorithmen nicht erreichbar, sind die Komplexitäten $O(1)$ und $O(log(n))$. Bereits $O(n^2)$ ist für größere Eingabemengen in der Regel nicht mehr günstig, kann aber bei einfachen Berechnungen und kleineren Werten für n problemlos eingesetzt werden.

Hinweis: Einfluss der Eingabedaten

Einige Algorithmen verhalten sich abhängig von den Eingabedaten unterschiedlich. Für Quicksort ergibt sich im durchschnittlichen Fall eine Komplexität von $n \cdot log(n)$, die aber im Extremfall auf n^2 ansteigen kann. Da die O-Notation den »Worst Case« beschreibt, wird Quicksort eine Komplexität von $O(n^2)$ zugeordnet.

Auswirkungen der Komplexität auf die Programmlaufzeit

Mögen die durch eine spezielle O-Komplexität errechneten Zahlen für eine Menge von Eingabewerten n manchmal auch abschreckend sein, so sagen diese jedoch nichts über die konkrete Ausführungszeit aus, sondern nur über deren Wachstum bei einer Vergrößerung der Eingabemenge. Wie bereits anhand des einführenden Beispiels deutlich wird, macht die O-Notation keine Aussage über die Dauer einzelner Berechnungsschritte: Das Inkrement `i++` und der Datenbankzugriff `createPersonInDb(i)` wurden jeweils mit $O(1)$ bewertet, obwohl der Datenbankzugriff auf die Ausführungszeit bezogen um mehrere Größenordnungen teurer als das Inkrement ist.

Für »normale« Anweisungen ohne Zugriffe auf externe Systeme, wie Dateisystem, Netzwerk oder Datenbank, also Additionen, Zuweisungen usw., ist der Einfluss von n bei heutigen Computern für typische Business-Applikationen mit Benutzerinteraktionen vielfach nicht ausschlaggebend: Für kleine n (< 1000) ist die tatsächliche Laufzeit bei den Komplexitäten $O(n)$ bzw. $O(n^2)$ und sogar manchmal noch $O(n^3)$ heutzutage eher gering – das heißt jedoch nicht, dass man nicht möglichst optimale Algorithmen verwenden sollte. Vielmehr gilt im Umkehrschluss: Man kann auch mit einer ersten funktional korrekten Umsetzung starten und diese in Produktion bringen. Die optimierte Variante lässt sich dann einige Zeit später ausrollen.

Der obigen Argumentation folgend, werden auch mehrfach ineinander verschachtelte Schleifen mit der Komplexität $O(n^2)$ oder $O(n^3)$ absolut gesehen oftmals viel schneller ausgeführt als Datenbankabfragen über ein Netzwerk mit der Komplexität $O(n)$. Ähnliches gilt für eine Suche in einem Array ($O(n)$) und einen Zugriff auf ein Element einer hashbasierten Datenstruktur ($O(1)$). Für kleine n kann die Berechnung der Hashwerte länger dauern als eine lineare Suche. Je größer allerdings n wird, desto mehr wirkt sich die schlechtere Komplexitätsklasse auf die tatsächliche Laufzeit aus.

22.2 Einsatz geeigneter Datenstrukturen

Wie bereits in der Einleitung angedeutet, lassen sich viele Performance-Probleme auf Fehler im Algorithmus oder in der Benutzung ungeeigneter Datenstrukturen zurückführen. Auch hier zitiere ich gerne wieder Jon Bentley [3]: »Eine richtige Sicht der Daten strukturiert tatsächlich die Programme. [...] Immer für einfachen Code zu sorgen, ist normalerweise der Schlüssel zur Korrektheit!« Im Folgenden werden einige Implikationen auf die Performance durch die Wahl der Datenstrukturen Arrays, Listen, Sets und Maps erläutert und mithilfe der eingeführten O-Notation bewertet. Dabei betrachten wir einige gebräuchliche Operationen wie Einfügen oder Zugriff auf Elemente.

Aus dem Nähkästchen: Wahl von Datenstrukturen

Ich habe einmal einen Programmteil überarbeitet, der Jobs und Gruppen von Jobs verwalten sollte: Eigentlich ein klarer Fall für den Einsatz von Baumstrukturen und der klassische Ansatz wäre das KOMPOSITUM-Muster (vgl. Abschnitt 18.2.4). Der damalige Entwickler hatte wenig Erfahrung und kannte Baumstrukturen nur unzureichend (allerdings war dies Ende der 90er in C++ auch keine gebräuchliche, sofort verfügbare Datenstruktur).

Um Jobs und Gruppen von Jobs zu verwalten, wurde versucht, den Job-Baum in einer Liste zu speichern. So etwas kann man machen, allerdings erfordert dies eine präzise Vorstellung, wie man Baumoperationen auf Listenoperationen abbildet. Selbstverständlich sind die Operationen »Einfügen« und »Löschen« von Jobs nicht einfach zu realisieren: In einer Liste müssen diverse Elemente verschoben und anschließend einige Referenzanpassungen vorgenommen werden. Bei Verwendung eines Baums als Datenstruktur sind nur einfache Operationen auf Baumknoten auszuführen, die Komplexität wird durch die Datenstruktur gekapselt.

Der Einsatz eines Baums als Datenstruktur half dabei, den zuvor extrem komplizierten Sourcecode innerhalb von wenigen Tagen komplett umzustellen und zu vereinfachen. Sowohl Komplexität als auch Umfang wurden deutlich reduziert. Zusätzlich kam es zu dem positiven Nebeneffekt, dass alle durch den ursprünglichen Ansatz verursachten Fehler gleich mit behoben wurden. Neben den positiven Auswirkungen auf Wartbarkeit und Erweiterbarkeit war die neue Realisierung deutlich performanter, vor allem für die Anwendungsfälle, in denen die Tiefe und Breite des Baums aufgrund einer komplexen Job-Gruppen-Struktur zunahm.

22.2.1 Einfluss von Arrays und Listen

In vielen Programmen müssen Daten in Arrays bzw. Listen gespeichert oder verarbeitet werden. Am Beispiel der Datenstrukturen `ArrayList<E>`, `LinkedList<E>`, `Vector<E>` und Array weise ich auf mögliche Folgen für die Performance hin.

Die Klassen `ArrayList<E>` und `Vector<E>`

Erinnern wir uns an den Aufbau und die Wirkungsweise der Datenstrukturen `ArrayList<E>` und `Vector<E>`, die in Kapitel 6 beschrieben sind. Beide speichern ihre Daten in einem Array und stellen somit eine komfortable Hülle um ein Array dar, die eine Schnittstelle mit Containerfunktionalität anbietet.

Zugriff auf ein Element an Position i Der Zugriff auf beliebige Elemente wird durch einen indizierten Array-Zugriff an Position i realisiert und dessen Ausführungszeit ist unabhängig von der Anzahl der gespeicherten Elemente ($O(1)$).

Neues Element an Position i hinzufügen Wenn an einer beliebigen Position i ein Element eingefügt wird, müssen alle Elemente mit einem Index $>= i$ nach hinten verschoben werden, um Platz für das neue Element zu schaffen ($O(n)$). Reicht die Kapazität nicht mehr aus, so muss zusätzlich eine Erweiterung der Datenstruktur erfolgen. Es wird ein neues, um 50 % vergrößertes Array erzeugt[5], und alle Elemente aus dem alten Array werden in dieses neue kopiert ($O(n)$).

Im günstigsten Fall wird ein neues Element an der letzten Position der Datenstruktur hinzugefügt. Wenn genügend Platz vorhanden ist, muss die Referenz im Array lediglich an der gewünschten Position abgelegt werden ($O(1)$). Ist jedoch kein Platz für das neue Element mehr vorhanden, so muss dafür Platz geschaffen werden ($O(n)$).

Element an Position i entfernen Wenn an einer beliebigen Position i ein Element gelöscht wird, müssen alle dahinter liegenden Elemente mit einem Index $> i$ um eine Position nach vorne kopiert werden. Dadurch ergibt sich eine Komplexität von $O(n)$.

Die Klasse `LinkedList<E>`

Die Klasse `LinkedList<E>` arbeitet intern mit untereinander verbundenen Knoten zur Speicherung von Elementen.[6] Es finden beim Einfügen und Löschen keine Verschiebe- und Kopieraktionen statt, da jeweils nur Referenzen zwischen den Knoten angepasst werden müssen. Betrachten wir, wie sich dies im Gegensatz zur Speicherung in einer `ArrayList<E>` bzw. einem `Vector<E>` auswirkt.

[5] Schauen Sie in den Sourcen des JDKs nach, um solche internen Details kennenzulernen.

[6] In einer `LinkedList<E>` werden drei Referenzen (Daten, Vorgänger und Nachfolger) statt einer in einer `ArrayList<E>` gespeichert. Dadurch wird im Vergleich zur `ArrayList<E>` etwa der dreifache Speicher für die Verwaltung verbraucht, allerdings nicht zusammenhängend.

Zugriff auf ein Element an Position i Um auf ein Element an einer Position i zuzugreifen, muss vor dem Zugriff ($O(1)$) zunächst bis zu dieser gewünschten Position über alle Vorgängerknoten iteriert werden ($O(n)$).

Neues Element an Position i hinzufügen Der Einfügevorgang besteht lediglich aus dem Erzeugen eines neuen Knotens und aus einigen Referenzanpassungen und besitzt daher die Komplexität $O(1)$. Allerdings führt die vorher durchzuführende Suche nach der Einfügeposition insgesamt zur Komplexität $O(n)$.

Element an Position i entfernen Der Löschvorgang besteht nur aus Referenzanpassungen und besitzt die Komplexität $O(1)$. Auch hier führt die vorher durchzuführende Suche nach der Löschposition insgesamt zur Komplexität $O(n)$.

Betrachtung von Vor- und Nachteilen

In den vorherigen Abschnitten wurden bereits einige Auswirkungen verschiedener Operationen besprochen. Es ergibt sich folgende Vermutung: Die Anzahl der Daten, die Dynamik in der Zusammensetzung und die Position von Änderungen innerhalb der Datenstruktur besitzen großen Einfluss auf die Performance. Die Effekte sollen sich für verschiedene Implementierungen des Interface `List<E>` deutlich unterscheiden. Rekapitulieren wir die wichtigsten Punkte: Für die `ArrayList<E>` bzw. den `Vector<E>` vermuten wir Performance-Probleme durch Einfüge- und Löschoperationen, die zu einem Verschieben von einigen Elementen führen – im ungünstigsten Fall sogar von allen. Beide Operationen sind auch für die `LinkedList<E>` durch die Suche nach der korrekten Position für Einfüge- oder Löschoperationen ausgehend vom Start- bzw. Endknoten aufwendig. Beides scheint für größere Datenmengen nicht optimal zu sein.

Wir folgen den Empfehlungen aus Abschnitt 22.1 und führen deshalb Messungen durch, um die obigen Vermutungen zu bestätigen oder zu widerlegen. Für die Klassen `ArrayList<E>`, `LinkedList<E>` und `Vector<E>` erfolgen jeweils Messungen der Operationen Einfügen, Löschen und Iterieren für 10.000, 100.000 und 1.000.000 Elemente. Für die `ArrayList<E>` ermitteln wir zusätzlich die Laufzeiten mit initialer Kapazität von 100.000 Elementen. Das korrespondierende Programm ist stark gekürzt in folgendem Listing gezeigt:

```
public final class ListPerformanceExample
{
    private static final SimplePerson OBJ_TO_ADD = new SimplePerson("Test", 42);

    private static final long[] maxElements = { 10_000, 100_000, 1_000_000 };

    public static void main(final String[] args)
    {
        performListTests("ArrayList", new ArrayList<SimplePerson>());
        performListTests("ArrayList IK", new ArrayList<SimplePerson>(100_000));
        performListTests("Vector", new Vector<SimplePerson>());
        performListTests("LinkedList", new LinkedList<SimplePerson>());

    }
```

```
private static void performListTests(final String dsName,
                                     final List<SimplePerson> list)
{
    for (final long max : maxElements)
    {
        System.out.println("Element count " + max);

        list.clear();
        PerformanceUtils.startMeasure(dsName + " add front");
        addPersonFront(list, max);
        PerformanceUtils.stopMeasure(dsName + " add front");
        PerformanceUtils.printTimingResult(dsName + " add front");

        PerformanceUtils.startMeasure(dsName + " remove front");
        removePersonFront(list, max);
        PerformanceUtils.stopMeasure(dsName + " remove front");
        PerformanceUtils.printTimingResult(dsName + " remove front");

        // ...
```

Zur Erinnerung: Die Messergebnisse sind für JDK 14.0.1 auf einem iMac (Quad-Core i7 mit 4 GHz) entstanden und in Tabelle 22-3 dargestellt. Zum Vergleich können Sie das Programm LISTPERFORMANCEEXAMPLE ausführen.

Tabelle 22-3 *Performance-Messungen für Listen*

		add [ms]			remove [ms]			[ms]
Typ	**Elemente**	**Anfang**	**Mitte**	**Ende**	**Anfang**	**Mitte**	**Ende**	**for**
`ArrayList`	10.000	11	1	0	4	1	0	0
`ArrayList`	100.000	575	351	1	451	203	2	1
`ArrayList`	1.000.000	72.177	29.792	8	65.535	28.775	1	0
`ArrayList IK`	10.000	14	1	0	3	1	0	0
`ArrayList IK`	100.000	501	225	0	513	226	0	0
`ArrayList IK`	1.000.000	82.296	31.028	8	75.573	30.799	1	1
`Vector`	10.000	4	2	0	4	4	1	1
`Vector`	100.000	526	227	1	559	214	1	1
`Vector`	1.000.000	87.493	29.360	12	70.094	28.972	99	13
`LinkedList`	10.000	2	38	0	1	43	1	0
`LinkedList`	100.000	3	3.924	1	1	3.629	0	0
`LinkedList`	1.000.000	77	726.338	13	23	733.759	14	3

Bevor ich die Ergebnisse inhaltlich bewerte, möchte ich noch etwas zur Genauigkeit der Angaben sagen: Da die Zeitmessung auf einem System immer gewissen Schwankungen unterliegt und es sich um auf Millisekunden gerundete Angaben handelt, sind Zahlenwerte von 0 bis 20 ms durchaus als nahezu gleich anzusehen. Ein Wert zwischen etwa 50.000 ms und 100.000 ms entspricht ungefähr 1 bis 2 Minuten, 700.000 ms sind demnach etwa 11 Minuten. Erwartungsgemäß erhält man diesen schlechtesten Wert bei der `LinkedList<E>` für das Hinzufügen und Löschen in der Mitte. Interessanterweise ist aber das Entfernen in der Mitte mit Java 14 oftmals doppelt bis dreimal so schnell wie mit Java 8.

Entgegen den zuvor geäußerten Erwartungen wirken sich Vergrößerungsschritte der Klasse `ArrayList<E>` nicht signifikant aus. Dies zeigen die Zahlen, die für die `ArrayList<E>` ohne eine Festlegung einer initialen Kapazität und für eine solche mit der Anfangskapazität von 100.000 Elementen, die `ArrayList IK`, ermittelt wurden. Auch die in anderer Literatur als »teuer« bezeichneten synchronisierten Methoden der Klasse `Vector<E>` haben bei *nicht* nebenläufigen Zugriffen auf die Datenstruktur kaum merklichen Einfluss – das ändert sich erst dann, wenn eine Vielzahl an Threads um die Locks konkurriert. Zudem kann man erkennen, dass ein Iterieren mit einer `for`-Schleife über alle Elemente relativ schnell und günstig ist. Problematisch sind, wie erwartet, Einfüge- bzw. Löschoperationen. Allerdings sind die gemessenen Unterschiede bis etwa 10.000 gespeicherte Elemente recht gering. Darunter, etwa für 1.000 Einträge, sind die ermittelten Zeiten unter der Messbarkeitsgrenze und werden daher hier nicht aufgelistet. Nur bei größeren Datenmengen machen sich die zuvor diskutierten Effekte sehr deutlich bemerkbar. ***Die gemessenen Werte zeigen, dass die Einfüge- bzw. Löschposition entscheidenden Einfluss auf die Performance besitzt***. Für Array-basierte Listen ist das Einfügen und Löschen im vorderen Bereich der Liste am teuersten und nimmt Richtung Ende der Datenstruktur signifikant ab, da viel weniger Elemente bewegt werden müssen. Für die `LinkedList<E>` sind Einfügen und Löschen am Ende und Anfang der Datenstruktur nahezu unabhängig von der Menge der gespeicherten Werte. Dafür machen sich Positionen im Bereich der Mitte der Datenstruktur sehr viel stärker bemerkbar, als man es intuitiv vermuten würde. Beim Einfügen bzw. Löschen von einer Million Daten an der Mittenposition sind Array-basierte Listen etwa um den Faktor 25 schneller.

Lediglich bei einer großen Anzahl an Elementen und einer hohen Dynamik in der Zusammensetzung am Beginn der Liste kann die `LinkedList<E>` aufgrund ihres internen Aufbaus besser als eine `ArrayList<E>` geeignet sein. Dies gilt allerdings nur dann, wenn im Programmverlauf wenig indizierte Zugriffe erfolgen.

Insgesamt betrachtet skalieren die Array-basierten Listen besser. Die `LinkedList<E>` zeigt verheerende Performance-Einbrüche bei großen Datenmengen (ab einigen 100.000 Elementen). Das gilt vor allem bei Aktionen nahe der Listenmitte. Für derartige Zugriffe sind die `ArrayList<E>` und der `Vector<E>` deutlich besser geeignet. Wie wir später anhand eines Beispiels sehen werden, gilt dies im Besonderen für Tabellenmodelle. Der Einsatz einer `LinkedList<E>` ist dort extrem kontraproduktiv.

Tipp: Konfigurierbarkeit durch Einsatz des Interface `List<E>`

Programmiert man konsequent gegen das Interface `List<E>`, so kann man die Wahl einer konkreten Listenimplementierung konfigurierbar machen und diese mithilfe einer FABRIKMETHODE (vgl. Abschnitt 18.1.2) erzeugen. Dieses Vorgehen bietet sich in der Praxis immer dann an, wenn man die Dynamik und das Datenvolumen schlecht im Voraus schätzen kann. Durch nachfolgende Profiling-Messungen lässt sich die für den Anwendungsfall geeignete Realisierung ermitteln.

22.2.2 Optimierungen für `Set` und `Map`

Nachdem wir verschiedene Implementierungen von Listen auf ihre Performance untersucht haben, werfen wir nun einen Blick auf die Realisierungen der Interfaces `Set<E>` und `Map<K,V>`. Diese sind bereits für viele Anwendungsfälle ausreichend optimiert. Durch die passende Wahl einer für einen konkreten Anwendungsfall geeigneten Realisierung sind jedoch weitere Performance-Gewinne möglich. Die folgenden Abschnitte beschreiben daher verschiedene Performance-Implikationen und die Laufzeiten einiger wichtiger Zugriffsmethoden (Einfügen, Löschen und Zugriff auf Elemente) für konkrete Typen von Sets und Maps.

Die Klassen `HashSet<E>` und `HashMap<K,V>`

Die Realisierung der Klasse `HashSet<E>` verwendet intern die Klasse `HashMap<K,V>`. Daher können hier – bezogen auf die Auswirkungen auf die Performance – beide Datenstrukturen gemeinsam betrachtet werden. Die Implementierung der Klasse `HashMap<K,V>` ist extrem performant. Die Abfrage der Anzahl gespeicherter Einträge (`size()`) besitzt eine Laufzeit von $O(1)$. Auch die gebräuchlichsten Operationen `put()`, `get()`, `remove()` und `containsKey()` haben alle in der Regel eine Laufzeit von $O(1)$. Nur bei einer extrem ungünstig gewählten Hashfunktion (z. B. Implementierung der `hashCode()`-Methode, die einen konstanten Wert liefert) kann das natürlich nicht mehr gelten. Weitere Details finden Sie in Abschnitt 6.1.7.

Einfluss von `equals()`, `compareTo()` und `hashCode()` Neben der Wahl einer geeigneten Datenstruktur lässt sich eine weitere Verbesserung der Performance indirekt dadurch erreichen, dass man die steuernden Methoden optimiert. Wie bereits in Abschnitt 6.1.9 diskutiert, sind dies für Sets und Maps die Methoden `equals()`, `hashCode()` und `compareTo()` bzw. `compare()`. Die Methoden `equals()` und `compareTo()` besitzen in der Regel nur geringen Einfluss auf die Performance. Für beide Methoden besteht ein Optimierungspotenzial darin, zum einen lediglich die tatsächlich für den Vergleich relevanten Attribute zu betrachten und zum anderen Auto-Boxing und Auto-Unboxing möglichst zu vermeiden. Abschnitt 22.7.3 geht auf die Auswirkungen dieser Optimierung genauer ein. Auf keinen Fall sollte man auf dieser CPU-basierten Ebene beginnen, ohne vorher alle anderen Optimierungsmöglichkeiten ausgeschöpft zu haben.

Die Methode `hashCode()` kann im Gegensatz zu den zuvor besprochenen Methoden größeren Einfluss auf die tatsächliche Performance besitzen. Bei der Implementierung sollten eine möglichst gleichmäßige Verteilung bzw. eine gute Streuung der berechneten Hashwerte und damit eine ausgewogene Verteilung der Objekte auf die Buckets angestrebt werden. Ansonsten nehmen die Kollisionen zu und im Extremfall verhält sich ein `HashSet<E>` bzw. eine `HashMap<K,V>` nahezu wie eine lineare Liste mit vorangestellter Hashberechnung. Damit werden die Zugriffszeiten deutlich schlechter als bei einer ausgewogenen Verteilung der Objekte auf alle Buckets.

Neben der Implementierung der Methode `hashCode()` sind für hashbasierte Container bei Performance-Optimierungen zudem folgende weitere Dinge zu beachten:

1. **Kapazität und Füllgrad** – Um Kollisionen möglichst zu vermeiden und eine gute Performance beim Zugriff zu erzielen, sollte der Füllgrad erfahrungsgemäß maximal im Bereich von ca. 75 % liegen. Wird der Füllgrad größer gewählt, so wird zwar der Speicher besser genutzt, allerdings kommt es dann auch häufiger zu Kollisionen, wodurch sich die Kosten beim Zugriff auf Elemente erhöhen.
2. **Vermeidung von Größenanpassungen und Rehashing** – Die initiale Kapazität sollte auf der geplanten Nutzlast basieren: Die voraussichtliche Anzahl von Elementen wird dazu mit dem Faktor 1,3 multipliziert (vgl. Abschnitt 6.1.7). Dadurch bleiben zwar etwa ein Viertel aller Buckets unbelegt, aber man vermeidet so weitgehendst automatische Größenanpassungen, die wiederum dazu führen, dass die gespeicherten Elemente neu auf die Buckets verteilt werden müssen. Bekanntermaßen kostet ein solcher Umsortiervorgang zunächst einmal Rechenzeit, sorgt aber dafür, dass spätere Zugriffe wieder performant sind.

Die Klasse `ConcurrentHashMap<K,V>`

Die Klasse `ConcurrentHashMap<K,V>` realisiert eine auf hohe Parallelität ausgelegte hashbasierte Map, deren Methodenkontrakte sich nicht, wie der Name vermuten lässt, an denen der Klasse `HashMap<K,V>`, sondern an denen der Klasse `Hashtable<K,V>` orientieren. Dadurch ist diese Map ein Ersatz für eine `Hashtable<K,V>`. Soll die `ConcurrentHashMap<K,V>` als Ersatz der Klasse `HashMap<K,V>` dienen, so ist dies in der Regel problemlos möglich. Es dürfen jedoch aufgrund der gewählten Umsetzung keine `null`-Werte für Schlüssel und Wert gespeichert werden.

Die zuvor gemachten Performance-Betrachtungen für Methoden der Klasse `HashMap<K,V>` gelten gleichermaßen für die Klasse `ConcurrentHashMap<K,V>`. Allerdings sind die konkreten Ausführungszeiten der Methoden einer `ConcurrentHashMap<K,V>` etwas höher, da intern zusätzlich einige Aktionen notwendig sind, um Thread-Sicherheit zu erreichen. Nur in einem Punkt gibt es einen signifikanten Unterschied: Die Abfrage der Größe über die Methode `size()` ist bei der `ConcurrentHashMap<K,V>` nicht in konstanter Zeit ($O(1)$) möglich, sondern erfordert signifikant mehr Rechenzeit, da alle Elemente zur Ermittlung der Größe traversiert werden müssen ($O(n)$). Aufgrund der genannten Arbeitsweise sollte die `ConcurrentHashMap<K,V>` mit Bedacht und gezielt dann verwendet werden, wenn tatsächlich Bedarf an Nebenläufigkeit mit häufigen, konkurrierenden Zugriffen besteht. Bei seltenen konkurrierenden Zugriffen kann die `ConcurrentHashMap<K,V>` ihre Vorteile nicht adäquat ausspielen.

Würde man eine `ConcurrentHashMap<K,V>` anstelle einer `HashMap<K,V>` für alle Anwendungsfälle nutzen, die eine hashbasierte Speicherung ohne konkurrierende Zugriffe benötigen, so suggerierte man damit zudem, dass ein Programmstück auf Parallelität ausgelegt ist. Ist dies nicht der Fall, ist die Wahl dieser Datenstruktur irreführend, da sie nicht nahe am zu lösenden Problem ausgerichtet ist.

Die Klassen `TreeSet<E>` und `TreeMap<K,V>`

Da die Klasse `TreeSet<E>` unter Verwendung der Klasse `TreeMap<K,V>` implementiert ist, können wieder beide Datenstrukturen gemeinsam betrachtet werden.

Die in vielen Anwendungsfällen eingesetzten Methoden `containsKey()`, `get()`, `put()` und `remove()` besitzen aufgrund der Verwendung eines balancierten Baums als Datenstruktur alle eine garantierte Laufzeit von $O(log(n))$. Eine Größenabfrage über `size()` besitzt die Komplexität $O(1)$. Beide Realisierungen sind demnach sehr performant. Da die Daten in einer sortierten Reihenfolge in diesem Baum abgelegt werden (müssen), sind die Ausführungsgeschwindigkeiten für `put()` und `remove()` etwas geringer als diejenigen einer `HashMap<K,V>`.

Die Klassen `ConcurrentSkipListSet<E>` und `ConcurrentSkipListMap<K,V>`

Die Klasse `ConcurrentSkipListSet<E>` ist eine Thread-sichere Realisierung des Interface `SortedSet<E>` und verwendet zur Implementierung ihrer Funktionalität eine spezielle Map, die `ConcurrentSkipListMap<K,V>`. Für beide Klassen können im Gegensatz zu den nicht Thread-sicheren Realisierungen `TreeSet<E>` und `TreeMap<K,V>` sowohl Einfüge- als auch Löschoperationen gefahrlos durch mehrere Threads ausgeführt werden.

Die zuvor gemachten Performance-Betrachtungen für die Methoden der Klasse `TreeMap<K,V>` gelten hier gleichermaßen. Analog zur `ConcurrentHashMap<K,V>` werden die Methoden einer `ConcurrentSkipListMap<K,V>` durch den Mehraufwand zum Erreichen von Thread-Sicherheit etwas langsamer als die einer `TreeMap<K,V>` abgearbeitet. Weiterhin gilt hier wieder, dass die Abfrage der Größe über `size()` nicht in konstanter Zeit ($O(1)$) möglich ist, sondern deutlich mehr Rechenzeit erfordert ($O(n)$). Demzufolge sollten beide Datenstrukturen nur dann verwendet werden, wenn tatsächlich Bedarf an (hoher) Parallelität besteht.

22.2.3 Design eines Zugriffsinterface

Neben dem Einfluss einiger Algorithmen des Collections-Frameworks kann auch die Gestaltung des Zugriffs auf eine Datenstruktur Performance-relevant sein.

Aufgabenkontext und Anforderungsdefinition

Nehmen wir an, eine Klasse verwaltet `Person`-Objekte in einer beliebigen Containerklasse, etwa einer `ArrayList<Person>`. Nehmen wir weiter an, ein Anwendungsfall bestünde darin, sämtliche Wohnorte von Personen zu ermitteln und diese bereitzustellen. Es ist demnach nur ein kleiner Teil eines `Person`-Objekts, der Wohnort, für Klienten von Interesse.

Betrachten wir nun mögliche Realisierungen eines Zugriffsinterface und diskutieren kurz deren Vor- und Nachteile.

Realisierung einer spezialisierten Zugriffsmethode

Zur Umsetzung der Anforderungen wird lediglich eine Liste von Städten benötigt. Eine naheliegende Realisierungsidee besteht darin, ein `Set<String>` durch eine Iteration über alle gespeicherten Kunden und den Aufruf von `getCity()` zu ermitteln:

```
public Set<String> getCitySet()
{
    return persons.stream().
                   map(Person::getCity).
                   collect(Collectors.toSet());
}
```

Durch die Realisierung der Methode `getCitySet()` sind die Anforderungen sehr gut erfüllt. Auch die Datenkapselung ist gelungen: Es wird kein direkter Zugriff auf die interne Datenstruktur `persons` geboten. Die Aufbereitung der Ergebnisse in der Methode `getCitySet()` ist vorteilhaft: Die Kopplung ist gelöst und die Ergebnisse können unabhängig von `Person`-Objekten verarbeitet werden. Sogar die interne Form der Speicherung kann geändert werden, ohne Auswirkungen auf Klienten zu verursachen.

Realisierung mithilfe eines Iterators

Im vorherigen Beispiel wurde von `Person`-Objekten vollständig abstrahiert und nur auf einer Menge von Strings gearbeitet. Dadurch wird bereits eine gute Datenkapselung erreicht. Möchte man noch weiter von der eingesetzten Datenstruktur abstrahieren, könnte man auf die Idee kommen, Klienten lediglich einen Iterator auf die Ergebnisdatenstruktur zu liefern. Diese Idee wird durch folgende Methode `getCityIterator()` umgesetzt, in der zunächst über die Methode `getCitySet()` eine Ergebnismenge mit den Namen von Städten erstellt und anschließend ein `Iterator<String>` zurückgeliefert wird:

```
public final Iterator<String> getCityIterator()
{
    return getCitySet().iterator();
}
```

Mit dieser Realisierung werden die beiden Ziele gute Datenkapselung und lose Kopplung erreicht. Diese Lösung scheint wirklich richtig gut zu sein! Oder doch nicht?

API-Design `Collection<E>` vs. `Iterator<E>`

Die Rückgabe einer Ergebnismenge als `Set<String>` ermöglicht es Klienten, die von ihnen benötigte Containerfunktionalität direkt zu nutzen. Sie können beispielsweise durch einen Aufruf von `size()` feststellen, wie viele Elemente die Ergebnismenge enthält, oder mit `contains(Object)` abfragen, ob ein spezielles Element gespeichert ist. Die Realisierung dieser Containerfunktionalität erfordert mit einem Iterator jeweils einen manuellen Durchlauf über alle Elemente. Das kann aufwendig werden, wenn derartige Funktionalität häufig benötigt wird.

Die Einschränkung einer Rückgabe auf das Interface `Iterator<String>` ist dann unpraktisch, weil dadurch die Auswertung von Ergebnissen mithilfe der Algorithmen und Methoden der Containerklassen des Collections-Frameworks erschwert wird. Allerdings bietet die Lösung mit Iterator eine bessere Kapselung und löst die Kopplung. Je nach Anwendungsfall ist also die eine oder andere Lösung zu bevorzugen.

Zuvor habe ich motiviert, warum es bei der Rückgabe kleiner Datenmengen häufig praktischer ist, das `Collection<E>`-Interface zu nutzen, statt lediglich Zugriff auf ein `Iterator<E>`-Objekt anzubieten. Für sehr große Ergebnismengen kann es sinnvoll sein, anstelle des Typs `Collection<E>` ein `Iterator<E>`-Objekt zurückzugeben. Außerdem ist ein Iterator vorzuziehen, wenn die Ergebnismenge zwischen verschiedenen Rechnern übertragen werden muss. Die Idee dahinter ist, statt einmalig eine große Ergebnismenge bereitzustellen, diese schritt- oder blockweise anzufordern. Benötigen Berechnungen lediglich Zugriff auf aufeinanderfolgende Elemente, so kann durch einen Iterator ein sukzessives Durchlaufen der Daten gewährleistet werden, ohne viel Speicher zu verbrauchen. Diese Technik kommt beispielsweise bei der Suche im Internet und der Präsentation von Ergebnissen zum Einsatz. Häufig reicht bereits der erste Schwung an möglichen Treffern aus, der relativ schnell bereitgestellt werden kann. Müsste stattdessen zunächst die Ergebnismenge vollständig an einen Klienten übertragen werden, so hätte dieser eventuell recht lange zu warten. Außerdem müsste er die gelieferte Datenmenge zwischenspeichern und verarbeiten können.

Info: Einfluss des Algorithmus und der Ausführungsschicht

Leider sieht man immer wieder unnötige, aber aufwendige Berechnungen. Beispielsweise werden mithilfe von SQL-Abfragen viele Datenbankeinträge ausgewählt und die ermittelten Ergebnisdaten als Liste von DTOs (vgl. Abschnitt 3.4.5) an einen Aufrufer übertragen. Wird aber nur die Größe der Liste abgefragt und ist der Inhalt der Datenobjekte nicht von Interesse, so war die aufwendige Konstruktion der DTOs sowie das Übertragen der Daten überflüssig. Als Folge müssen alle temporär konstruierten Objekte anschließend vom Garbage Collector weggeräumt werden.

Sinnvollerweise ermittelt man über eine `SELECT COUNT(*)`-Abfrage einfach die Anzahl der Datensätze als `int`-Wert. Dies kommt der Performance zugute, da zum einen die Aktionen in der Datenbank selbst erfolgen (also nahe am zu lösenden Problem) und zum anderen viel weniger Speicher zum Transport und weniger Zeit zum Aufbereiten der Ergebnisse benötigt wird.

Die für beide Varianten eingesetzten Algorithmen sind nahezu gleich. Allerdings sind die ausführenden Stellen unterschiedlich. Einmal wird das Zählen der Einträge auf Applikationsebene gelöst, im zweiten Fall direkt in der Datenbank. An diesem Beispiel erkennt man gut, dass die Wahl des richtigen Abstraktionsniveaus sowie der passenden ausführenden Systemkomponente großen Einfluss auf die Performance besitzen können.

22.3 Lazy Initialization

Die sogenannte ***Lazy Initialization*** ist eine Optimierungstechnik, deren Ziel es ist, so wenig laufzeitintensive Aktionen wie möglich durchzuführen. Daher werden nur tatsächlich benötigte oder nur wenig Aufwand verursachende Programmteile beim Programmstart initialisiert. Konkret heißt das: Berechnungen und Initialisierungen komplexer Objekte werden so lange verzögert, bis von anderen Programmkomponenten darauf zugegriffen wird, wodurch man Arbeitsschritte vermeidet, deren Ergebnisse eventuell niemals benötigt werden.

Lazy Initialization am Beispiel

Wir wollen die Optimierung durch Lazy Initialization anhand eines vereinfachten Beispiels aus der Praxis nachvollziehen. Betrachten wir dazu eine Applikation, die Kunden und deren Adressen (Privat-, Firmen- und Lieferadresse) verwaltet.[7] Zur Unterscheidung verschiedener Adressinformationen ist eine `enum`-Aufzählung `AddressType` mit den Werten `HOME`, `WORK`, `DELIVERY` definiert. Kunden werden durch die Klasse `Customer` modelliert. Diese aggregiert diverse andere Klassen wie folgt:

```
public class Customer
{
    private final long customerId;
    private AddressInfo homeAddress, workAddress, deliveryAddress;
    private CustomerInfo customerInfo;
    private ContactInfo contactInfo;

    public Customer(final long newCustomerId)
    {
        customerId = newCustomerId;

        homeAddress = new AddressInfo(customerId, AddressType.HOME);
        workAddress = new AddressInfo(customerId, AddressType.WORK);
        deliveryAddress = new AddressInfo(customerId, AddressType.DELIVERY);

        customerInfo = new CustomerInfo();
        contactInfo = new ContactInfo();
    }

    // ...
```

Im Konstruktor der Klasse `Customer` werden verschiedene aggregierte `AddressInfo`-Objekte erzeugt. Für diese existieren Zugriffsmethoden analog zu der folgenden, die einen direkten Zugriff auf interne Referenzen auf `AddressInfo`-Objekte bieten:

```
public AddressInfo getHomeAddress()
{
    return homeAddress;
}
```

[7] Die Idee zur Darstellung der Lazy Initialization anhand dieser Klassenstruktur stammt aus einem Vortrag von Peter Haggar auf der Java One 2001. Ich habe das gesamte Beispiel aber bezüglich Problembeschreibung und Lösungsmöglichkeiten stark erweitert.

```
public AddressInfo getWorkAddress()
{
    return workAddress;
}

public AddressInfo getDeliveryAddress()
{
    return deliveryAddress;
}
```

Dadurch ist es Klienten allerdings möglich, den Objektzustand, hier die Bestandteile der Adressen, unkontrolliert von außen zu verändern. Das ist normalerweise unerwünscht und zu vermeiden. Da die am Ende dieses Abschnitts dargestellte Lösung dieses Problem auf elegante Art und Weise behebt, gehe ich im Folgenden auf eine andere Abhilfe nicht weiter ein.

Nehmen wir an, die obigen Methoden würden von einer Applikation verwendet, um einige Informationen in der folgenden Methode auszugeben:

```
public static void printHomeAddress(final int customerId)
{
    final Customer customer = new Customer(customerId);
    final AddressInfo address = customer.getHomeAddress();

    System.out.println("Street" + address.getStreet());
    System.out.println("City" + address.getCity());

    //...
}
```

Betrachtet man das Listing für sich allein, scheint zunächst kein Problem zu existieren. Dieser erste Entwurf stellt sich allerdings beim Testen und anschließenden Debugging als Engpass heraus: Man erwartet, dass die Methode `printHomeAddress(int)` schnell abgearbeitet würde, doch die Ausführung dauert immer einige Sekunden.

Analyse

Bevor man größere Umbaumaßnahmen beginnt, sollte man durch Sourcecode-Analyse und Profiling-Messungen eine konkrete Vorstellung der Performance-Bremsen haben.

Für unser Beispiel erhält man durch Profiling einen Hinweis, dass ziemlich viel Zeit im Konstruktor der aggregierten Klasse `AddressInfo` verbracht wird. Schaut man die Implementierung an, so erkennt man, dass dort Informationen mit Datenbankabfragen im Konstruktor ermittelt werden:

```
AddressInfo(final long customerId, final AddressType addressType)
{
    this.customerId = customerId;
    this.addressType = addressType;

    initAddress(DBAccess.selectAddressByAddressType(customerId, addressType));
}
```

Ein solches Vorgehen kann zum einen sehr laufzeitintensiv sein und zum anderen zu Problemen durch nicht freigegebene Ressourcen beim Auftreten von Exceptions führen. Abschnitt 16.3.11 stellt dies als BAD SMELL: RESOURCE LEAKS DURCH EXCEPTIONS IM KONSTRUKTOR vor.

Bezogen auf die Performance erkennen wir folgendes Problem: Das komplexe `Customer`-Objekt wird immer vollständig mit all seinen aggregierten Objekten erzeugt. Allerdings benötigt die Applikation in dieser Ausbaustufe nur einen ganz geringen Teil eines `Customer`-Objekts, hier lediglich die über `getHomeAddress()` ermittelte Anschrift. Alle anderen aggregierten Teilkomponenten werden zwar erzeugt, aber niemals verwendet. Würde es sich dabei nur um kleine, einfache Objekte handeln, so wäre dies nicht allzu kritisch. Bei schwergewichtigen Objekten mit Datenbankzugriffen wie in diesem Fall wirkt sich dies allerdings negativ auf die Performance und den Speicherverbrauch aus. Betrachten wir nun, was man dagegen tun kann.

Einführen von Lazy Initialization

Bei der Konstruktion schwergewichtiger Objekte sind durch den gezielten Einsatz von Lazy Initialization – in diesem Fall zur Ermittlung der Adressinformationen – gute Performance-Gewinne zu erzielen. Dies gilt vor allem, wenn dadurch Netzwerkzugriffe oder Datenbankabfragen innerhalb von häufig durchlaufenen Programmstellen vermieden werden können. Teilweise scheint als Folge die Handbremse gelöst worden zu sein. Der Lazy Initialization liegen dazu folgende Gedanken zugrunde:

1. Verzögere Berechnungen und Initialisierungen wenn möglich: Konstruiere nur sofort benötigte Teile eines Objekts, aber nicht optionale, aggregierte Komponenten.
2. Initialisiere Referenzen auf optionale, aggregierte Komponenten mit `null` (alternativ `Optional.empty` oder mit dem NULL-OBJEKT-Muster).
3. Erzeuge Komponenten erst dann, wenn sie tatsächlich benötigt werden. Ein Stellvertreterobjekt gemäß dem PROXY-Muster (vgl. Abschnitt 18.3.6) kann helfen, mit dem ehemals schwergewichtigen Objekt bereits ansatzweise arbeiten zu können.

Betrachten wir, wie sich die Anwendung dieser Maßnahmenliste auf den Sourcecode des Beispiels auswirkt. Beginnen wir mit dem Konstruktor der Klasse `Customer`. Hier werden gemäß den Punkten 1 und 2 alle `AddressInfo`-Referenzen mit `null` initialisiert:

```
Customer(final int newCustomerId)
{
    customerId = newCustomerId;

    homeAddress = null;
    workAddress = null;
    deliveryAddress = null;
    customerInfo = new CustomerInfo();
    contactInfo = new ContactInfo();
}
```

Da die Klassen `CustomerInfo` und `ContactInfo` keine Datenbankabfragen durchführen und leichtgewichtig sind, wird für diese auf den Einsatz von Lazy Initialization verzichtet. Auch hier gilt: ***So wenig wie möglich, so viel wie nötig***. Der Grund ist einfach: Durch Lazy Initialization führt man Komplexität ein und erhöht somit die Wahrscheinlichkeit für Fehler. Wird die Initialisierung von aggregierten Objekten verzögert, so müssen dann gemäß Punkt 3 die jeweiligen Zugriffsmethoden prüfen, ob bereits eine Initialisierung erfolgt ist und gegebenenfalls ein benötigtes Objekt erzeugen:

```
// Achtung: Nicht Thread-sicher!
public AddressInfo getHomeAddressLazy()
{
    if (homeAddress == null)
    {
        homeAddress = new AddressInfo(this.customerId, AddressType.HOME);
    }
    return homeAddress;
}
```

Diese Methode ist funktional in Ordnung, weist jedoch noch Probleme bezüglich korrekter Initialisierung und Thread-Sicherheit auf. Darauf geht der folgende Abschnitt ein.

22.3.1 Konsequenzen des Einsatzes der Lazy Initialization

Der Einsatz von Lazy Initialization führt zu mehr Komplexität im Sourcecode: Als Folge muss an diversen Stellen geprüft werden, ob benötigte Komponenten bereits initialisiert sind oder nicht. Dies erhöht die Gefahr für Probleme, wenn Multithreading eingesetzt wird. Unter ungünstigen Bedingungen – und diese treten laut Murphy's Law mit ziemlicher Sicherheit auf – kommt es zu Synchronisationsproblemen bei der Initialisierung bzw. beim Zugriff auf Attribute. Dies ist dann der Fall, wenn mehrere Threads fast gleichzeitig zugreifen, möglicherweise veraltete Werte sehen und somit mehrfach eine Initialisierung durchführen. Das erinnert an die Probleme von `getInstance()`-Methoden beim SINGLETON-Muster (vgl. Abschnitt 18.1.4).

Synchronisationsprobleme

Als erste Abhilfe kommt das Schlüsselwort `synchronized` zum Einsatz:

```
public synchronized AddressInfo getHomeAddressLazyCorrected()
{
    if (homeAddress == null)
    {
        homeAddress = new AddressInfo(this.customerId, AddressType.HOME);
    }
    return homeAddress;
}
```

Werden alle Zugriffe auf Adressen derart geschützt, so verhindert man eine Parallelausführung der Zugriffsmethoden. Über jeweils eigene Synchronisationsobjekte kann bei Bedarf für etwas mehr Nebenläufigkeit gesorgt werden:

```
private static final Object homeAddressLock = new Object();
private static final Object workAddressLock = new Object();
```

Die Zugriffsmethoden werden dann jeweils über den speziellen Lock des korrespondierenden Synchronisationsobjekts geschützt. Abschnitt 10.2 beschreibt dies im Detail. Bei Bedarf nach mehr Parallelität und einer unterschiedlichen Sperre für Lese- bzw. Schreibzugriffe sollte man den Einsatz von `ReadWriteLock`s in Betracht ziehen – jedoch nur, wenn wenige Schreibzugriffe erfolgen sollen, weil ansonsten längere Verzögerungen zu beobachten sind.

Initialisierungsprobleme

Greift man auf lazy-initialisierte Attribute nicht über die dafür vorgesehenen Zugriffsmethoden zu, so kann es leicht zu `NullPointerException`s durch eine fehlende Initialisierung der Objekte kommen. Dies lässt sich zwar durch konsequentes Einführen von `null`-Prüfungen und durch Erzeugen benötigter Komponenten lösen, allerdings bläht sich der Sourcecode dadurch enorm auf. Zum einen dupliziert man die Prüfungen und zum anderen wird schnell eine wichtige Programmstelle übersehen.

Eine fehlertolerantere Lösung lässt sich durch den Einsatz des NULL-OBJEKT-Musters (vgl. Abschnitt 18.3.2) erzielen. Wir implementieren dazu eine Klasse `EmptyAddressInfo` wie folgt:

```
public class EmptyAddressInfo extends AddressInfo
{
    public static final long NO_ID = -1;

    public EmptyAddressInfo()
    {
        super(NO_ID, AddressType.UNKNOWN);
    }
    // ...
```

Nach kurzer Überlegung erkennt man allerdings, dass es unelegant ist, von der Klasse `AddressInfo` zu erben. Zum einen erzeugt dies eine ungewünschte Abhängigkeit von der Basisklasse. Zum anderen werden im Konstruktor der Basisklasse bereits Datenbankverbindungen aufgebaut, die für das Null-Objekt nicht benötigt werden. Damit sich die Klasse `EmptyAddressInfo` gemäß ihrer Basisklasse `AddressInfo` verhält, muss man zudem eine künstliche »ungültige« Id (`NO_ID`) sowie einen speziellen Wert `UNKNOWN` für den Aufzählungstyp `AddressType` definieren. Diese Lösung ist damit unnatürlich, kompliziert und wirkt sich dadurch negativ auf die Verständlichkeit aus. Es ist daher sinnvoll, ein gemeinsames Interface `AddressInfoIF` herauszufaktorieren und die Realisierung wie in Abbildung 22-3 gezeigt vorzunehmen.

Abbildung 22-3 `AddressInfoIF`*-Interface als Basis für Klasse und Null-Objekt*

Durch Einsatz des neu eingeführten Null-Objekts wird die Applikation toleranter gegenüber Zugriffen auf vorher uninitialisierte Variablen:

```
private static final AddressInfoIF EMPTY_ADDRESS = new EmptyAddressInfo();

Customer(final int newCustomerId)
{
    customerId = newCustomerId;

    homeAddress = EMPTY_ADDRESS;
    workAddress = EMPTY_ADDRESS;
    deliveryAddress = EMPTY_ADDRESS;
    // ...
```

Diese Lösung ist aber problematisch, da sie die Initialisierungsprobleme lediglich verlagert und Anwendungsfehler verschleiert: Es kommt im Fehlerfall nun nicht mehr zu Programmabstürzen, sondern zu merkwürdigem, fehlerhaftem Programmverhalten. Wir wollen eine sichere und elegante Lösung erstellen, dabei den Zugriff einfach gestalten und die Details der Erzeugung und Initialisierung verstecken. Die Grundlage dafür bildet das gerade herausfaktorierte Interface `AddressInfoIF`.

22.3.2 Lazy Initialization mithilfe des PROXY-Musters

Mithilfe des in Abschnitt 18.3.6 vorgestellten PROXY-Musters kann eine vorteilhafte Erweiterung in Form einer Abstraktionsschicht um die Lazy Initialization gebaut werden. Wenn die Initialisierung zentral im Proxy erfolgt, lassen sich Anwendungsfehler vermeiden. Das resultierende Klassendiagramm ist in Abbildung 22-4 dargestellt.

Der Proxy `LazyAddressInfo` führt die Lazy Initialization für die Applikation vollständig transparent durch. Dazu wird in jeder Realisierung einer im Interface `AddressInfoIF` deklarierten Zugriffsmethode zunächst die Methode `ensureAddressInfoExistence()` aufgerufen. Diese sorgt dafür, dass beim ersten Zugriff ein entsprechendes `AddressInfo`-Objekt erzeugt wird. Zur Vermeidung von Multithreading-Problemen ist diese Methode synchronisiert. Erfolgt niemals ein Me-

Abbildung 22-4 *Lazy Initialization mit der Klasse* `LazyAddressInfo`

thodenaufruf an das `AddressInfoIF`, so wird demzufolge auch kein `AddressInfo`-Objekt erzeugt. Eine mögliche Umsetzung sieht wie folgt aus:

```
public class LazyAddressInfo implements AddressInfoIF
{
    private AddressInfoIF     realAddressInfo;

    private final long        customerId;
    private final AddressType addressType;

    LazyAddressInfo(final long customerId, final AddressType addressType)
    {
        this.customerId = customerId;
        this.addressType = addressType;
    }

    // ...

    private synchronized void ensureAddressInfoExistence()
    {
        if (realAddressInfo == null)
        {
            realAddressInfo = new AddressInfo(customerId, addressType);
        }
    }

    public String getStreet()
    {
        ensureAddressInfoExistence();
        return realAddressInfo.getStreet();
    }

    public String getCity()
    {
        ensureAddressInfoExistence();
        return realAddressInfo.getCity();
    }

    // ...
```

Um den Proxy zu verwenden, sind in diesem Beispiel lediglich Anpassungen im Konstruktor durchzuführen:

```
Customer(final int newCustomerId)
{
    customerId = newCustomerId;

    homeAddress = new LazyAddressInfo(customerId, AddressType.HOME);
    workAddress = new LazyAddressInfo(customerId, AddressType.WORK);
    deliveryAddress = new LazyAddressInfo(customerId, AddressType.DELIVERY);
    // ...
    customerInfo = new CustomerInfo();
    contactInfo = new ContactInfo();
}

public AddressInfoIF getHomeAddress()
{
    return homeAddress;
}
```

Wenn alle Aufrufstellen ein Proxy-Objekt vom Typ `LazyAddressInfo` verwenden, anstatt auf das eigentliche Objekt zuzugreifen, führt erst ein Zugriff auf die Daten etwa über `getStreet()` zu einer Konstruktion eines `AddressInfo`-Objekts, was aber für den Aufrufer transparent bleibt. Auch müssen die Zugriffsmethoden keine Initialisierungen durchführen und können vereinfacht werden. Durch den Einsatz des Proxys können sämtliche `null`-Prüfungen und zugehörige Objektkonstruktionen aus dem Applikationscode entfernt werden. Dadurch sind alle den Sourcecode aufblähenden und fehleranfälligen Stellen überflüssig, die als Folge der ersten, naiven Realisierung der Lazy Initialization entstanden sind. Als weiterer positiver Nebeneffekt wurde durch das Einführen des Interface `AddressInfoIF` die Kopplung zur implementierenden Klasse gelöst. Da in diesem Fall im Interface sogar nur Leseoperationen auf unveränderliche Daten angeboten werden, sind die Adressinformationen zudem nach außen unveränderlich. Wie die Methode `getHomeAddress()` exemplarisch zeigt, ist nicht einmal eine Synchronisierung in der `Customer`-Klasse nötig. Auch darum kümmert sich der Proxy.

Empfehlungen

Nachdem wir unter Betrachtung möglicher Fallstricke und Lösungen die Lazy Initialization schrittweise realisiert haben, leiten wir folgende Hinweise ab:

1. Verwende Lazy Initialization immer mit Bedacht und Vorsicht und möglichst in Kombination mit dem PROXY-Muster (vgl. Abschnitt 18.3.6). Dessen Einsatz reduziert die Gefahr von uninitialisierten Attributen sowie von Synchronisationsproblemen bei Multithreading und erhöht die Lesbarkeit des Sourcecodes.
2. Verwende Lazy Initialization lediglich für spezielle, aufwendige Initialisierungen. Prüfe dazu, ob große Objekte tatsächlich immer bereits alle Daten im Voraus halten müssen oder ob Teile nicht erst bei Bedarf angelegt werden können.
3. Vermeide den Einsatz von Lazy Initialization für alle Komponenten, die mit ziemlicher Sicherheit verwendet werden oder deren Erzeugung nicht sehr teuer ist.

Fazit

Bis hierher haben wir lediglich die Vereinfachungen in der Applikation selbst durch Einführen eines Proxys betrachtet. Abschließend wollen wir kurz die Auswirkungen auf die Performance der Methode `printHomeAddress(int)` besprechen: Zur Ausgabe der Adresse eines Kunden muss immer noch eine Datenbankabfrage erfolgen und die `homeAddress` ermittelt werden. Allerdings werden als Folge der Lazy Initialization für die Attribute vom Typ `AddressInfo` bei der Konstruktion eines `Customer`-Objekts keine `AddressInfo`-Objekte mehr erzeugt, sondern dies geschieht erst bei Bedarf während des Zugriffs über `get()`-Methoden. Da in der eingangs als laufzeitproblematisch vorgestellten Methode `printHomeAddress(int)` die beiden Attribute `workAddress` und `deliveryAddress` nicht benötigt werden, spart man mit dieser Technik folglich das Auslesen jener Werte aus der Datenbank. Der Performance-Gewinn liegt demnach in zwei eingesparten Datenbankzugriffen.

Wie bereits angedeutet und im folgenden Abschnitt weiter vertieft, besitzen Zugriffe auf externe Systeme häufig deutlich negative Auswirkungen auf die Performance. Überflüssige Aufrufe in diese Systeme zu vermeiden, kann somit die Performance spürbar verbessern.

22.4 Optimierungen am Beispiel

Anhand eines Optimierungsbeispiels möchte ich den Einsatz des in Abschnitt 22.1.4 erwähnten Tools `VisualVM` motivieren, um mögliche Performance-Probleme aufzuspüren. Zudem bekommen Sie ein Gefühl für Lösungsansätze und Umsetzungen von Optimierungen. Dieses Beispiel basiert auf einer von mir vor langer Zeit (JDK 1.2) tatsächlich durchgeführten Optimierung einer Tabellenkomponente. Diese enthielt neben der textuellen Darstellung von Einträgen verschiedene Grafiken, die zur Anzeige aus Dateien nachgeladen wurden.

Problemkontext

Damals war das Zeichnen und Scrollen mit bis zu etwa 100 Tabelleneinträgen ausreichend schnell. Als jedoch Tests mit wenigen Tausend Einträgen erfolgten, wurde die Benutzeroberfläche schrecklich langsam und man konnte nur noch stockend durch die Liste navigieren. Heutzutage sind sowohl die Rechner als auch die JVM wesentlich schneller geworden. Damit wir überhaupt ein sinnvolles Performance-Tuning durchführen können, müssen wir die Randbedingungen enorm verschärfen.

Betrachten wir eine Beispielapplikation, die eine Tabelle mit 50.000 Einträgen von `Person`-Objekten darstellt. Deren Speicherung erfolgt in einem Tabellenmodell `PersonTableModel` in einer `LinkedList<Person>`. Ein spezieller Renderer `SimpleImageTableCellRenderer` lädt die anzuzeigenden Grafiken bei jedem Zeichenvorgang aus dem Dateisystem. Folgendes Listing zeigt den relevanten Auszug aus der Implementierung:

```
public final class ListOptimizationExample extends JFrame
{
   private final PersonTableModel personTableModel;
   private final JTable           personTable;

   public ListOptimizationExample(final int personCount,
                                  final ListType listType,
                                  final TableCellRenderer cellRenderer)
   {
       setTitle("ListOptimizationExample Persons: " + personCount + " / Using: "
        + listType + " / Renderer: " + cellRenderer.getClass().getSimpleName());

       personTableModel = new PersonTableModel(listType, personCount);
       personTable = new JTable(personTableModel);
       personTable.setRowHeight(64);
       personTable.setDefaultRenderer(ImageIcon.class, cellRenderer);

       getContentPane().add(new JScrollPane(personTable));

       setDefaultCloseOperation(DISPOSE_ON_CLOSE);
   }

   public static void main(final String[] args)
   {
       final JFrame demoframe = new ListOptimizationExample(50_000,
                      ListType.LinkedList, new SimpleImageTableCellRenderer());

       demoframe.setSize(700, 500);
       demoframe.setVisible(true);
   }
   // ...
```

Listing 22.1 *Ausführbar als* '**LISTOPTIMIZATIONEXAMPLE**'

Führt man das Programm LISTOPTIMIZATIONEXAMPLE aus, so zeigt sich, dass das Scrolling etwas träge ist. Auch beim Vergrößern des Fensters kommt es zu Nachzieheffekten. Die Applikation sieht in etwa wie in Abbildung 22-5 aus.

Abbildung 22-5 *Optimierungsbeispiel: Tabelle mit 50.000 Einträgen*

Analyse

Zuerst wollen wir durch Messungen diejenigen Programmbereiche ermitteln, die sich negativ auf die Performance auswirken. Dazu starten wir das Tool `VisualVM` und die obige Applikation. Die Profiling-Messungen zeigen zwei Hotspots: Zum einen ist das Laden der Grafiken über `loadTileImage(int)` teuer und zum anderen sehen wir, dass relativ viel Zeit für die Zugriffe auf das Datenmodell über `getValueAt(int, int)` verbraucht wird. Abbildung 22-6 zeigt die zugehörige Profiling-Messung.

Abbildung 22-6 *Profiling einer* `LinkedList` *mit 50.000 Einträgen*

Zudem erkennen wir beim Betrachten der CPU-Auslastung immer wieder Spitzen bis ca. 20 % – bei älteren JDKs auch bis 50 %. Außerdem sind diverse Speicherbereinigungen durch die Garbage-Collection zu sehen. Abbildung 22-7 stellt dies dar.

Abbildung 22-7 *Profiling einer* `LinkedList`*: Das CPU-Profil*

Zu den Belastungsspitzen kommt es beim Scrollen durch die Liste. Das lässt befürchten, dass bei der Speicherung von wesentlich mehr Datensätzen die Performance weiter einbricht, die Applikation also schlecht skaliert.

I/O-bound-Optimierungen

Durch die Profiling-Messungen erkennen wir als vorrangigen Hotspot die Methode `loadTileImage(int)` der Klasse `SimpleImageTableCellRenderer`. Wir beginnen dort mit der Analyse und betrachten die Methode genauer:

```
public ImageIcon loadTileImage(final int i)
{
   final int imageNo = (i % getNumOfTiles());
   final File imageFile = new File(PATH_TO_IMAGES, tileFileNames[imageNo]);
   try
   {
      return new ImageIcon(ImageIO.read(imageFile)); // Dateisystemzugriff
   }
   catch (final IOException e)
   {
      return EMPTY_IMAGE; // FALLBACK: EMPTY IMAGE
   }
}
```

Neben der eigentlichen Methode sollten wir auch die Aufrufhierarchie kritisch untersuchen: `getTableCellRendererComponent()` ⇒ `getTileImage()` ⇒ `loadTileImage()`. Schauen wir uns die Implementierung der Klasse `SimpleImageTableCellRenderer` an:

```
public class SimpleImageTableCellRenderer extends DefaultTableCellRenderer
{
    private final String   PATH_TO_IMAGES = "src/main/resources/tiles/";
    private final String[] tileFileNames  = { "tile_gras_1.jpg",
                            "tile_gras_2.jpg", "tile_rock_1.jpg",
                            "tile_rock_2.jpg", "tile_water.jpg" };

    private final ImageIcon EMPTY_IMAGE = new ImageIcon(new BufferedImage(1, 1,
                                                  BufferedImage.TYPE_INT_RGB));

    public SimpleImageTableCellRenderer()
    {
        setHorizontalTextPosition(JLabel.RIGHT);
        setHorizontalAlignment(JLabel.LEFT);
    }

    public Component getTableCellRendererComponent(JTable table, Object value,
          boolean isSelected, boolean hasFocus, int row, int column)
    {
        super.getTableCellRendererComponent(table, null, isSelected, false,
                                            row, column);
        if (column >= 2)
        {
            // Ermitteln der Bilder
            setIcon(getTileImage(row + column - 2));
        }
        return this;
    }

    public ImageIcon getTileImage(final int i)
    {
        // Direkte Weiterleitung an loadTileImage()
        return loadTileImage(i);
    }
```

```
    public ImageIcon loadTileImage(final int i)
    {
        final int imageNo = (i % getNumOfTiles());
        final File imageFile = new File(PATH_TO_IMAGES, tileFileNames[imageNo]);
        try
        {
            // Dateisystemzugriff
            return new ImageIcon(ImageIO.read(imageFile));
        }
        catch (final IOException e)
        {
            return EMPTY_IMAGE; // FALLBACK: EMPTY IMAGE
        }
    }

    public final int getNumOfTiles()
    {
        return tileFileNames.length;
    }
}
```

Eine Analyse dieser Klasse ergibt, dass bei jedem Zeichnen der Tabelle mit diesem Renderer durch die Aufrufe von `getTileImage(int)` und danach `loadTileImage(int)` Zugriffe auf das Dateisystem erfolgen. Das ist ein Ansatzpunkt – insbesondere weil lediglich eine feste Anzahl an Grafiken verwendet wird. Folglich kann als Optimierung das Laden der Grafiken besser einmalig bei der Konstruktion des Renderers erfolgen. Ein Caching der Bilder erlaubt, anschließend performant darauf zuzugreifen. Dazu erweitern wir die Renderer-Klasse dahingehend, dass eine Speicherung der Grafiken im Attribut `tileIcons` in Form eines Arrays von `ImageIcon`-Objekten erfolgt. Es entsteht die folgende, optimierte Renderer-Klasse `CachedImageTableCellRenderer`, die das Scrollen deutlich flüssiger macht.

```
public final class CachedImageTableCellRenderer extends
                                            SimpleImageTableCellRenderer
{
    private final ImageIcon[] tileIcons;

    public CachedImageTableCellRenderer()
    {
        final int numOfBackgrounds = getNumOfTiles();
        tileIcons = new ImageIcon[numOfBackgrounds];

        for (int i = 0; i < numOfBackgrounds; i++)
        {
            tileIcons[i] = loadTileImage(i);
        }
    }

    public ImageIcon getTileImage(final int i)
    {
        final int index = i % getNumOfTiles();
        return tileIcons[index];
    }
}
```

Ein Start des nachfolgenden Programms LISTOPTIMIZATIONEXAMPLEIOIMPROVED zeigt dies.

```
public static void main(final String[] args)
{
    final JFrame demoframe = new ListOptimizationExample(50_000,
                                 ListType.LinkedList,
                                 new CachedImageTableCellRenderer());
    demoframe.setSize(700, 500);
    demoframe.setVisible(true);
}
```

Listing 22.2 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEIOIMPROVED'**

Möchte man weitere Optimierungen vornehmen, so ist zu bedenken, dass sich als Folge von Optimierungen sehr häufig das Laufzeitprofil der Applikation verändert und es zu einer Verschiebung von Hotspots kommt: Zuvor erkannte Hotspots können an Bedeutung verlieren und andere Programmteile können zu Hotspots werden. Es ist daher wichtig, erneute Messungen durchzuführen und nicht direkt mit der Optimierung des zuvor ermittelten zweiten Hotspots – hier der Methode `getValueAt(int, int)` – fortzufahren. Eine Folgemessung ergibt die in Abbildung 22-8 dargestellten Hotspots und zeigt, dass in diesem Fall die Methode `getValueAt(int, int)` auch weiterhin Performance-kritisch ist und sich nun sogar an erster Stelle befindet.

Abbildung 22-8 *Profiling einer* `LinkedList` *mit verbessertem I/O (50.000 Einträge)*

Die in der ersten Messung als primär problematisch erkannte Methode `loadTileImage(int)` spielt nach der Optimierung für die Performance praktisch keine Rolle mehr. Tatsächlich ist sie diesbezüglich sogar derart unbedeutend geworden, dass sie nicht einmal mehr in der abgebildeten Hotspot-Liste auftaucht.

Memory-bound-Optimierungen

Wie nun auch durch Messung nachgewiesen, stellt die Methode `getValueAt(int, int)` immer noch den limitierenden Faktor dar. Allerdings sind die gefühlten Auswirkungen auf die Performance bei 50.000 Datensätzen relativ gering: Das Scrolling könnte noch minimal flüssiger erfolgen. Erst wenn man die Anzahl der Datensätze wie im folgenden Listing verdreifacht, also auf 150.000 `Person`-Objekte erhöht, wird das Scrolling zäher.

```
public static void main(final String[] args)
{
    final JFrame demoframe = new ListOptimizationExample(150_000, // 150.000
                                 ListType.LinkedList,
                                 new CachedImageTableCellRenderer());

    demoframe.setSize(700, 500);
    demoframe.setVisible(true);
}
```

Listing 22.3 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEIOIMPROVED2'**

Um das Antwortzeitverhalten beim Scrolling noch weiter zu verbessern, sollten wir die Methode `getValueAt(int, int)` analysieren. Schauen wir auf deren Realisierung:

```
public Object getValueAt(final int rowIndex, final int columnIndex)
{
    final Person person = persons.get(rowIndex);

    if (columnIndex == 0)
        return person.getName();
    if (columnIndex == 1)
        return person.getGender();

    return ""; // "" => Kein Text für Bilder
}
```

Das Attribut `persons` ist eine `List<Person>` und verwendet hier eine `LinkedList<Person>`. Auf den ersten Blick ist daran nichts auszusetzen, aber die Ausführungszeit ist nicht zufriedenstellend. Der Grund ist einfach: Eine `LinkedList<E>` ist für indizierte Zugriffe nicht optimal. Aufgrund der Performance-Betrachtung verschiedener Listenimplementierungen in Abschnitt 22.2.1 wissen wir, dass eine `LinkedList<E>` zum Zugriff auf Elemente an einer beliebigen Position immer eine Iteration durch alle Vorgänger benötigt. Dadurch entstehen Kosten von $O(n)$. Beim Zeichnen der Tabelle fallen diese Kosten für jede sichtbare Zeile erneut an. Verwendet man statt einer `LinkedList<E>` eine `ArrayList<E>`, so sollte sich dies extrem positiv auswirken, da Zugriffe nur noch Kosten von $O(1)$ verursachen. Überprüfen wir diese Vermutung, indem wir eine `ArrayList<Person>` zur Datenspeicherung verwenden:

```
public static void main(final String[] args)
{
    final JFrame demoframe = new ListOptimizationExample(150_000,
                                 ListType.ArrayList, // ArrayList
                                 new CachedImageTableCellRenderer());

    demoframe.setSize(700, 500);
    demoframe.setVisible(true);
}
```

Listing 22.4 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEIOANDDSIMPROVED'**

Durch diese Modifikation ergibt sich die erwartete Verbesserung: Die CPU-Belastung bleibt selbst bei vielen Scroll-Aktionen immer unter rund 10 % und außerdem ist kein wirklicher Hotspot mehr feststellbar. Selbst bei über 100.000 Aufrufen der jeweiligen Methoden bleiben die Ausführungszeiten unter 1 Sekunde (vgl. Abbildung 22-9).

Profiling results

Results: View: Collected data: Snapshot

Name	Self Time (CPU)		Total Time (CPU)		Invocations
ch22_optimierungen.examples.SimpleImageTableCellRenderer.**getTableCellRendererComponent**	110 ms	(33.3%)	193 ms	(43.9%)	106'467
ch22_optimierungen.examples.ListOptimizationExample$PersonTableModel.**getColumnClass** (int)	58.1 ms	(17.6%)	58.1 ms	(13.2%)	149'053
ch22_optimierungen.examples.CachedImageTableCellRenderer.**getTileImage** (int)	56.5 ms	(17.1%)	83.4 ms	(18.9%)	106'467
ch22_optimierungen.examples.ListOptimizationExample$PersonTableModel.**getValueAt** (int, int)	44.3 ms	(13.4%)	44.3 ms	(10.1%)	149'053
ch22_optimierungen.examples.ListOptimizationExample$PersonTableModel.**getRowCount** ()	34.1 ms	(10.4%)	34.1 ms	(7.8%)	58'730
ch22_optimierungen.examples.SimpleImageTableCellRenderer.**getNumOfTiles** ()	26.8 ms	(8.1%)	26.8 ms	(6.1%)	106'467

Abbildung 22-9 *Profiling einer* `ArrayList` *mit 150.000 Einträgen*

Die erzielte Verbesserung wird noch deutlicher, wenn man nun die Anzahl der Datensätze auf eine Million erhöht. Die Performance ist nicht merklich schlechter als zuvor für 150.000 Einträge. Überprüfen Sie es selbst, indem Sie folgendes Programm LISTOPTIMIZATIONEXAMPLEONEMILLION ausführen:

```
public static void main(final String[] args)
{
   final JFrame demoframe = new ListOptimizationExample(1_000_000, // 1.000.000
                                ListType.ArrayList,
                                new CachedImageTableCellRenderer());

   demoframe.setSize(700, 500);
   demoframe.setVisible(true);
}
```

Listing 22.5 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEONEMILLION'**

Fazit

Anhand eines Beispiels habe ich das grundsätzliche Vorgehen bei Optimierungen vorgestellt: Man beginnt mit Messungen, um die tatsächlich kritischen Bereiche zu bestimmen. Anschließend werden diese einzeln nacheinander bearbeitet. Durch nachfolgende Messungen stellt man dann sicher, dass es wirklich zu Verbesserungen gekommen ist. Dieses Vorgehen wiederholt man, bis eine zufriedenstellende Performance erreicht ist.

22.5 I/O-bound-Optimierungen

In diesem Abschnitt betrachten wir Möglichkeiten, die Ein- und Ausgabe zu beschleunigen. Interagiert ein Programm mit dem Dateisystem oder einem Netzwerk, so sind signifikante Performance-Steigerungen zu erreichen, wenn feingranulare Zugriffe vermieden oder Daten gepuffert werden. Im Folgenden wollen wir diese und einige weitere Techniken und ihre Auswirkungen auf die Performance genauer kennenlernen.

22.5.1 Technik – Wahl passender Strategien

Bei der Interaktion mit einem Netzwerk kann man sowohl durch Pufferung als auch durch eine geschickte Wahl der Repräsentationsform der zu übertragenden Daten die Performance verbessern. Beispiele dafür sind der Einsatz von Dekorierern zur Pufferung sowie die Anpassung des Serialisierungsvorgangs.

Optimierungen beim Einsatz von Streams

Das Problem beim direkten (naiven) Einsatz von Streams ist, dass sowohl das Lesen als auch das Schreiben ohne Pufferung erfolgt, d. h., es wird genau ein Byte zur Zeit verarbeitet. Gleiches gilt analog für die Subklassen von `Reader` und `Writer`, die zeichenbasiert arbeiten. Eine Datei byte- bzw. zeichenweise zu lesen ist allerdings nicht sehr performant. Durch Einsatz von Dekorierern zur Pufferung werden jeweils größere Datenblöcke gelesen und geschrieben. Welch enorme Geschwindigkeitssteigerungen sich dadurch erzielen lassen, werden wir am Beispiel des Kopierens von Dateien kennenlernen.

Folgende Varianten zum Kopieren wurden untersucht:

- **Einsatz einer byteweisen Kopie (A)** – Nachfolgend wird die Hilfsmethode `copyBytewise(InputStream, OutputStream)` entwickelt, die Daten Byte für Byte kopiert, indem zunächst ein Byte gelesen und dann geschrieben wird:

```
public static void copyBytewise(final InputStream is, final OutputStream os)
       throws IOException
{
    int data = -1;
    while ((data = is.read()) != -1)
    {
        os.write(data);
    }
    os.flush();
}
```

- **Einsatz eines gepufferten Streams (B)** – Eine Pufferung der Zugriffe erreicht man durch Ummantelung mit den Dekorierern `BufferedInputStream` bzw. `BufferedOutputStream`. Zum Resource Handling nutzen wir ARM, wodurch keine direkten Aufrufe von `close()`-Methoden der Streams erforderlich sind:

```
try (final InputStream bufferedIn = new BufferedInputStream(inStream),
     final OutputStream bufferedOut = new BufferedOutputStream(outStream))
{
    copyByteWise(bufferedIn, bufferedOut);
}
```

- **Einsatz eines eigenen Puffers (C)** – Statt die gepufferten Streams aus dem JDK zu benutzen, kann man eine Pufferung selbst implementieren. Diese Funktionalität kapselt man zweckmäßig innerhalb einer Methode. Dies wird durch die Methode `copyOwnBuffering(InputStream, OutputStream)` wie folgt realisiert:

```
public static void copyOwnBuffering(final InputStream is,
                                    final OutputStream os)
                   throws IOException
{
    final byte[] buffer = new byte[BUFFER_SIZE];
    int length = -1;
    while ((length = is.read(buffer, 0, BUFFER_SIZE)) != -1)
    {
        os.write(buffer, 0, length);
    }
    os.flush();
}
```

- **Einsatz der Klasse `FileChannel` (D)** – Zum Kopieren nutzen wir als letzte Variante die Klasse `FileChannel` und deren Methode `transferTo(long, long, java.nio.channels.WritableByteChannel)`, die direkt auf Betriebssystemebene arbeitet:

```
final FileChannel sourceChannel = inStream.getChannel();
final FileChannel destChannel = outStream.getChannel();

sourceChannel.transferTo(0, sourceChannel.size(), destChannel);
```

- **Einsatz der Methode `transferTo()` (E)** – Seit Java 9 existiert die Methode `transferTo()` in der Klasse `FileInputStream`, mit der sich die Kopieraktion kurz und prägnant schreiben lässt:

```
inStream.transferTo(outStream);
```

Als Eingabe dienen zwei PDF-Dateien unterschiedlicher Größe von ca. 729 KB und ca. 13,6 MB. In der folgenden Tabelle 22-4 sind die Ergebnisse verschiedener Performance-Messungen auf einem iMac (Quad-Core i7 mit 4 GHz) mit JDK 14.0.1 zusammengefasst – der Messwert von 0 ms wurde für die Berechnungen auf 1 ms aufgerundet und mit einem * markiert.

Tabelle 22-4 *Performance-Gewinn durch Pufferung beim Kopieren*

	Variante					Faktor				
Größe	**A**	**B**	**C**	**D**	**E**	**A-B**	**A-C**	**B-C**	**C-D**	**D-E**
729 KB	2.346 ms	34 ms	1 ms	4 ms	1* ms	69	2.346	34	0.25	4
13,6 MB	46.044 ms	70 ms	21 ms	9 ms	9 ms	657	2.192	4.3	2,3	1

Wie man leicht sieht, ist bereits durch den Einsatz von Pufferung (Variante B) ein enormer Performance-Gewinn zu erzielen, der etwa zwischen Faktor 70 bis 650 im Vergleich zum ungepufferten Kopieren (Variante A) liegt. Eine deutliche Beschleunigung um zusätzlich etwa Faktor 3 bis 35 kann man nochmals erzielen, wenn man die Pufferung selbst durchführt (Variante C). Etwas verwunderlich ist, dass der Einsatz von Channels (Variante D) mitunter langsamer als der Einsatz der zuvor vorgestellten gepufferten Variante ist, obwohl hier direkt auf Betriebssystemebene gearbeitet wird. Hieran sieht man, dass sich der Aufwand für Spezialanpassungen nicht in jedem Fall lohnt. Hendrik Schreiber hat diesen Effekt bereits 2002 für das JDK 1.4 in seinem Buch »Performant Java programmieren« [64] beschrieben. Allerdings wurden insbesondere mit JDK 8 deutliche Verbesserungen für Channels (Variante D) erzielt und diese ziehen nunmehr sehr häufig mit der Geschwindigkeit eigener Pufferungen gleich bzw. übertreffen sie teilweise. Tatsächlich ist die am besten lesbare Variante E mit `transferTo()` auch die performanteste.

Kompression und ZIP-Archive

Zur Optimierung von I/O ist es neben der Pufferung eine gute Idee, die Setup- und Transferzeiten zu reduzieren, indem man zu übertragende Daten oder Dateien in einem ZIP-Archiv zusammenfasst. Jeder kennt es vom Kopieren von Dateien auf eine externe Festplatte oder auf einen USB-Stick: Im Dateisystem ist das Kopieren einer größeren Datei viel schneller als die Übertragung vieler kleinerer Dateien. Es lohnt sich in der Regel, zunächst diese Dateien in einem ZIP-Archiv zusammenzufassen, selbst wenn man dabei keine Kompression vornimmt. Es sind dann zwar zwei zusätzliche Schritte (ZIP-Archiv erzeugen und auslesen) notwendig, aber trotzdem kann diese Variante für viele kleine Dateien deutlich günstiger sein als einzelne Dateitransfers. Diese Ideen liegen auch der Speicherung in JAR-Dateien zugrunde.

Bei Übertragungen über ein Netzwerk kann man zusätzlich die Komprimierung einschalten. Im Normalfall sollten die Einsparungen durch die weniger zu übertragenden Bytes die Zeiten zum Komprimieren und Entpacken mehr als ausgleichen. Dafür können die im JDK im Package `java.util.zip` definierten Klassen `ZipInputStream` und `ZipOutputStream` genutzt werden, um einen Stream um genau diese ZIP-Funktionalität zu erweitern.

Optimierungen der Serialisierung

Bei der Serialisierung werden Objekte in einen `ObjectOutputStream` geschrieben bzw. aus einem `ObjectInputStream` eingelesen. Die dazu notwendige Transformation von Objekten in eine Folge von Bytes bzw. umgekehrt erfolgt automatisch, wenn eine Klasse das Interface `Serializable` erfüllt (vgl. Abschnitt 12.3).

Obwohl diese Automatik für viele Fälle praktisch ist, kann sie sich aber in einigen Situationen negativ auf die Performance auswirken. Weil die dahinterliegende Komplexität vor dem Aufrufer versteckt wird, bleibt auch verborgen, dass immer der gesamte Objektgraph gespeichert bzw. erzeugt wird. Dieser kann beim Einsatz von Vererbung und vielen aggregierten Objekten durchaus komplex sein, wodurch auch der Aufwand zur Serialisierung steigt. Anders formuliert: Wenn Serialisierung eingesetzt wird, ist es sinnvoll, zu überlegen, ob wirklich alle Zustandsinformationen gespeichert werden müssen und welche Informationen bei einem späteren Einlesen eventuell ohne Probleme berechnet oder anderweitig ermittelt werden können.

Betrachten wir dies konkret für Swing-GUI-Komponenten. Deren Serialisierung ist im Vergleich zu anderen, normalen Objekten sehr teuer: GUI-Komponenten benötigen mehrere KB, da sie ihre komplette AWT- und Swing-Ableitungshierarchie enthalten. Häufig ist aber eine Serialisierung dieser Daten nicht unbedingt notwendig bzw. sogar nicht sinnvoll. Für ein Label wäre es in vielen Fällen ausreichend, nur den Text zu speichern. Daraus leiten wir ab, dass es oftmals praktischer und effizienter ist, nur die Datenmodelle zu speichern. Werden GUI-Komponenten in eigenen, zu serialisierenden Objekten aggregiert, so sollten diese explizit vom Vorgang der Serialisierung per Schlüsselwort `transient` ausgeschlossen werden. Als Folge sind allerdings spezielle Anpassungen beim Einlesen durchzuführen. Details dazu beschreibt Abschnitt 12.3.2.

22.5.2 Technik – Caching und Pooling

Um häufig benötigte Daten in einem Speicher mit kurzen Zugriffszeiten vorrätig zu halten, setzt man Caching ein. Dadurch lassen sich teilweise enorme Verbesserungen der Ausführungszeit erreichen. Gleiches gilt für eine gepufferte Verarbeitung von Dateien. Beide Themen wurden bereits ausführlich betrachtet (vgl. Abschnitt 22.4 für das Beispiel der Grafikdateien). Daher erfolgt in diesem Abschnitt keine weitere Beschreibung des Cachings für die Ein- und Ausgabe.

22.5.3 Technik – Vermeidung unnötiger Aktionen

Wie in Abschnitt 22.3 detailliert am Beispiel der Lazy Initialization vorgestellt, kann das Vermeiden nicht benötigter Aktionen sowohl die Ausführungszeit als auch den Speicherbedarf einer Applikation reduzieren. Beispiele sind das Prüfen des Log-Levels vor der Aufbereitung aufwendiger Log-Ausgaben sowie die Reduktion von Remote Calls in verteilten Applikationen.

Optimierungen beim Logging

Verschiedene Techniken zur Optimierung beim Logging wurden bereits in Abschnitt 9.4.2 vorgestellt. Hier wird daher nur die grundsätzliche Idee noch mal kurz zusammengefasst: Man versucht, das Aufbereiten und das Schreiben von Log-Ausgaben möglichst zu reduzieren. Dazu gibt es grundsätzlich zwei wichtige Varianten: Zum einen kann man mit formatierten Log-Ausgaben und Lambdas arbeiten und zum anderen empfiehlt es sich, vor komplexeren oder umfangreichen Log-Ausgaben zunächst das Log-Level zu prüfen, bevor Log-Ausgaben aufbereitet werden und erfolgen. Somit vermeidet man einerseits eine Erzeugung der Texte zur Log-Ausgabe, wodurch weniger temporäre Stringobjekte entstehen. Andererseits sorgen sinnvoll gewählte Log-Level dafür, dass seltener in eine Log-Datei geschrieben wird. Dies kommt zudem der Lesbarkeit und Auswertbarkeit der entstehenden Log-Dateien zugute.

Reduktion von Remote Calls

Im Netzwerk und in verteilten Applikationen kann man durch Unachtsamkeit leicht Performance-Probleme auslösen. Ich werde im Folgenden kurz vorstellen, welche Fallstricke auf dem Weg zu einem gelungenen und gleichzeitig performanten API für eine verteilte Applikation lauern.

Wenn eine Interaktion mit anderen Systemen erforderlich ist, so müssen die Klassen, die vom Netzwerk aus angesprochen werden sollen, dazu Remote-Interfaces anbieten. Wenn man diese externen Schnittstellen einer Applikation ähnlich zu den Schnittstellen innerhalb eines Programms definiert, so entstehen viele feingranulare Methodenaufrufe, die alle als einzelne Kommandos über das Netzwerk versendet werden müssen. Als Folge wird für jeden Methodenaufruf mindestens ein Netzwerktelegramm erzeugt. Abbildung 22-10 deutet dies an.

Abbildung 22-10 *Remote Calls*

Analogie: Optimierung in der Realität

Mögliche Performance-Probleme von Remote Calls lassen sich sehr anschaulich an einem Beispiel aus der Realität verdeutlichen. Man stelle sich vor, es wäre ein Fußballabend mit einigen Freunden in vollem Gange. Irgendwann möchte jeder Gast natürlich auch etwas trinken. Ziemlich aufwendig wäre es, wenn man bei jeder Nachfrage nach einem gekühlten Getränk immer wieder zum Supermarkt fahren würde, um dieses dort zu kaufen. In der Realität ist sofort einsichtig, dass diese Verhaltensweise extrem unsinnig und zeitlich zudem ungünstig ist. Selbstverständlich fährt man einmal zum Supermarkt und deckt sich mit einer Menge an Getränken ein. So offensichtlich diese Vorgehensweise in der Realität auch ist, so ist eine Umsetzung in entsprechende Remote Calls nicht ganz so trivial.

Folgende Abhilfemaßnahmen bieten sich an:

- **Aufrufe zusammenfassen** – Nachrichten werden zu einer größeren Nachricht zusammengefasst.
- **Daten zusammenfassen** – Daten werden zu einem größeren Datenpaket zusammengefasst. Dies kann gemäß dem Muster VALUE OBJECT (vgl. Abschnitt 3.4.5) in seiner Ausprägung als DATA TRANSFER OBJECT (DTO) geschehen.

Aufrufe zusammenfassen Um die Anzahl von Remote Calls zu reduzieren, kann man Methodenaufrufe zusammenfassen. Dies geht besonders gut für `set()`-Methoden, denen man dann statt eines Parameters mehrere übergibt:

```
interface SimpleWriteIF
{
    void setValue1(int value1)
    void setValue2(int value2)
    void setValue3(int value3)
}

interface OptimizedRemoteWriteIF
{
    void setValues(int value1, int value2, int value3)
}
```

Bei `get()`-Methoden ist diese Technik nicht ganz so einfach einzusetzen, da sich nur ein Wert zurückgeben lässt – ein Abhilfe können die mit Java 14 eingeführten `records` bringen (vgl. Abschnitt 14.1.3).

Daten zusammenfassen Man kann ein Data Transfer Object (DTO) erstellen, das die Ergebnisse mehrerer `get()`-Aufrufe bündelt. Im einfachsten Fall speichert man dort die Attribute einzeln. Eine alternative Technik besteht darin, eine `HashMap<K,V>` mit benannten Attributen zu nutzen. Für beide Arten der Umsetzung werden dann mehrere feingranulare Aufrufe in einen Aufruf mit einem DTO umgewandelt und dann die entsprechenden Antworten generiert, wie dies in Abbildung 22-11 zu sehen ist.

Abbildung 22-11 *Optimierung von Remote Calls*

Zum Transport über ein Netzwerk sollte das DTO möglichst leichtgewichtig sein, indem bevorzugt primitive Typen als Datenattribute Anwendung finden. Abschnitt 22.6.4 beschreibt die Gründe genauer.

Auswirkungen Fasst man `set()`-Aufrufe zusammen, so reduziert man n Remote Calls auf lediglich einen Aufruf. Allerdings wird dadurch das Interface recht stark durch technische Gegebenheiten beeinflusst. In solchen Fällen bietet sich eine Abstraktionsebene vor dem eigentlichen Netzwerkzugriff an. Dies könnte durch ein Proxy-Objekt gemäß dem PROXY-Muster (vgl. Abschnitt 18.3.6) realisiert werden.

Fasst man `get()`-Aufrufe zusammen, so kann man auch hier wieder n Remote Calls auf lediglich einen Aufruf reduzieren. Allerdings muss dann ein DTO eingesetzt und während der Kommunikation gefüllt werden. Dazu sind nun statt n `get()`-Aufrufen die doppelte Anzahl notwendig: zum einen n Aufrufe aufseiten des Servers zum Befüllen des DTO, zum anderen bis zu n weitere Aufrufe aufseiten des Klienten zum Ermitteln der Werte. Da es sich in beiden Fällen um vergleichsweise günstige lokale Aufrufe handelt, ist dies ein geringer Preis für die eingesparten Remote Calls.

22.6 Memory-bound-Optimierungen

Die Größe des Hauptspeichers, der einer Applikation zur Verfügung steht, kann signifikanten Einfluss auf die Performance haben. Dies gilt vor allem für Systeme mit wenig Arbeitsspeicher oder für Applikationen, die mit riesigen Datenmengen arbeiten. Die Größe des ausführbaren Programms und dessen Speicherverbrauch zur Laufzeit sind daher weitere wichtige Aspekte bei Optimierungen.

Abschnitt 22.6.1 geht kurz auf die Auswirkungen von Größe und Nutzung des Hauptspeichers ein und zeigt einige Möglichkeiten, die Garbage Collection zu optimieren. Abschnitt 22.6.2 beschreibt die Vorteile, die sich ergeben, wenn man häufig benutzte Objekte zwischenspeichert (***Objekt-Caching***) bzw. alternativ bereits erzeugte Objekte recycelt und mit geänderter Parameterbelegung wiederverwendet (***Objekt-Pooling***). Konkret wird dies für die Stringverwaltung von Java genutzt. Abschnitt 22.6.3 stellt weitere mögliche Optimierungspotenziale bei Strings vor. Abschließend wird in Abschnitt 22.6.4 erläutert, warum man für Attribute in der Regel primitive Datentypen den entsprechenden Wrapper-Klassen vorziehen sollte.

22.6.1 Technik – Wahl passender Strategien

Größe und Nutzung des Hauptspeichers

Steigt der von einer Applikation verbrauchte Speicher ständig bis nahe an die der JVM zugewiesene Grenze, so kann sich dies äußerst negativ auf die Programmantwortzeit auswirken. Wurde die maximale Größe des benötigten Speichers initial falsch eingeschätzt und ist deswegen das Programm infolge häufiger Garbage Collections träge, lässt sich eine Beschleunigung erreichen, indem man per Aufrufparameter `-Xmx` beim Programmstart mehr Speicher bereitstellt. Dieses Vorgehen kann hilfreich sein, stellt jedoch keine allgemeingültige Lösung dar. Man vermeidet damit lediglich, sich mit vermeintlichen Performance-Problemen in Programmteilen zu beschäftigen, die nur zufällig durch schlecht gewählte Rahmenbedingungen entstanden sind. Nach dieser, in der Regel einmalig wirksamen Optimierungsmaßnahme kann man sich dann tatsächlich vorliegenden Performance-Problemen widmen. Ist eine Applikation aber unersättlich bezüglich des Speichers, so sollte man nicht einfach iterativ die Speichergröße anpassen, sondern die verursachenden Stellen aufspüren und beheben. Einige dafür passende Techniken wurden bereits in Abschnitt 22.2 als Hinweise zur Wahl geeigneter Datenstrukturen und Rückgabewerte bei großen Datenmengen diskutiert.

Mythos: Viel hilft viel

Um eine gute Performance zu erzielen, ist es in der Regel nicht sinnvoll, der JVM einfach möglichst viel Speicher zuzuteilen. Dies führt teilweise nur dazu, dass die Garbage Collection mehr Zeit für die Speicherbereinigung benötigt. Ratsam ist es, zuvor den tatsächlichen Speicherverbrauch zu analysieren.

Tipp: Auswirkungen von Threads

Wenn ein Programm sehr viele Threads zur Verarbeitung mehrerer unabhängiger Aufgaben startet, so wirkt sich dies auf den Speicherbedarf aus. Threads werden normalerweise auf Betriebssystem-Threads abgebildet und sind dadurch in ihrer Anzahl begrenzt. Zudem benötigen Threads weitere Systemressourcen sowie eigenen Speicher, etwa zur Bereitstellung eines Stacks.

Bei Tausenden zu startenden Threads kann man einer Applikation dann weniger Speicher über die Option `-Xmx` zur Verfügung stellen. Es gilt: Je weniger Threads, desto größer kann der Wert von `-Xmx` gewählt werden und umgekehrt. Die Ursache liegt darin, dass Threads und genutzte Stacks nicht im Speicherbereich der JVM, sondern im Speicher des Betriebssystems verwaltet werden.

Optimierungen der Garbage Collection

Java befreit den Entwickler mit der automatischen Garbage Collection von der manuellen Freigabe des Speichers. Allerdings hat man durch den Automatismus nur wenig Einflussmöglichkeiten auf den Ablauf bzw. den Zeitpunkt von Garbage Collections. Man kann die Garbage Collection aber durch diverse Aufrufparameter der JVM parametrieren, etwa verschiedene Aufräumalgorithmen wählen.

Statt aufwendige Überlegungen anzustellen, ist es einfacher und pragmatischer, die verschiedenen Implementierungen der Garbage-Collection-Algorithmen und deren Einfluss auf die Performance auszuprobieren. Dazu kann man das in Abschnitt 22.1.4 erwähnte Tool `VisualGC` zur Beobachtung von Garbage Collections benutzen.

22.6.2 Technik – Caching und Pooling

Objekt-Caching

Wie bereits für die Ein- und Ausgabe erwähnt, setzt man Caching ein, um häufig benötigte Daten in einem schnell zugreifbaren Speicher vorzuhalten. Bei umfangreichen Datenmengen und dadurch bedingt längeren Zugriffszeiten auf Objekte kann man einen LRU-Cache implementieren, der die zuletzt benutzten Objekte verwaltet.

Zur Demonstration des Einsatzes von Caching auf die Performance betrachten wir hier eine Liste `persons` von `Person`-Objekten. Darin erfolgt auf zwei verschiedene Arten eine Suche nach Objekten anhand ihres Namens. Die erste Realisierung nutzt eine lineare Suche und wird durch die nachfolgend gezeigte Methode `findPersonByName(String)` implementiert:

```
private Person findPersonByName(final String name)
{
    for (final Person current : persons)
    {
        if (current.getName().equals(name))
        {
            return current;
        }
    }
    return null;
}
```

Im ungünstigsten Fall ist dabei ein vollständiges Durchlaufen der Liste nötig, um festzustellen, ob ein passendes `Person`-Objekt vorhanden ist oder nicht. Zur Optimierung setzen wir die aus Abschnitt 6.1.10 bekannte generische Klasse `LruLinkedHashMap<K,V>` ein. Als Abstraktion erfolgen optimierte Zugriffe auf die gespeicherten `Person`-Objekte über eine Zugriffsmethode `getPerson(String)`. Diese nutzt wiederum initial die Methode `findPersonByName(String)`, um den Cache transparent für die Applikation zu füllen. Ein erster Zugriff auf ein `Person`-Objekt kostet daher maximal $O(n)$. Alle Folgezugriffe auf dieses oder ein anderes im Cache gespeichertes `Person`-Objekt sind dann aber sehr schnell in $O(1)$ möglich, da hier ein Lookup in der `LruLinkedHashMap<String, Person>` erfolgt:

```
public Person getPersonUsingCache(final String name)
{
    Person cachedPerson = cacheMap.get(name);

    if (cachedPerson != null)
    {
        return cachedPerson;
    }

    final Person person = findPersonByName(name);
    if (person != null)
    {
        cacheMap.put(name, person);
    }
    return person;
}
```

Mit folgender `main()`-Methode erzeugen wir 10.000 `Person`-Objekte. Anschließend werden als Testszenario 1.000, 10.000 und 100.000 Zugriffe auf eine Menge von immer denselben 10 `Person`-Objekten durchgeführt. Ein Cache der Größe 10 wäre demnach ausreichend, um die Zugriffe extrem zu beschleunigen.

Das folgende Listing zeigt die Klasse `LruListCacheExample`, die die zuvor beschriebene Funktionalität der Performance-Messung umsetzt:

```
public final class LruListCacheExample
{
    private final LruLinkedHashMap<String, Person> cacheMap;
    private final List<Person>                     persons;

    public LruListCacheExample(final List<Person> persons, final int cacheSize)
    {
        this.cacheMap = new LruLinkedHashMap<>(cacheSize);
        this.persons = persons;
    }

    public static void main(final String[] args)
    {
        final int PERSON_COUNT = 10_000;
        final List<Person> persons = new ArrayList<>();
        for (int i = 0; i < PERSON_COUNT; i++)
        {
            final Person newPerson = new Person("Person " + i, Gender.MALE, i);
            persons.add(newPerson);
        }

        final int[] cacheSizes = { 10, 20, 7 };
        for (int size : cacheSizes)
        {
            System.out.println("CacheSize = " + size);
            final LruListCacheExample cache =
                                        new LruListCacheExample(persons, size);

            performTests(cache, "findPersonByName", false);
            performTests(cache, "getPersonUsingCache", true);
        }
    }

    public static void performTests(final LruListCacheExample cache,
                                    final String info, final boolean useCache)
    {
        final long[] maxIterationCount = { 1000, 10_000, 100_000 };
        for (long max : maxIterationCount)
        {
            System.out.println("Element count " + max);

            PerformanceUtils.startMeasure(info);
            for (long i = 0; i < max; i++)
            {
                for (int j = 0; j < 10; j++)
                {
                    final String name = "Person " + (j * 1000);
                    if (useCache)
                        cache.getPersonUsingCache(name);
                    else
                        cache.findPersonByName(name);
                }
            }
            PerformanceUtils.stopMeasure(info);
            PerformanceUtils.printTimingResult(info);
        }
    }
    // ...
```

Listing 22.6 *Ausführbar als* **'LRULISTCACHEEXAMPLE'**

Führen wir nun das Programm LRULISTCACHEEXAMPLE aus, um zu untersuchen, welchen Einfluss verschiedene Cache-Größen tatsächlich haben. Mit meinem iMac (Quad-Core i7 mit 4 GHz) und JDK 14.0.1 erhalte ich die in Tabelle 22-5 aufgelisteten Zeiten, wobei – wie bei den vorherigen Messungen auch – nicht die absoluten Werte, sondern die Relationen von Interesse sind.

Tabelle 22-5 *Performance-Gewinn Caching*

	Cache-Größe 10			Cache-Größe 7		
	Anzahl Elemente			Anzahl Elemente		
Zugriffsvariante	**1.000**	**10.000**	**100.000**	**1.000**	**10.000**	**100.000**
`findPersonByName()`	185 ms	1.709 ms	17.010 ms	171 ms	1.698 ms	17.064 ms
`getPersonUsingCache()`	2 ms	13 ms	98 ms	172 ms	1.712 ms	17.016 ms

Anhand der Werte erkennt man, dass die Zeiten des indizierten Zugriffs über die Methode `findPersonByName(String)` erwartungsgemäß linear wachsen. Weiterhin zeigt sich deutlich, dass sich Caching, hier durch Zugriffe über die Methode `getPersonUsingCache(String)`, nicht in jedem Fall positiv auswirkt. Stattdessen sind die Faktoren Cache-Größe, Auswahl der zu cachenden Daten und Lokalität der Zugriffe ganz entscheidend. Erfolgen ständig Zugriffe auf beliebige Elemente und werden diese immer sofort in den Cache übernommen, so profitiert man nicht durch den Einsatz von Caches. Zum Teil ist sogar ein negativer Effekt zu messen. Dies macht die in diesem Beispiel bewusst zu klein gewählte Cache-Größe von 7 deutlich. ***Sind Zugriffsstrategie, Cache-Größe usw. passend gewählt, so kann man extreme Performance-Gewinne erzielen***. Die Zugriffszeit ist dann nahezu konstant und nicht von der Anzahl der gespeicherten Daten abhängig. Eine Vergrößerung des Cache auf 20 Einträge bringt – wie Sie sicher schon vermuteten – allerdings keine nennenswerten Performance-Gewinne mehr. Die gemessenen Ausführungszeiten sind (abgesehen von möglichen Messungenauigkeiten und gegebenenfalls Einflüssen durch den Hotspot-Optimierer) fast identisch zu denen für einen Cache der Größe 10.

Tipp: Probleme von Caches

Beim Einsatz von Caches muss man große Sorgfalt walten lassen, dies gilt vor allem bei Nebenläufigkeit und bei Mehrprozessormaschinen. Man benötigt dann ein spezielles Protokoll, um einen Abgleich mehrerer Caches sicherzustellen. Diese Problematik ist auch unter dem Begriff ***Cache-Kohärenz*** bekannt. Auch besteht das Problem der Datenkonsistenz, wenn einige Programmteile am Cache vorbei auf die Datenquellen zugreifen.

Objekt-Pools

Die Idee beim Einsatz von sogenannten ***Objekt-Pools*** ist es, Objekte nicht ständig neu zu erzeugen. Stattdessen werden Objekte einer initial erzeugten Menge von Objekten verwendet und nach Gebrauch wieder an den Pool zurückgegeben.

Im Apache Commons-Framework[8] findet sich folgendes Interface zur Realisierung von Objekt-Pools:

```
public interface ObjectPool
{
    Object borrowObject();
    void returnObject(Object borrowed);
}
```

Aus einem solchen Objekt-Pool wird eine momentan unbenutzte Instanz ausgewählt. Diese wird dann vom Aufrufer für den Einsatz initialisiert, sodass auf eine (möglicherweise aufwendige) Konstruktion des Objektes selbst verzichtet werden kann. Diese Form des Objekt-Recyclings ist hilfreich, wenn zu erzeugende Objekte komplex und daher in ihrer Instanziierung teuer sind. Ein weiteres Einsatzgebiet ist die Einschränkung der Anzahl erzeugter Objekte, z. B. weil die durch diese Objekte verwalteten Hardwareressourcen beschränkt sind. Dies gilt auch für den Fall, dass die Anzahl der Objektzustände relativ gering ist und daher eine begrenzte Anzahl unveränderlicher Objekte existieren kann, die sämtliche Einsatzgebiete abdecken. Das einfachste Beispiel dafür ist die Klasse `Boolean` aus dem JDK, die lediglich zwei Zustände besitzt.

Wie in Abschnitt 5.2 besprochen, existieren auch für die Wrapper-Klassen `Integer` und `Long` spezielle Caches und es findet ein Objekt-Pooling statt. Performancetechnisch ist ein `Long.valueOf(x)` einem `new Long[x]` vorzuziehen.

Bewertung War dieses Objekt-Pooling in älteren JVMs noch von größerem Nutzen für die Performance, so sind die JVMs im Laufe der letzten Jahre im Bereich der Objekterzeugung und der Garbage Collection deutlich schneller geworden und machen den Einsatz von Objekt-Pools in der Regel überflüssig.

Abgesehen von der Optimierung bei der Konstruktion, setzt Objekt-Pooling zwingend voraus, dass sich alle Programmteile korrekt verhalten, d. h. die Objekte vollständig freigeben und keine Referenzen darauf zwischenspeichern. Gibt ein Programmteil ein genutztes Objekt nach Gebrauch zwar an den Pool zurück, speichert aber weiterhin eine Referenz darauf, so ist dies problematisch: Objektreferenzen, die zwischengespeichert werden, zeigen auf mittlerweile neu initialisierte Objekte, die von anderen Programmteilen verwendet werden. Schnell kommt es zu Inkonsistenzen.

Meistens ist im Voraus nicht bekannt, wie viele Objekte in einem solchen Pool vorgehalten werden sollten. Werden weniger Objekte angefragt als bereitgestellt, so verschwendet man sowohl Speicherplatz als auch Rechenzeit (für die initiale Konstruktion). Werden jedoch mehr Objekte angefordert, als im Pool verfügbar sind, so muss eine spezielle Verwaltung einer Warteliste selbst programmiert werden. Als Abhilfe

[8] `http://commons.apache.org/`

kann man auch dynamisch wachsende Pools selbst realisieren. ***Aufgrund der genannten Einschränkungen sollte der Einsatz von Objekt-Pools jedoch gut überlegt werden***.

Beispiele im JDK Die JVM nutzt für Strings mit dem Stringliteral-Pool einen Objekt-Pool. Dieses Pooling erfolgt für die Applikation transparent. Insbesondere sind keine zusätzlichen Methodenaufrufe notwendig, sodass die zuvor genannten Nachteile allgemeiner Objekt-Pools nicht gelten. Weiterhin stellt die Klasse `Boolean`, wie bereits erwähnt, die simpelste Form eines Objekt-Pools dar. Aufgrund der Beschränkung auf zwei unveränderliche Objekte existieren bei dieser Realisierung keine negativen Auswirkungen durch nicht zurückgegebene oder veraltete Referenzen.

Wir messen nun die Zeiten zur Objektkonstruktionen per `new Boolean(true)` gegen die Nutzung gemeinsamer Instanzen aus dem Pool per `Boolean.TRUE` für die Speicherung von jeweils 10 Millionen `Boolean`-Werten in einem Array:

```
public static void main(final String[] args)
{
    final Boolean[] testArray = new Boolean[10_000_000];
    reportMemory();
    performTests(testArray, "Boolean.TRUE", false);
    reportMemory();
    performTests(testArray, "new Boolean(true)", true);
    reportMemory();
}

public static void performTests(final Boolean[] testArray,
                                final String info,
                                final boolean createNew)
{
    PerformanceUtils.startMeasure(info);
    for (int i = 0; i < testArray.length; i++)
    {
        if (createNew)
            testArray[i] = new Boolean(true);
        else
            testArray[i] = Boolean.TRUE;
    }
    PerformanceUtils.stopMeasure(info);
    PerformanceUtils.printTimingResult(info);
}

private static void reportMemory()
{
    MemoryInfo.gcAndSleep5s();
    System.out.println(MemoryInfo.statistics());
}
```

Listing 22.7 *Ausführbar als* '**BOOLEANPOOLINGOPTIMIZATION**'

Diese Optimierung deckt mehrere Ebenen und Techniken ab: Es erfolgt eine Optimierung bezüglich des Speicherverbrauchs durch den Verzicht der Konstruktion von 10 Millionen `Boolean`-Objekten und eine Optimierung bezüglich der Laufzeit der Objekterzeugungen. Tabelle 22-6 zeigt die Ergebnisse der Messungen.

Tabelle 22-6 *Performance-Gewinne durch* `Boolean`*-Pooling*

	`new Boolean(true)`	`Boolean.TRUE`
Ausführungszeit	313 ms	71 ms
Speicherverbrauch	159.938.664 Bytes	3.102.624 Bytes

Anhand der Zahlen kann man zwei Dinge ablesen: Die JVM ist sehr performant für Objekterzeugungen, hier am Beispiel für `Boolean`-Objekte. Daher lohnt sich ein Objekt-Pooling zur Einsparung von CPU-Zyklen eher selten. Zwar finden wir einen Faktor von ca. 5, aber relativ zu 10 Millionen Objektkonstruktionen sind die absoluten Zeiten von 313 ms zu 71 ms oft vernachlässigbar, vor allem bei Interaktionen über das Netzwerk. Interessanterweise schwanken die Werte gemessen mit meinem iMac (Quad-Core i7 mit 4 GHz) zwischen den Updates des JDKs enorm: Obige Zahlen wurden mit JDK 14.0.1 gemessen, Java 8 unterschied sich mit 590 ms zu 11 ms deutlich. Was ich damit sagen will ist: ***Messen Sie zum Feintuning unbedingt mehrmals und auf der Version der JVM, die Sie verwenden.***

Auch hier habe ich einmal wieder mit JMH (vgl. Abschnitt 14.4) nachgemessen, um einen Eindruck von der Genauigkeit der Eigenimplementierung des Benchmarks zu erhalten. Da die Ergebnisse recht ausführlich sind, werden sie nachfolgend auf das Wesentliche gekürzt gezeigt – trotzdem wird deutlich, dass die Variante mit Pooling schneller ist, wenn auch mit JMH scheinbar nicht so stark wie in der Eigenmessung:

```
Benchmark                                          Mode  Cnt      Score  Units
BooleanPoolingOptimizationJMH.booleanFromPool      avgt   25  70111.008  us/op
BooleanPoolingOptimizationJMH.newlyCreatedBooleans avgt   25  93112.918  us/op
```

Entscheidender ist jedoch der Einfluss auf den Speicherbedarf. Hier kann es durch Objekt-Pooling zu deutlichen Einsparungen kommen. Wir sehen in diesem Fall etwa den Faktor 50, wobei das sehr stark von der Speicherfreigabe abhängt und noch deutlich stärker sein kann. Führen wir eine Überschlagsrechnung durch und teilen den verbrauchten Speicher durch die Anzahl der erzeugten Objekte, so erkennen wir, dass jedes `Boolean`-Objekt etwa 16 Bytes verbraucht. Weitere Details zum Einfluss der Wahl von Datentypen auf den Speicherverbrauch werden in Abschnitt 22.6.4 besprochen.

Hinweis: Messungenauigkeiten beim Speicherverbrauch

Für gemessene Ausführungszeiten habe ich bereits bei der Diskussion der für Datenstrukturen ermittelten Messwerte mögliche Ungenauigkeiten thematisiert. Ähnliches gilt für Messungen des Speicherverbrauchs. Diese sind aufgrund der Arbeit des Garbage Collectors nur Annäherungswerte, können aber häufig trotzdem auf Probleme aufmerksam machen.

22.6.3 Optimierungen der Stringverarbeitung

In diesem Abschnitt wollen wir analysieren, warum der massive Einsatz von Strings und Stringoperationen zu Performance-Problemen führen kann. Häufig liegt dies daran, dass Strings intuitiv wie veränderbare Objekte – auch bedingt durch ihr API – eingesetzt werden, aber intern als unveränderliche Objekte implementiert sind. Was heißt das genau? Jede Veränderung an einem String führt zu einer neuen Instanz. So entstehen schnell viele temporäre Stringobjekte, die alsbald wieder verworfen werden, da sie nur ein Teilergebnis einer komplexeren Operation sind. Wie bereits erwähnt, ist die JVM bezüglich der Objekterzeugung und der Garbage Collection seit ihren Anfängen viel performanter geworden. Strings bergen jedoch ein weiteres Problem: Sie werden in den Stringliteral-Pool (vgl. Abschnitt 5.3.1) eingetragen und belegen Speicher im Metaspace-Bereich, der extrem selten aufgeräumt wird. Teilergebnisse von Stringoperationen belegen so diesen Bereich potenziell für längere Zeit.

`StringBuffer` / `StringBuilder` als Abhilfe?

Häufig wird als Optimierung von Stringoperationen der Einsatz der Klasse `StringBuffer` bzw. `StringBuilder` vorgeschlagen. Das Problem temporär erzeugter `String`-Instanzen lässt sich dadurch mildern. Das dazu notwendige Vorgehen wurde in Abschnitt 5.3 vorgestellt. Manchmal werden diese Klassen bereits genutzt, obwohl nur einige wenige Verknüpfungen von Strings durchgeführt werden müssen. Darunter leidet allerdings die Lesbarkeit, und außerdem wird der Sourcecode länger.

Betrachten wir ein Programm, das die Laufzeit für verschiedene Stringoperationen innerhalb einer Schleife mit einer Million Durchläufen ermittelt und so eine Abschätzung von Performance-Gewinnen beim Einsatz einer speziellen Methode erlaubt:

```
public static void main(final String[] args)
{
    final String name = "Sarah vom Auetal";
    final int counter = 1_000_000;

    String result = "";

    // Messung mit String +
    result = measureStringPlus(name, counter);

    // Messung mit String +=
    result = measureStringPlusEquals(name, counter);

    // Messung mit StringBuilder
    result = measureStringBuilderAppend(name, counter);

    // Messung mit StringBuilder und initialer Kapazität
    result = measurePresizedStringBuilderAppend(name, counter);

    // Messung mit StringBuffer
    result = measureStringBufferAppend(name, counter);

    // Messung mit StringBuffer und initialer Kapazität
    result = measurePresizedStringBufferAppend(name, counter);
```

```
    // Messung mit String und Formatter
    result = measureStringAndFormatter(name, counter);

    // Messung mit StringBuilder, initialer Kapazität und Formatter
    result = measureStringBuilderAndFormatter(name, counter);

    // Ausgabe des Ergebnisses vermeidet Weg-Optimierung der Berechnungen
    System.out.println(result);
}

private static String measureStringPlus(final String name, final int counter)
{
    String result = "";
    PerformanceUtils.startMeasure("String +");
    for (int i = 0; i < counter; i++)
    {
        result = "Mein Hund ist " + i + " Jahre alt und heißt " + name + ".";
    }
    PerformanceUtils.stopMeasure("String +");
    PerformanceUtils.printTimingResultWithAverage("String +", counter);
    return result;
}

private static String measureStringPlusEquals(final String name,
                                              final int counter)
{
    String result = "";
    PerformanceUtils.startMeasure("String +=");
    for (int i = 0; i < counter; i++)
    {
        result = "Mein Hund ist ";
        result += i;
        result += " Jahre alt und heißt ";
        result += name;
        result += ".";
    }
    PerformanceUtils.stopMeasure("String +=");
    PerformanceUtils.printTimingResultWithAverage("String +=", counter);
    return result;
}
// ...
```

Listing 22.8 *Ausführbar als* '**STRINGBENCHMARK**'

In Tabelle 22-7 sind die Ergebnisse des auf meinem iMac (Quad-Core i7 mit 4 GHz) und JDK 14.0.1 ausgeführten Benchmarks zusammengefasst, wobei weitere, im Listing nicht explizit gezeigte Varianten der Konkatenation auf ihre Performance untersucht wurden, etwa die Auswirkungen der Angabe einer passenden initialen Größe für einen `StringBuffer`. Das wird in der Tabelle als »Presized« gekennzeichnet.

Tabelle 22-7 *Performance-Vergleiche von Stringkonkatenationen*

Variante	Dauer	Durchschnitt
`+`	127 ms	0,000127 ms
`+=`	207 ms	0,000207 ms
`StringBuilder`	144 ms	0,000144 ms
`StringBuilder` »Presized«	140 ms	0,000140 ms
`StringBuffer`	130 ms	0,000130 ms
`StringBuffer` »Presized«	118 ms	0,000118 ms
`String.format()`	1.677 ms	0,001677 ms
`String` und `Formatter`	1.273 ms	0,001273 ms
`StringBuilder` und `Formatter`	1.268 ms	0,001268 ms

Die Aktionen habe ich in einem Benchmark mit JMH (vgl. Abschnitt 14.4) erneut ausgeführt. Mit für den einen oder anderen verblüffenden Ergebnissen. Die einfache Konkatenation ist extrem performant – die Messungen sind hier gekürzt gezeigt:

```
Benchmark                                                Mode  Cnt  Score
StringJoinBenchmark.measurePresizedStringBufferAppend    avgt   25  0.073 ms/op
StringJoinBenchmark.measurePresizedStringBuilderAppend   avgt   25  0.055 ms/op
StringJoinBenchmark.measureStringAndFormatter            avgt   25  0.884 ms/op
StringJoinBenchmark.measureStringBufferAppend            avgt   25  0.106 ms/op
StringJoinBenchmark.measureStringBuilderAndFormatter     avgt   25  0.860 ms/op
StringJoinBenchmark.measureStringBuilderAppend           avgt   25  0.066 ms/op
StringJoinBenchmark.measureStringBuilderAppendWithReset  avgt   25  0.041 ms/op
StringJoinBenchmark.measureStringFormat                  avgt   25  0.900 ms/op
StringJoinBenchmark.measureStringPlus                    avgt   25  0.024 ms/op
StringJoinBenchmark.measureStringPlusEquals              avgt   25  0.024 ms/op
```

Diskussion

Man sieht eindrucksvoll, dass der oftmals verpönte Operator '+' für das Verketten weniger Strings am besten abschneidet. Allerdings sollte man bedenken, dass die Messergebnisse deutlich von Lauf zu Lauf variieren, die aufgezeigten Tendenzen aber durchaus gelten.

Abgesehen von den formatierten Varianten sind kaum relevante Geschwindigkeitsunterschiede ermittelbar. Gemäß diesen Messwerten kann man oft die übersichtliche Schreibweise mit dem Operator '+' den schlechter lesbaren Varianten mit `StringBuilder` vorziehen. ***Bitte beachten Sie, dass hier in den Schleifen auch immer noch die jeweiligen Objekte, z. B. `StringBuilder`, neu erzeugt wurden***. Eine Optimierung besteht darin, etwa nur einmalig einen `StringBuffer` zu erzeugen und diesen wiederzuverwenden, indem man dessen Länge nach jedem Durchlauf auf den Wert 0 setzt. ***Mit dieser Modifikation sind '+' und `StringBuilder` nahezu gleich schnell***.

Anhand der Messwerte sieht man auch, dass Stringaufbereitungen durch die `Formatter`-Klassen bzw. die `format()`-Methode etwa um den Faktor 10 – 15 langsamer sind als die Konkatenation mit dem Operator '+'.

22.6.4 Technik – Vermeidung unnötiger Aktionen

Zur Vermeidung einer aufwendigen Konstruktion schwergewichtiger Objekte haben wir in Abschnitt 22.3 bereits die Technik Lazy Initialization kennengelernt. Hierbei werden komplexe, schwergewichtige Objekte in mehrere kleinere, leichtgewichtigere Objekte aufgeteilt, um optionale Teile nur bei Bedarf nachzuladen bzw. zu initialisieren. Der aktuelle Abschnitt führt die Gedanken des Speichersparens und der Vermeidung unnötiger Aktionen auf feingranularer Ebene fort. Bitte beachten Sie, dass sich derartige Mikrooptimierungen nur für sehr große Datenbestände oder extrem häufig ausgeführte Programmteile lohnen. Keinesfalls sollten Sie damit starten, bevor Sie nicht auf anderen Ebenen optimiert haben.

Reduziere die Kosten der Objekterzeugung

Fast alles ist in Java ein Objekt. Demnach werden in Applikationen oftmals viele Objekte angelegt, was sich negativ auf die Performance auswirken kann: Die Objektkonstruktionen kosten geringfügig Laufzeit. Zwar ist die JVM diesbezüglich sehr performant geworden, sodass dies heutzutage normalerweise kein Problem mehr darstellt. Jedoch besitzt die Erzeugung von Objekten indirekten Einfluss auf die Performance: Zum einen kommt es dadurch später zu einer erhöhten Aktivität des Garbage Collectors, um diese Objekte wieder freizugeben. Zum anderen kann auch der von den Objekten belegte Speicher zu einem Mangel an freiem Speicher führen. Daraus resultieren eventuell Garbage Collections, die (wahrnehmbare) Pausen in der Abarbeitung verursachen.

Je weniger Objekte (unnötig) erzeugt werden, desto mehr Speicher steht der Applikation zur Verfügung. Dazu kann man versuchen, die Kosten der Objekterzeugung und die Anzahl der erzeugten Objekte zu minimieren, etwa indem man Lazy Initialization nutzt und primitive Datentypen anstelle korrespondierender Instanzen der Wrapper-Klassen bevorzugt. Außerdem lässt sich teilweise die Freigabe von Objekten durch den Garbage Collector erleichtern, indem Referenzvariablen möglichst schnell auf `null` gesetzt werden, wenn diese nicht mehr benötigt werden. Der Garbage Collector kann die nicht mehr referenzierten Objekte einsammeln und eventuell (etwas) früher freigeben.

Hinweis: Ausführungszeit von `new`

Ein `new` in Java führt im Moment des Aufrufs nicht unbedingt zu einer Bereitstellung von Speicher durch das Betriebssystem, da der Speicher in der Regel bereits in der JVM verfügbar ist. Daher kann ein `new` in Java sogar schneller sein als in C++.

Bevorzuge primitive Datentypen anstelle von Wrapper-Klassen

Bei sehr großen Datenmengen lässt sich die Performance positiv beeinflussen, wenn man primitive Typen anstelle der zugehörigen Wrapper-Klassen verwendet. Das liegt an folgenden Eigenschaften primitiver Typen:

- Sie sind schnell zu erzeugen (kein Aufwand zur Laufzeit).
- Sie sind schnell im Zugriff (keine Referenzen).
- Sie verbrauchen etwas weniger Speicher als Instanzen der Wrapper-Klassen, z. B. 4 Bytes (`int`) zu 16 Bytes (`Integer`).
- Sie bergen nicht die Gefahr uninitialisierter Referenzvariablen und daraus resultierender `NullPointerException`s.

Am Beispiel der Realisierung von Value Objects (VOs) (vgl. Abschnitt 3.4.5) kann man sich die Vorteile des Einsatzes primitiver Typen verdeutlichen. Nachfolgendes Beispiel zeigt zwei mögliche Varianten der Klassen `BadVO` und `GoodVO` zum Austausch von Daten zwischen Applikationen. Die Daten bestehen hier zur Vereinfachung der Darstellung nur aus einer Kennung `id` und zwei Werten `value1` sowie `value2`.

Die Klasse `BadVO` nutzt diverse Wrapper-Klassen zur Definition ihrer Attribute und speichert demnach Objektreferenzen:

```
class BadVO implements Serializable
{
    Long id;
    Integer value1;
    Double value2;
}
```

Eine solche Realisierung kann erforderlich sein, um für optionale Attribute »kein Wert« ausdrücken zu können. Dazu nutzt man `null` als Wert. Benötigt man diese Funktionalität nicht, so verbraucht der Ansatz deutlich mehr Speicher als die im Folgenden gezeigte `GoodVO`-Variante, die primitive Datentypen für die Attribute verwendet:

```
class GoodVO implements Serializable
{
    long id;
    int value1;
    double value2;
}
```

Betrachten wir dies im Detail, indem wir die Größe beider VOs anhand der verwendeten Datentypen berechnen: Mit dem Programm VOSERIALIZATIONSIZEEXAMPLE ermitteln wir, dass ein `BadVO` 118 Bytes benötigt, ein `GoodVO` dagegen nur 74 Bytes (vgl. folgenden Praxistipp). Beachten Sie bitte, dass bei der Serialisierung auch der Package- und Klassenname gespeichert werden, weshalb nicht nur die reine Nutzlast sichtbar wird. Anhand der ermittelten Größenangaben liegt das maximale Einsparpotenzial schätzungsweise bei ca. 50 %, d. h., ein bevorzugt aus primitiven Typen zusammengesetztes Objekt ist nur etwa halb so groß wie ein VO, bestehend aus Wrapper-

Klassen. Dazu habe ich ein von den Attributen her umfangreiches VO-Pärchen erstellt, wo wir 602 Byte zu 303 Bytes für die Variante mit primitiven Datentypen ermitteln.

Diese Optimierung lohnt sich vor allem beim Einsatz von Zahlentypen. Bei VOs, die auch Objektreferenzen und Strings speichern, sind die Unterschiede deutlich geringer. Neben dem reinen Speicherverbrauch ist bei der Übertragung von VOs zwischen verschiedenen JVMs noch ein weiterer Aspekt von Interesse. Die Reduktion der zu übertragenden Datenmenge führt zu einer gesteigerten Geschwindigkeit bei der Kommunikation.

Tipp: Größenabschätzungen für Objekte

Der Speicherverbrauch eines Objekts lässt sich anhand seiner Attribute berechnen. Für primitive Typen gilt: `byte` ≡ 1, `short` ≡ 2, `int` ≡ 4, `long` ≡ 8, `float` ≡ 4, `double` ≡ 8 Bytes. Referenzvariablen benötigen mindestens 8 Bytes, ebenso wie jede Instanz der Klasse `Object` – `String`-Instanzen noch mal deutlich mehr. Die Wrapper-Klassen belegen konstant 16 Bytes anstatt der unterschiedlichen Größen der primitiven Typen. Zusatzinformationen findet man im Buch »Java Platform Performance – Strategies and Tactics« von Steve Wilson und Jeff Kesselman [73] sowie online unter `http://www.javaworld.com/javaworld/javatips/jw-javatip130.html`.

22.7 CPU-bound-Optimierungen

In wenigen Fällen wird die Ausführungsgeschwindigkeit eines Programms maßgeblich durch die reine Bearbeitungszeit der Anweisungen in der CPU aufgrund von nicht optimalen Berechnungen verursacht. Im Folgenden werden verschiedene CPU-bound-Optimierungstechniken lediglich kurz vorgestellt, weil die meisten dieser Optimierungen nur nach allen anderen Optimierungen überhaupt in Betracht gezogen werden sollten. In Ihrer täglichen Praxis wird das vermutlich eher selten der Fall sein. Dabei gibt es lediglich eine Ausnahme, nämlich die Technik Memoization, die zur Optimierung bei rekursiven Berechnungen gezeigt wird. Diese stellt nahezu immer eine sehr sinnvolle Optimierung dar.

22.7.1 Technik – Wahl passender Strategien

Manchmal existiert offensichtlich ein Performance-technisch besser geeigneter Algorithmus für ein zu lösendes Problem als der derzeit gewählte. Bei Performance-Problemen kann dann durch die Wahl des besseren Algorithmus für Abhilfe gesorgt werden. Dies haben wir bereits in Abschnitt 22.1.3 für die Berechnung einer Summe kennengelernt.

Beispiel: Rekursive vs. iterative Berechnung der Fibonacci-Zahlen

Ein weiteres Beispiel für die Wahl einer passenden Strategie ist, statt Rekursion eine iterative Lösung zu nutzen, etwa für die Berechnung von Fibonacci-Zahlen. Schauen wir kurz die rekursive Definition an:

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & \forall n > 1 \end{cases}$$

Die Implementierung davon ist zwar rekursiv einfach möglich – allerdings nicht performant, weil es zu vielen Selbstaufrufen kommt:

```
public static long fibonacciRecursive(long n)
{
    if (n == 0 || n == 1)
        return n;

    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}
```

Es existiert eine deutlich effizientere, nicht rekursive Möglichkeit zur Berechnung:

```
public static long fibonacciIterative(final long n)
{
    if (n == 0 || n == 1)
        return n;

    long fib_n_2 = 0;
    long fib_n_1 = 1;
    long fib_n = 1;

    for (long i = 2; i <= n; i++)
    {
        fib_n = fib_n_2 + fib_n_1;

        fib_n_2 = fib_n_1;
        fib_n_1 = fib_n;
    }

    return fib_n;
}
```

Schreiben wir ein kleines Programm, um die Performance-Unterschiede zwischen beiden Varianten der Berechnung zu demonstrieren:

```
public static void main(final String[] args)
{
   final int[] values = { 10, 30, 35, 36, 37, 38, 39, 40,
                          41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 55, 57 };

   for (final int currentValue : values)
   {
      System.out.println("currentValue: " + NumberFormat.getIntegerInstance().
                                                          format(currentValue));
      System.out.println("----------------------------------------");

      final long start1 = System.nanoTime();
      final long fibRecursive = fibonacciRecursive(currentValue);
      final long end1 = System.nanoTime();

      System.out.println("fibonacciRecursive took: " +
                         TimeUnit.NANOSECONDS.toMillis(end1 - start1) + " ms");
      System.out.println("fibonacciRecursive is: " + fibRecursive);

      final long start2 = System.nanoTime();
      final long fibIterative = fibonacciIterative(currentValue);
      final long end2 = System.nanoTime();

      System.out.println("fibonacciIterative took: " +
                         TimeUnit.NANOSECONDS.toMillis(end2 - start2) + " ms");
      System.out.println("fibonacciIterative is: " + fibIterative);

      System.out.println("----------------------------------------");
   }
}
```

Listing 22.9 *Ausführbar als* **'FIBONACCIEXAMPLE'**

Führen wir das Programm FIBONACCIEXAMPLE aus, so sehen wir, dass bis etwa zur 40. Fibonacci-Zahl die Berechnungen recht zügig vonstattengehen. Danach wird deutlich, wie extrem die Laufzeit selbst auf einem iMac mit Quad-Core i7 mit 4 GHz und JDK 14.0.1 rasant zunimmt (vgl. folgende Tabelle 22-8).

Tabelle 22-8 *Vergleich rekursive und iterative Berechnung von Fibonacci-Zahlen*

	Eingabewert und Ausführungszeit in ms							
Berechnung	**10**	**30**	**35**	**40**	**45**	**50**	**55**	**57**
Rekursiv	0	4	39	416	4.524	49.294	551.062	1.511.893
Iterativ	0	0	0	0	0	0	0	0

Bereits für Zahlen ab ca. 45 macht sich die exponentielle Zeitzunahme drastisch bemerkbar – bei der iterativen Variante ist die Laufzeit zwar theoretisch $O(n)$, aber praktisch unbedeutend. Das liegt auch daran, dass n hier eher kleinere Werte annimmt.

Iterative Variante für riesige Wertbereiche Um den Performance-Vorteil der iterativen Berechnung zeigen zu können, wollen wir auch viel größere Fibonacci-Zahlen, etwa mit Werten für n von 1.000 und 10.000 berechnen. Dazu muss man aufgrund des beschränkten Wertebereiches von `long` das obige Programm nur ein wenig abwandeln und für Berechnungen die Klasse `BigInteger` nutzen:

```
public static BigInteger fib(final int n)
{
    if (n == 0)
        return BigInteger.ZERO;
    if (n == 1)
        return BigInteger.ONE;

    BigInteger fib_n_2 = BigInteger.ZERO;
    BigInteger fib_n_1 = BigInteger.ONE;
    BigInteger fib_n = BigInteger.ONE;

    for (int i = 2; i <= n; i++)
    {
        fib_n  = fib_n_2.add(fib_n_1);
        fib_n_2 = fib_n_1;
        fib_n_1 = fib_n;
    }
    return fib_n;
}

public static void main(final String[] args)
{
    System.out.println(fib(10));    // 55
    System.out.println(fib(20));    // 6.765
    System.out.println(fib(30));    // 832.040
    System.out.println(fib(40));    // 102.334.155
    System.out.println(fib(50));    // 12.586.269.025
    // über 200 Ziffern: 434665576869374564356885276750 40...849228875
    System.out.println(fib(1000));
    // über 2.000 Ziffern: 364476487643178326662161200510 75...947366875
    System.out.println(fib(10_000));
}
```

Listing 22.10 *Ausführbar als* **'FIBONACCIBIGINTEGEREXAMPLE'**

Selbst für riesige Fibonacci-Zahlen bzw. Ergebnisse erhält man beim Start des Programms FIBONACCIBIGINTEGEREXAMPLE nur Laufzeiten im Millisekundenbereich.

Hinweis: Rekursion als Performance-Problemindikator

Rekursion ist immer ein potenzieller Ansatzpunkt für Optimierungen. Nicht immer sollte man beim Verzicht auf Rekursion aber derart enorme Gewinne wie bei der Berechnung der Fibonacci-Zahlen erwarten. Zudem existiert mit Memoization eine effiziente Technik zur Optimierung. Somit sollte man auf Rekursion setzen, wenn diese das Problem besser beschreibt, und auf eine iterative Variante, wenn diese ebenso klar ist.

22.7.2 Technik – Caching und Pooling

Häufiger liest man, dass man Berechnungen dadurch optimieren sollte, dass Ergebnisse oder Werte vorausberechnet und in Caches oder Lookup-Tabellen gespeichert werden. Ganz so generell ist die Aussage sicher nicht richtig, aber mit Bedacht in den richtigen Situationen eingesetzt, kann Caching einen fundamentalen Unterschied machen. Schauen wir uns zwei Beispiele an, zunächst eins, das verdeutlicht, dass Caching für einfachere Berechnungen kaum Einfluss hat, und ein anderes, wo man das Potenzial sehr gut erkennt.

(Gegen-)Beispiel 1: Caching für mathematische Berechnungen

Früher wurde ein Caching häufig für Berechnungen von Sinus- oder Kosinuswerten vorgeschlagen. Ich habe Messungen angestellt, und diese zeigen nur geringe Auswirkungen auf die Performance. Das gilt selbst für aufwendigere Berechnungen durch Aufrufe von `Math.exp(double)`. Eine Million Aufrufe benötigen inklusive Speicherung in einem Array mit JDK 14.0.1 auf meinem iMac (Quad-Core i7 mit 4 GHz) nur 6 ms, sodass sich eine Zwischenspeicherung bei einer derartigen Berechnungsgeschwindigkeit eher selten lohnt. Diese Messung verdeutlicht nochmals die eingangs des Kapitels gemachte Aussage, dass heutzutage Performance-Optimierungen auf Sourcecode-Ebene oftmals nur minimale Auswirkungen auf die Gesamtperformance besitzen.

Vergleichen Sie die Ausführungszeiten auf Ihrem Rechner durch Aufruf des folgenden Programms MATHEXPEXAMPLE:

```
public static void main(final String[] args)
{
    final int counter = 1000 * 1000;
    final double[] testArray = new double[counter];

    PerformanceUtils.startMeasure("MathExpExample");
    for (int i = 0; i < counter; i++)
    {
        testArray[i] = Math.exp(i);
    }
    PerformanceUtils.stopMeasure("MathExpExample");
    PerformanceUtils.printTimingResult("MathExpExample");
}
```

Listing 22.11 *Ausführbar als* **'MATHEXPEXAMPLE'**

Gerade beim Einsatz von Lookup-Tabellen für mathematische Berechnungen muss man sich darüber im Klaren sein, dass dieses Vorgehen auch negative Seiten besitzt:

- **Genauigkeit** – Man wird sich auf eine gewisse Menge an vordefinierten Werten beschränken, etwa die Sinuswerte auf ein Grad genau. Für einige Berechnungen mag diese Genauigkeit ausreichend sein, für andere wiederum nicht.

- **Speicherverbrauch** – Je genauer bzw. umfangreicher die Lookup-Tabellen werden, desto mehr Speicher belegen sie zwangsläufig. Hier existiert ein Tradeoff zwischen verschiedenen Optimierungszielen. Wahrscheinlich wirken sich aber riesige Lookup-Tabellen ungünstiger aus als das, was sie an CPU-bound-Optimierung herausholen. Hier sind wieder umfangreiche Messungen gefragt.

Zwischenfazit Obwohl ich die Betrachtung dieser Thematik bewusst mit einem Gegenbeispiel begonnen habe, ist die Idee der Optimierung durch eine Berechnung im Voraus an sich korrekt und immer dann sinnvoll einzusetzen, wenn man schwergewichtige Objekte wiederverwenden kann oder aber ein Caching nicht mit einer Einschränkung der Lesbarkeit verbunden ist. Insbesondere bei Datenbankzugriffen kann man durch ein geschicktes Caching einiges an Performance-Optimierung erzielen. Aber es gibt noch eine weitere Variante, wo sich Caching lohnt, man spricht dabei von Memoization.

Beispiel 2: Memoization für rekursive Berechnungen

Zwar lassen sich viele Algorithmen mithilfe von Rekursion verständlich und zugleich elegant beschreiben. Allerdings war uns für die Berechnung der Fibonacci-Zahlen auch aufgefallen, dass Rekursion teilweise zu vielen Selbstaufrufen führt, die sich negativ auf die Performance auswirken können. Als Abhilfe existiert eine Technik, die sich ***Memoization*** nennt. Diese verfolgt die gleichen Ideen wie das Caching oder Zwischenspeichern zuvor berechneter Werte. Man vermeidet dadurch mehrfache Ausführungen, indem bereits ermittelte Ergebnisse für Folgeaktionen wiederverwendet werden.

Praktischerweise kann man Memoization einem bestehenden Algorithmus oft leicht hinzufügen und muss diesen nur minimal anpassen. Dazu benötigen wir eine Hilfsmethode, die die eigentliche Berechnungsmethode aufruft, sowie vor allem eine Datenstruktur zur Speicherung von Zwischenergebnissen. In diesem Fall nutzen wir eine Map, die an die Berechnungsmethode übergeben wird:

```
static long fibonacciOptimized(final int n)
{
    return fibonacciMemo(n, new HashMap<>());
}
```

In der ursprünglichen Methode umrahmen wir die eigentliche Berechnung mit den Aktionen zur Memoization: Bei jedem Berechnungsschritt schauen wir zuerst in der Map nach, ob bereits ein Ergebnis vorliegt, und geben dieses in dem Fall zurück. Ansonsten führen wir den Algorithmus wie zuvor aus, mit der minimalen Abwandlung, dass wir das Berechnungsergebnis in einer Variablen zwischenspeichern, um es zum Abschluss passend in der Lookup-Map hinterlegen zu können:

```
static long fibonacciMemo(final int n, final Map<Integer, Long> lookupMap)
{
    // MEMOIZATION: prüfe, ob vorberechnetes Ergebnis existiert
    if (lookupMap.containsKey(n))
        return lookupMap.get(n);

    // normaler Algorithmus mit Hilfsvariable für Resultat
    long result = 0;
    // rekursiver Abbruch
    if (n == 0 || n == 1)
        result = n;
    // rekursiver Abstieg
    else
        result = fibonacciMemo(n - 1, lookupMap) +
                 fibonacciMemo(n - 2, lookupMap);

    // MEMOIZATION: speichere berechnetes Ergebnis
    lookupMap.put(n, result);
    return result;
}
```

Führt man diese Varianten mit Memoization als Programm FIBONACCIOPTIMIZED-EXAMPLE für die 45. Fibonacci-Zahl aus, so liefert sie nach weniger als 1 Millisekunde ein Ergebnis. Das gilt ebenfalls für die 50. und die 70. Fibonacci-Zahl. Für größere müsste man dann wieder auf den Typ `BigInteger` wechseln.

```
public static void main(final String[] args)
{
    for (final int currentValue : new int[] { 10, 30, 40, 45, 50, 70 })
    {
        System.out.println("currentValue: " + currentValue);
        final long start = System.nanoTime();
        final long fibRecursive = fibonacciOptimized(currentValue);
        final long end = System.nanoTime();

        System.out.println("fibonacciOptimized took: " +
                           TimeUnit.NANOSECONDS.toMillis(end - start) + " ms");
        System.out.println("fibonacciOptimized is: " + fibRecursive);
        System.out.println("----------------------------------------");
    }
}
```

Listing 22.12 *Ausführbar als* '**FIBONACCIOPTIMIZEDEXAMPLE**'

22.7.3 Technik – Vermeidung unnötiger Aktionen

Der Einsatz primitiver Datentypen anstelle von Wrapper-Klassen ist gleich gut lesbar, kann aber Vorteile in der Ablaufgeschwindigkeit und bezüglich des Speicherverbrauchs bewirken. Zudem vermeidet man so Auto-Boxing und Auto-Unboxing. Auch für diese Optimierung findet man in der Literatur häufig übertriebene Aussagen zu deren Auswirkung auf die Performance. Um überhaupt Effekte messen zu können, müssen extrem viele Auto-Boxing-Vorgänge stattfinden.

In den ersten beiden Auflagen dieses Buchs verwendete ich zur Demonstration eine `equals(Object)`-Methode. Ab der 3. Auflage habe ich mich für ein artifizielles

Beispiel entschieden, das die Dinge besser verdeutlicht. Im folgenden Programm wird eine Summe mithilfe einer Schleife und einer Addition mit dem Wert 1 berechnet. Einmal benutzen wir dazu einen `long` und für die zweite Summation ein `Long`-Objekt. Dadurch kommt es beim Addieren zu Auto-Boxing und Auto-Unboxing:

```
public static void main(final String[] args)
{
   final long[] loopCounts = {1_000_000, 10_000_000, 100_000_000,
                              1_000_000_000, 10_000_000_000L, 100_000_000_000L};

   for (final long loopCount : loopCounts)
   {
      System.out.println("LoopCount: " + NumberFormat.getIntegerInstance().
                                                  format(loopCount));
      System.out.println("----------------------------------------");

      final long start1 = System.nanoTime();

      long sumAsPrimitive = 0;
      for (long i = 0; i < loopCount; i++)
      {
          sumAsPrimitive += 1;
      }

      final long end1 = System.nanoTime();
      System.out.println("Primitive sum took: " +
                         TimeUnit.NANOSECONDS.toMillis(end1 - start1));
      // Zugriff auf die Variable, um Optimierung zu vermeiden
      System.out.println("Primitive sum is: " + sumAsPrimitive);

      final long start2 = System.nanoTime();

      Long sumAsWrapper = 0L;
      for (long i = 0; i < loopCount; i++)
      {
          // Hier kommt es zu Auto-Boxing und Auto-Unboxing
          sumAsWrapper += 1;
      }

      final long end2 = System.nanoTime();
      System.out.println("Wrapper/Auto-/Un-)Boxing sum took: " +
                         TimeUnit.NANOSECONDS.toMillis(end2 - start2));
      // Zugriff auf die Variable, um Optimierung zu vermeiden
      System.out.println("Wrapper sum is: " + sumAsWrapper);

      System.out.println("----------------------------------------");

      final double factor = (end2 - start2) / (double) (end1 - start1);
      System.out.println(String.format("Factor %.2f", factor));
      System.out.println("----------------------------------------");
   }
}
```

Listing 22.13 *Ausführbar als* '**CPUBOUNDOPTIMIZATIONEXAMPLE**'

Führen wir das Programm CPUBOUNDOPTIMIZATIONEXAMPLE aus, um ein wenig mehr Gewissheit über den Einfluss von Auto-Boxing auf die Performance zu erhalten. Wie performant Berechnungen trotz Auto-Boxing und Auto-Unboxing sind, zeigt folgende Tabelle 22-9.

Tabelle 22-9 *Performance-Vergleich zwischen Wrapper-Objekten und primitiven Typen*

	Anzahl Summationen				
Berechnung	**1.000.000**	**10.000.000**	**100.000.000**	**1.000.000.000**	**10.000.000.000**
Auto-Boxing (AB)	18 ms	85 ms	377 ms	2.889 ms	31.447 ms
Primitive Typen (PT)	4 ms	39 ms	130 ms	1.319 ms	12.846 ms
Faktor AB : PT	4,53	2,18	2,69	2.19	2,45

Insgesamt sind Berechnungen mit primitiven Typen natürlich schneller, im Durchschnitt um etwa den Faktor 2 – 5. ***Allerdings zeigen die absoluten Zeiten, dass durch diese Optimierung tatsächlich nur sehr wenig Laufzeitgewinn zu erzielen ist***. Dies wird insbesondere klar, wenn man bedenkt, wie viele Durchläufe nötig sind, um die gemessenen Zeiten zu produzieren: Selbst eine Milliarde Additionen benötigen nicht einmal drei Sekunden.

Fazit

Bitte erinnern Sie sich nochmals an meine eingangs gemachte Aussage: ***Low-Level-Optimierungen sollten Sie im Normalfall nicht durchführen müssen***. Bevor Sie daran denken, so etwas zu tun, schauen Sie, ob nicht auf höherer Ebene eine zweckmäßigere Optimierung vorgenommen werden kann.

Hinweis: Verwendeter Rechner und JDK

Die Messungen für diese 5. Auflage wurden zur besseren Vergleichbarkeit alle auf einem iMac (Quad-Core i7 mit 4 GHz) und JDK 14.0.1 durchgeführt.

Erfahrungsgemäß weichen die Messungen auf verschiedenen Rechnern zum Teil deutlich voneinander ab. Demnach stellen die von mir ermittelten Werte nur Anhaltspunkte dar und variieren abhängig nicht nur von Rechner zu Rechner, sondern auch zwischen unterschiedlichen JDK-Versionen, ja sogar innerhalb verschiedener Updates. Messen Sie durch Start der jeweiligen Gradle-Tasks also selbst nach. Die grundsätzlichen Tendenzen sollten Sie aber auch auf anderer Hardwareausstattung und sogar anderen JDK-Versionen gut nachvollziehen können.

22.8 Weiterführende Literatur

Weiterführende Informationen zu Optimierungen finden sich in folgenden Büchern sowie auf der unten angegebenen Internetseite:

- »**Performant Java programmieren**« von Hendrik Schreiber [64]
  Dieses Buch ist sehr lesenswert und gibt viele interessante Einblicke in das Gebiet der Performance-Optimierung. Es verliert sich nicht in Details, sondern bringt die Ideen und Hinweise klar auf den Punkt. Daher empfehle ich dieses Buch ausdrücklich.
- »**Java Performance Tuning**« von Jack Shirazi [67]
  Dieser Klassiker unter den Performance-Tuning-Büchern beleuchtet das Thema Performance-Optimierung auf breiter Front, geht in einigen Bereichen allerdings sehr auf Optimierungen auf Sourcecode-Ebene ein.
- »**Java Platform Performance – Strategies and Tactics**« von Steve Wilson und Jeff Kesselman [73]
  Dieses Buch ist neben dem von Jack Shirazi ein weiterer Klassiker unter den Performance-Tuning-Büchern und gibt einen guten Überblick über das Thema Performance-Optimierung, steigt allerdings nicht so detailliert in die jeweiligen Themen ein wie das zuvor genannte Buch.
- »**Java Tuning White Paper**« unter `http://java.sun.com/performance/reference/whitepapers/tuning.html`
  Diese Dokumentation enthält einige nützliche Hinweise und auch Links auf weitere Seiten, etwa zur Optimierung der Garbage Collection.

23 Schlussgedanken

Zum Abschluss dieses Buchs möchte ich Ihnen ein paar persönliche Gedanken zum Thema Entwickeln, Erstellen von Designs und Unit Tests sowie zum Vorgehen beim Entwurf von Softwaresystemen in Form eines von mir praktizierten Vorgehens darlegen, das ich den »Think first then code and test«-Ansatz nenne.

Zu guter Letzt wünsche ich Ihnen viel Spaß beim Entwickeln und Unit-Testen und weniger Stress durch die neu gewonnene Sicherheit und das Vertrauen in den eigenen Sourcecode nach der Lektüre dieses Buchs mit seiner Vielzahl an Empfehlungen und Praxistipps.

Mein »Think first then code and test«-Ansatz

Der Weg zu qualitativ hochwertiger und testbarer Software erfordert sauberes Design und eine sorgfältige, gewissenhafte Implementierung, was eine strukturierte Arbeitsweise und gute Werkzeuge voraussetzt. Für den konkreten Ablauf beim Entwurf von Software gibt es diverse, teils kontroverse Vorgehensweisen. Ich halte von Extremen nichts, denn die Wirklichkeit ist nie schwarz oder weiß. Vielmehr möchte ich nachfolgend beschreiben, welches Vorgehen sich für mich beim Entwurf in den letzten zwanzig Jahren bewährt hat, um stabile, wartbare und (nach wenigen Testläufen) nahezu fehlerfreie Software zu erstellen. Meiner Erfahrung nach ist weder der Weg des Big Upfront Design (BUD), bei dem alles bis in kleinste Detail versucht wird, vorab zu spezifizieren, noch derjenige, der ausschließlich auf Test-Driven Development (TDD) basiert ohne eine Einbettung in einen Architektur- und Designprozess, zielführend. Vielmehr sollte man eine inkrementelle iterative Kombination aus dem Besten beider Ansätze praktizieren.

Stellen wir uns folgende Frage: **Wie schreibt man qualitativ hochwertige Software (und Tests)?** Zur Beantwortung möchte ich mit einem Zitat von Antoine de Saint-Exupéry (aus dem Buch »Die Stadt in der Wüste«) beginnen:

> WENN DU EIN SCHIFF BAUEN WILLST, SO FANGE NICHT DAMIT AN, HOLZ ZU SAMMELN, PLANKEN ZU SCHNEIDEN UND DIE ARBEIT EINZUTEILEN, SONDERN ERWECKE IN DEN MENSCHEN DIE SEHNSUCHT NACH DEM WEITEN ENDLOSEN MEER.

Nach diesem gedanklichen Ausflug zum Erreichen großer Ziele gebe ich Ihnen nun ein paar Tipps mit auf den Weg, mit denen ich in einigen Projekten gute Erfahrungen

gemacht habe. Schauen wir uns nun also meinen »Think first then code and test«-Ansatz an. Wichtig ist mir, immer den gesunden Menschenverstand einzusetzen und in etwa nach folgenden Schritten vorzugehen:

1. **Verständnis** – Versuchen Sie das zu lösende Problem möglichst gut zu verstehen. Hierbei können etwa Anforderungsdokumente, Pflichtenhefte und insbesondere auch persönliche Gespräche mit den Kunden helfen.
2. **Grobentwurf und Prototpying** – Entwickeln Sie ein erstes Design und prüfen Sie Anforderungen, z. B. indem ein GUI-Prototyp entweder auf Papier oder mithilfe von Mock-up-Tools entsteht. Das hilft, mögliche Schwachpunkte oder fehlerhafte Annahmen aufzudecken.
3. **Refinement** – Klären Sie Fragen mit geeigneten Ansprechpartnern oder mithilfe von Dokumenten.
4. **Zerteilung** – Zerteilen Sie das System in kleinere, überschau- und beherrschbare Einzelbausteine und legen Sie die jeweiligen Verantwortlichkeiten fest.
5. **Implementierung und Test** – Nun erst beginnt die Entwicklung (auch die der Tests). Man kann mit einem ersten prototypischen Entwurf oder dem Erstellen von Testfällen anfangen. Zwischenzeitlich kann man auf Stub-Klassen oder Mocks zurückgreifen, wenn die konkrete Funktionalität noch nicht vollständig entwickelt ist. Beim Erstellen der Basisbausteine sorgt man mithilfe geeigneter Unit Tests dafür, dass diese im Idealfall fehlerfrei sind. Das ist einfacher möglich als bei komplexen Programmbausteinen, da die Basisbausteine oftmals nur wenige Eingabeparameter haben und nur einfache Dinge ausführen. Eine Spezifikation und Überprüfung fällt dann relativ leicht. Dazu haben wir bereits Unit Tests, Äquivalenzklassen, Mocking und parametrierte Tests als gutes Handwerkszeug kennengelernt.
6. **Zusammenfügen** – Fügen Sie die kleineren Komponenten zu größeren Einheiten zusammen. Entwickeln Sie die Interaktionen und testen Sie diese. Da man hier auf einer eher logischen Ebene arbeitet, fällt auch das Erstellen von Testfällen nicht mehr so schwer. Dabei verschwimmt die Grenze zwischen Unit Test, Integrationstest und Systemtest.
7. **Release und Review** – Stellen Sie einen solchen (Zwischen-)Stand dem Kunden vor und holen Sie Feedback ein. Nutzen Sie dieses zum Feintuning für die nächsten Arbeiten.

Die obigen Schritte werden wiederholt, bis das gewünschte Resultat erzielt wurde. Danach kann man sich über einen zufriedenen Kunden freuen, der die Software gerne benutzt, da sie von guter Qualität und auf seine Bedürfnisse zugeschnitten ist.

VII Anhang

A Grundlagen zur Java Virtual Machine

Dieser Anhang stellt einige Grundlagen zur Java Virtual Machine und zur Ausführung von Java-Programmen vor.

A.1 Wissenswertes rund um die Java Virtual Machine

Java ist eine Programmiersprache, die in einer Laufzeitumgebung, der ***Java Virtual Machine*** (***JVM***), ausgeführt wird. Im Gegensatz zu anderen Programmiersprachen, etwa C++, wird der Sourcecode beim Kompilieren mit dem Java-Compiler `javac` nicht in die Maschinensprache des jeweiligen Computers übersetzt, sondern in eine Zwischensprache, den plattformunabhängigen ***Bytecode***. Der Name rührt daher, dass die Instruktionen in Form von Bytes codiert sind. Dieser Bytecode wird nicht direkt vom Prozessor des Rechners ausgeführt. Stattdessen handelt es sich beim Bytecode um Befehle für einen speziellen virtuellen Computer, nämlich für die Java Virtual Machine. Diese stellt einen Computer im Computer dar und ermöglicht so eine Abstraktion von der darunterliegenden Hardware. Die Plattformunabhängigkeit von Java wird dadurch erreicht, dass für jedes Betriebssystem eine eigenständige JVM existiert, die Befehle im Bytecode-Format ausführt. Abbildung A-1 deutet den prinzipiellen Ablauf an.

Abbildung A-1 *Ablauf beim Kompilieren und Ausführen eines Java-Programms*

A.1.1 Ausführung eines Java-Programms

Für die Ausführung eines Java-Programms haben früher die ersten JVMs den Bytecode Instruktion für Instruktion interpretiert und abgearbeitet. Aufgrund der durchaus effizienten Arbeitsweise des Bytecode-Interpreters konnte damit bereits eine einigermaßen akzeptable Ausführungsgeschwindigkeit erzielt werden, die jedoch deutlich unter

der Geschwindigkeit kompilierter Programme lag. Als Folge davon haben die Ausführungszeiten dieser ersten JVMs lange Zeit das Gerücht genährt, Java-Programme würden (zu) langsam ablaufen. Kompilierte C++-Programme, die in die jeweilige Maschinensprache des Zielrechners übersetzt werden, waren bis zum Erscheinen von JDK 1.4 bzw. JDK 5 performanter. In den letzten Jahren wurden aber immer leistungsfähigere JVMs entwickelt, sodass sich die Ausführungsgeschwindigkeit von Java-Programmen immer mehr derjenigen kompilierter C++-Programme angenähert hat.

Heutzutage sind JVMs also extrem leistungsfähig. Das wird unter anderem dadurch erreicht, dass während bzw. parallel zu der eigentlichen Programmausführung der Bytecode in Maschinensprache übersetzt wird. Man spricht von einem Just-in-Time-Compiler (kurz JIT). Würde man diese Transformation für den gesamten Bytecode durchführen, so wäre das allerdings recht aufwendig und nicht besonders zielführend: Für selten durchlaufene Programmteile wiegt der erzielte Geschwindigkeitsgewinn bei der Ausführung der kompilierten Anweisungen nicht den zeitlichen Aufwand zur Transformation auf. Schlimmer noch: Diese Form der Optimierung ist manchmal sogar kontraproduktiv und langsamer als eine Ausführung per Interpreter. Aktuelle JVMs nutzen daher eine intelligentere Vorgehensweise bei der Programmausführung, nämlich das sogenannte Hotspot-Optimierungsverfahren.[1] Hierbei werden die häufig durchlaufenen Programmteile (die Hotspots) erkannt und nur diese kompiliert und optimiert. Abschnitt 22.1.3 geht auf einige Optimierungen im Detail ein.

A.1.2 Speicherverwaltung und Classloading

Um Java-Programme möglichst robust zu machen, wurden verschiedene Sicherheitsmechanismen in die Sprache integriert. Insbesondere wurde die Speicherverwaltung so gestaltet, dass man sich als Entwickler kaum darum kümmern muss: Der per `new` angeforderte Speicher für Objekte muss nicht explizit freigegeben werden. Das wird stattdessen automatisch durch eine spezielle Komponente der JVM, den ***Garbage Collector***, erledigt. Fehler durch zu früh oder mehrmals freigegebene Speicherbereiche sind damit ausgeschlossen. Allerdings verbleibt das Problem von nicht freigegebenem Speicher. Derartige ***Memory Leaks*** sind in Java im Vergleich zu Sprachen mit manuellem Speichermanagement eher selten, da die JVM diese automatisch sehr zuverlässig verhindert. Es gibt jedoch Spezialfälle, für die dies nicht gilt, nämlich genau dann, wenn nicht alle Referenzen auf Objekte korrekt freigegeben werden, etwa weil eine andere Programmkomponente eine Referenz weiterhin speichert. Details zur Garbage Collection beschreibt Abschnitt 12.4.

[1] Die Ausführungsmodi der JVM lassen sich über JVM-Aufrufparameter steuern: `-Xint` aktiviert den Interpreter-Modus. Ohne diese Angabe erfolgt eine Ausführung im Hotspot-Modus.

Achtung: Speicherverwaltung

Zwar befreit der Garbage Collector den Entwickler von der expliziten Speicherverwaltung und vermeidet mögliche Fehler einer manuellen Speicherfreigabe. Allerdings existieren verschiedene Varianten der Garbage Collection, die sich auf die Performance auswirken und bei Bedarf mit Bedacht gewählt werden sollten. Bei nicht zufriedenstellender Performance ist die Analyse der Garbage-Collection-Vorgänge mithilfe eines geeigneten Tools, etwa dem zuvor im Buch vorgestellten VisualVM (vgl. Abschnitt 22.1.4), sehr hilfreich, um mögliche Probleme aufspüren und beheben zu können.

Ein Schutz vor Fehlern beim Speicherzugriff wird durch verschiedene Sicherheitsmechanismen erreicht. Zum einen kann man mit Referenzen nur auf Objekte zugreifen, nicht aber mit dem Referenzwert rechnen, wie dies bei einigen anderen Sprachen möglich ist. Auch Bereichsüberschreitungen bei Array-Zugriffen werden von der JVM verhindert – als Folge werden automatisch `IndexOutOfBoundsException`s ausgelöst. Diese Sicherheitsmechanismen verhindern sowohl das versehentliche als auch das mutwillige Auslesen oder Beschreiben von Speicherbereichen, die eigentlich nicht adressierbar sein sollten. Selbst beim Auftreten derartiger Zugriffsprobleme verbleibt die JVM in einem definierten, arbeitsfähigen Zustand, und man kann kontrolliert über Exception Handling auf die Fehlersituation reagieren. Im besten Fall existiert dazu ein `catch`-Block zur Fehlerbehandlung. Dieser wird durch die JVM angesprungen, sodass die dortigen Anweisungen ausgeführt werden. Eine Fehlersituation bewirkt demnach nur, dass die Anweisungen, die der Exception auslösenden Programmstelle folgen, nicht mehr ausgeführt werden und stattdessen die Ausführung des Programms mit dem umgebenden `catch`-Block fortgesetzt wird. Findet sich kein passender `catch`-Block zur Fehlerbehandlung, d. h., wird die Exception nicht im Programm behandelt, so führt dies zu einem Abbruch aktiver Threads oder sogar des Programms, jedoch nicht zu einem Absturz der JVM.

Sicherheit und Classloading

Beim Start einer JVM ist immer eine Klasse anzugeben, deren `main()`-Methode ausgeführt werden soll, wie hier für eine Klasse `MyClass` gezeigt:

```
java MyClass
```

Zunächst muss der Bytecode der entsprechenden Klasse in die JVM geladen werden. Dazu dient eine Instanz eines sogenannten `ClassLoader`s, der selbst eine Java-Klasse ist. Diese spezielle Klasse lädt die Klassendateien (`.class`-Dateien). Was so alles an Klassen geladen wird, wenn Sie ein Java-Programm starten, sehen Sie durch Angabe der JVM-Option `-verbose` beim Aufruf:

```
java -verbose MyClass
```

Anhand der Ausgabe erkennt man, dass zunächst die wichtigsten Interfaces und Klassen des JDKs geladen werden:

```
[0.011s][info][class,load] opened: /Library/Java/JavaVirtualMachines/jdk-11.0.2.
      jdk/Contents/Home/lib/modules
[0.031s][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
[0.033s][info][class,load] java.lang.CharSequence source: jrt:/java.base
[0.033s][info][class,load] java.lang.String source: jrt:/java.base
[0.034s][info][class,load] java.lang.reflect.AnnotatedElement source: jrt:/java.
      base
[0.035s][info][class,load] java.lang.reflect.GenericDeclaration source: jrt:/
      java.base
[0.035s][info][class,load] java.lang.reflect.Type source: jrt:/java.base
[0.035s][info][class,load] java.lang.Class source: jrt:/java.base
[0.035s][info][class,load] java.lang.Cloneable source: jrt:/java.base
[0.013s][info][class,load] java.lang.ClassLoader source: jrt:/java.base
...
```

Bei diesen Ladevorgängen der Klassendateien finden einige Prüfungen statt, um zu verhindern, dass Systemklassen verändert werden – insbesondere der `ClassLoader` selbst. Anschließend wird durch die Komponente ***Bytecode Verifier*** sichergestellt, dass keine ungültigen Bytecode-Instruktionen in der `.class`-Datei enthalten sind. Derartige Instruktionen könnten durch Übertragungsfehler oder mutwillige Veränderungen am Bytecode entstehen. Beispielsweise besitzt die JVM den Befehl `iadd` zum Addieren von `int`-Zahlen. Würde dieser mit zwei `float`-Werten ausgeführt, käme es zu Problemen in der JVM. Der Bytecode Verifier prüft nun den gesamten Bytecode nach derartigen Verstößen. Wird ein solcher entdeckt, so wird ein `java.lang.VerifyError` ausgelöst und das Programm beendet. Damit wird verhindert, dass Instruktionen ausgeführt werden, die möglicherweise eine Fehlfunktion der JVM auslösen könnten.

Werden Klassen per Netzwerk oder aus anderen potenziell gefährlichen Quellen geladen, so ist es aus Sicherheitsgründen zum Schutz vor externen Angriffen wünschenswert, dass für diese Klassen eingeschränkte Rechte gelten. Genau das wird innerhalb der JVM durch eine Instanz der Klasse `SecurityManager` sichergestellt. Dieser kann einem ausgeführten Programm (bzw. dessen Klassen) zur Laufzeit gewisse Aktionen erlauben oder verbieten. Beispielsweise sind für derart geladene Klassen standardmäßig Dateisystemoperationen verboten. Das ist sinnvoll, um den Rechner, der das Programm ausführt, vor Angriffen aus dem Netz zu schützen.

Literaturverzeichnis

[1] Scott W. Ambler. *The Elements of Java Style*. Cambridge University Press, Cambridge, Mass., 1995.

[2] Ken Arnold, James Gosling und David Holmes. *The Java Programming Language*. Addison-Wesley, 4. Auflage, 2006.

[3] Jon Bentley. *Perlen der Programmierkunst. Programming Pearls*. Addison-Wesley, 2000.

[4] Cedric Beust und Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley, 2008.

[5] Joshua Bloch. *Effective Java*. Addison-Wesley, 2001.

[6] Joshua Bloch. *Effective Java*. Addison-Wesley, 2. Auflage, 2005.

[7] Joshua Bloch. *Effective Java*. Addison-Wesley, 3. Auflage, 2017.

[8] Joshua Bloch und Neil Gafter. *Java Puzzlers*. Addison-Wesley, 2005.

[9] Shyan R. Chidamber und Chris F. Kemerer. *A Metrics Suite for Object Oriented Design*. *IEEE Transactions on Software Engineering*, 20:476–493, 1994.

[10] Mike Clark. *Pragmatisch Programmieren: Projekt-Automatisierung*. Hanser Verlag, 2006.

[11] Peter Coad und Mark Mayfield. *Design mit Java*. Prentice Hall, 2. Auflage, 1999.

[12] Ben Collins-Sussman, Brian W. Fitzpatrick und C. Michael Pilato. *Versionskontrolle mit Subversion*. O'Reilly, 2. Auflage, 2006.

[13] James O Coplien. *Why Most Unit Testing is Waste*. *http://www.rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf*, 2014.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2. Auflage, 2001.

[15] Edsger W. Dijkstra. *The humble programmer*. *Communications of the ACM*, 15(10):859–866, 1994.

[16] Friedrich Esser. *Java 2 – Designmuster und Zertifizierungswissen*. Galileo Computing, 2001.

[17] Friedrich Esser. *Java 6 Core Techniken*. Oldenbourg, 2008.

[18] Michael Feathers. *Working Effectively With Legacy Code*. Prentice Hall, 2007.

[19] Ira R. Forman und Nate Forman. *Java Reflection in Action*. Manning, 2005.

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[21] Eric Freeman, Elizabeth Freeman, Kathy Sierra und Bert Bates. *Head First Design Patterns*. O'Reilly, 2004.
[22] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns – Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
[23] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996.
[24] Brian Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.
[25] Pete Goodliffe. *Code Craft: The Practice of Writing Excellent Code*. No Starch Press, 2007.
[26] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3. Auflage, 2005.
[27] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.
[28] Jörg Hettel und Manh Tien Tran. *Nebenläufige Programmierung mit Java: Konzepte und Programmiermodelle für Multicore-Systeme*. dpunkt.verlag, 2016.
[29] Ron Hitchens. *Java NIO*. O'Reilly, 2002.
[30] Allen Holub. *Taming Java Threads*. Apress, 2000.
[31] Allen Holub. *Holub on Patterns*. Apress, 2004.
[32] Cay S. Horstmann und Gary Cornell. *Core Java 2 – Band 1 Grundlagen*. Addison-Wesley, 7. Auflage, 2005.
[33] Cay S. Horstmann und Gary Cornell. *Core Java 2 – Band 2 Expertenwissen*. Addison-Wesley, 7. Auflage, 2005.
[34] Andrew Hunt und David Thomas. *Der Pragmatische Programmierer*. Hanser Verlag, 2003.
[35] Andrew Hunt und David Thomas. *Unit-Tests mit JUnit*. Hanser Verlag, 2004.
[36] Andrew Hunt und David Thomas. *Pragmatisch Programmieren: Versionsverwaltung mit CVS*. Hanser Verlag, 2005.
[37] Michael Hüttermann. *Agile Java-Entwicklung in der Praxis*. O'Reilly, 2008.
[38] Michael Inden. *Java 8 – Die Neuerungen*. dpunkt.verlag, 2. Auflage, 2015.
[39] Michael Inden. *Der Java-Profi: Persistenzlösungen und REST-Services*. dpunkt.verlag, 2016.
[40] Michael Inden. *Java 9 – Die Neuerungen*. dpunkt.verlag, 2017.
[41] Michael Inden. *Java – die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen*. dpunkt.verlag, 2020.
[42] Michael Inden. *Java Challenge*. dpunkt.verlag, 2020.
[43] Xiaoping Jia. *Object-Oriented Software Development Using Java*. Addison-Wesley, 2. Auflage, 2002.

[44] Tomasz Kaczanowski. *Practical Unit Testing with JUnit and Mockito*. kaczanowscy.pl, 2013.
[45] Joshua Kerievsky. *Refactoring To Patterns*. Addison-Wesley, 2005.
[46] Donald E. Knuth. *Structured Programming with go to Statements*. *ACM Journal Computing Surveys*, 6(4):268, Dezember 1974.
[47] Lasse Koskela. *Effective Unit Testing: A Guide for Java Developers*. Manning, 2013.
[48] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 2. Auflage, 2000.
[49] Johannes Link. *Softwaretests mit JUnit*. dpunkt.verlag, 2. Auflage, 2005.
[50] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
[51] Mike Mason. *Pragmatic Version Control Using Subversion*. Pragmatic Programmers, 2. Auflage, 2006.
[52] Steven J. Metsker. *Design Patterns Java Workbook*. Addison-Wesley, 2002.
[53] Khalid A. Mughal und Rolf W. Rasmussen. *A Programmers's Guide to Java SCJP Certification*. Addison-Wesley, 3. Auflage, 2009.
[54] Benjamin Muschko. *Gradle In Action*. Manning, 2014.
[55] Maurice Naftalin und Philip Wadler. *Java Generics and Collections*. O'Reilly, 2007.
[56] Johannes Nowak. *Fortgeschrittene Programmierung mit Java 5*. dpunkt.verlag, 2005.
[57] Scott Oaks und Henry Wong. *Java Threads*. O'Reilly, 3. Auflage, 2004.
[58] Rainer Oechsle. *Parallele und verteilte Anwendungen in Java*. Hanser Verlag, 2. Auflage, 2007.
[59] Rainer Oechsle. *Parallele und verteilte Anwendungen in Java*. Hanser Verlag, 5. Auflage, 2018.
[60] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly, 2009.
[61] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. Wiley, 2000.
[62] René Preißel und Bjørn Stachmann. *Git*. dpunkt.verlag, 2. Auflage, 2014.
[63] Stefan Roock und Martin Lippert. *Refactorings in großen Softwareprojekten*. dpunkt.verlag, 2004.
[64] Hendrik Schreiber. *Performant Java programmieren*. Addison-Wesley, 2002.
[65] Robert Sedgewick. *Algorithmen*. Addison-Wesley, 1992.
[66] Alan Shalloway und James R. Trott. *Design Patterns Explained*. Addison-Wesley, 2. Auflage, 2005.
[67] Jack Shirazi. *Java Performance Tuning*. O'Reilly, 2. Auflage, 2003.
[68] Kathy Sierra und Bert Bates. *SCJP – Sun Certified Programmer & Developer for Java 2*. Osborne, 2003.
[69] Kathy Sierra und Bert Bates. *SCJP – Sun Certified Programmer for Java 6 Study Guide*. McGraw-Hill, 2008.

[70] Bernhard Stephan. *Einstieg in Java 6 – Sonderausgabe: Verständliche und umfassende Einführung*. Galileo Computing, 3. Auflage, 2008.

[71] Venkat Subramaniam. *Programming Groovy 2: Dynamic Productivity for the Java Developer*. O'Reilly, 2013.

[72] Uwe Vigenschow und Björn Schneider. *Soft Skills für Softwareentwickler*. dpunkt.verlag, 2006.

[73] Steve Wilson und Jeff Kesselman. *Java Platform Performance – Strategies and Tactics*. Addison-Wesley, 2000.

Index

@Deprecated, 113, 318, 533, 884
    forRemoval, 318
    since, 318
@Deprecated-Annotation, 318
@Override, 225
@TODO, 916
@deprecated, 113, 318, 533, 884

Abarbeitung
    iterative, 214
    parallele, 476
    sequenzielle, 214, 476, 487
Abbildungstabelle, 372
Abhängigkeit
    zwischen Modulen, 845
Ableitung, 378
Ableitungshierarchie, 887, 907
    von Exceptions, 304
Abnahmetest, 1145
AbstractMap<K,V>, 372, 377
AbstractSet<E>, 348
Abstraktion, 1121
Abstraktionsebene, 993, 1122
    einheitliche, 994, 1104
    Implementierungsebene, 1105
    unterschiedliche, 1104
accept(), 210, 211
Access Control Proxy, 1085
Accessibility, 831
Accessor, 89
ActionListener, 203
Adapter, 1047
adjustInto(), 517, 519
Aggregation, 91, 378
Akzeptanztest, 1145
Algorithmus, 1292
allMatch(), 457, 470
Analyse
    dynamische, 1128
    statische, 1128
and(), 215, 216
Annotation, 40, 700
    @Deprecated, 113, 533, 884, 1006, 1115
    @Override, 225
    Override, 380
    auslesen, 706
    Auto-Complete-, 705
    Definition einer eigenen, 703
    Meta-, 705
    Standard-, 701
Annotation Processor, 701
Annotations, 681
Anomalie
    MIN_VALUE, 246
anonyme innere Klasse, 275
Ant, 63
Anti-Pattern, 1023
    Beobachter, 1095
    Observer, 1095
    Singleton, 1039
anyMatch(), 457, 470
AOP, 1046
Apache Commons, 530
Apache Commons CLI, 566
Apache Commons Configuration, 579
API, 102, 545, 1153
    Design, 1153
    fremdes, 571
API-Design, 1153
API-Problem, 545
Appender, 554
Application Programming Interface, 102
Applikationstest, 1145
apply(), 210, 218, 345
Äquivalenzklasse, 1164
Äquivalenzklassentest, 931, 1163
Arbeitsumgebung, 13
Architekturproblem, 1128
ARM, 308, 314
Array, 327, 330

Anpassung der Größe, 331
Kapazität, 331
ArrayBlockingQueue<E>, 639
ArrayIndexOutOfBoundsException, 894, 921
ArrayList<E>, 228, 338, 340, 1294, 1296
add(), 341, 342
get(), 341
Größenanpassung, 342
Kapazität, 340
remove(), 342
size(), 340
Speicherverbrauch, 342
trimToSize(), 343
Arrays, 226, 408, 454
asList(), 226, 406
binarySearch(), 394
compare(), 408
deepEquals(), 444
deepHashCode(), 444
deepToString(), 444
equals(), 408, 443, 444
hashCode(), 444
mismatch(), 408
parallelStream(), 480
stream(), 454
toString(), 226, 444
ArrayStoreException, 180, 419
Artefakt, 65
asLongStream(), 455
AspectJ, 1046
aspektorientierte Programmierung, 1046
Assert, 38
Assertion, 298, 314
-da, 316
-ea, 316
aktivieren, 316
deaktivieren, 316
AssertionError, 314
Assertions, 40
AssertJ, 1241
Assoziation, 91
Atomarität, 619, 620
AtomicInteger, 621
AtomicLong, 621
Attribut, 86
Präfix, 1109
Typpräfix, 1109
Aufrufkette, 441
mit Fallback-Strategie, 441
Aufzählung, 154
Ausdruck
regulärer, 266
Ausfallsicherheit, 670
Auslieferung, 18
äußere Qualität, 1150
Austauschbarkeit, 1121
Auto-Boxing, 247
Wissenswertes zu, 250
Auto-Unboxing, 247, 250
Wissenswertes zu, 250
AutoCloseable, 309
Automatic Module, 863
Bottom-up-Migration, 865
Migration, 868
Wissenswertes, 864
Automatic Resource Management, 308, 314
average(), 434, 469

Backpressure, 670
Backup, 22
Balancierung, 366
Basisklasse, 910
abstrakte, 128, 129
Baum, 1055
binärer, 365
Blatt, 365
Tiefe, 365
Wurzel, 365
Bedingung
boolesche, 215
behaves-like-Beziehung, 87
Beobachter, 996, 1087
als Anti-Pattern, 1095
Bereich
kritischer, 595
Betriebsblindheit, 1147
between(), 504, 506
BigInteger, 961
binärer Baum, 365
Binärsuche, 394, 1290
BinaryOperator, 473, 489
Binden
dynamisches, 96, 906, 938
Blackbox-Test, 1146
Blatt, 365, 1055
BlockingQueue<E>, 614, 639
offer(), 639
poll(), 639
put(), 639
take(), 639
body(), 738

BodyHandlers
ofString(), 738
Boilerplate-Code, 201, 401
Boolean, 247
Bottom-up-Migration, 865
Automatic Module, 865
Fallstrick, 867
Named Modules, 865
boxed(), 455
Boxing, 247
Branch, 21, 26
Breakpoint, 47
Bucket, 351
BufferedInputStream, 280
BufferedSubscription, 672
Build, 63
inkrementeller, 74
Builder, 1032
Bulk Operations
on Collections, 453
Bulk-Operation, 332
Business-Methode, 90, 104, 986, 988, 1122
Auswirkung auf Beobachter-Muster, 1094
Busy Waiting, 608
Bytecode, 1353
Bytecode Verifier, 1356

Cache, 373
Integer, 251
Long, 251
LRU-, 1326
Thread-lokaler, 595
Cache-Kohärenz, 619, 1329
Calendar, 500
Call-by-Reference, 97
Call-by-Value, 97
Callable, 647, 659
Callback-Interface, 215
CamelCase-Notation, 1159
CamelCase-Schreibweise, 1108
can-act-like, 140
can-act-like-Beziehung, 87, 125
cancel(), 673
CAS, 621
case
rückwärtskompatibel, 783
Cast, 96, 886
ClassCastException, 887
CBO, 1131
Character, 247
isDigit(), 954
isWhitespace(), 270
CharBuffer, 291
chars(), 454
CharSequence, 256, 490
Charset, 283
availableCharsets(), 284
Cp1252, 283
Cp850, 283
forName(), 284
ISO-8859-1, 284
UTF-16, 284
UTF-8, 284
CharsetDecoder, 283
CharsetEncoder, 283
Checked Exception, 304, 314, 920
Checkstyle, 1133
ChronoUnit, 498, 506
between(), 506
Class<T>, 221
ClassCastException, 348, 887
CLASSPATH, 18
Client, 123
Client-Server-Kommunikation, 311
Clock, 514
fixed(), 515
Clustering, 353
Codereview, 1106, 1110, 1265
Meeting, 1265
Probleme, 1267
psychologische Aspekte, 1266
Tipps, 1267
Codereview-Meeting, 1265
Coding Conventions, 962, 1101, 1106
Akzeptanz, 1106
Formatierung, 1107
Namensgebung, 1107
CollationKey, 763
Collator, 403, 759
getInstance(), 403
collect(), 457, 474, 482
Collection, 209, 345
Mengenoperation, 333
unveränderliche Kopien, 390
Collection-Factory-Methode, 387–389
Collection-Literal, 78, 386, 389
Collection<E>, 328, 332, 449, 450, 1301
add(), 228, 332
addAll(), 332
contains(), 332
containsAll(), 332

isEmpty(), 332
iterator(), 334
next(), 335
parallelStream(), 454
remove(), 332, 336
removeAll(), 332
removeIf(), 333
retainAll(), 332
size(), 332
stream(), 454
Collections
binarySearch(), 394
emptyList(), 410
emptyMap(), 410
emptySet(), 410
frequency(), 415
max(), 415
min(), 415
nCopies(), 415
replaceAll(), 416
shuffle(), 416
singleton(), 410
singletonList(), 410
singletonMap(), 410
unmodifiableList(), 338, 413
Collector, 468
Collectors, 482
counting(), 474
groupingBy(), 474
joining(), 474, 482
partitioningBy(), 474
summarizingInt(), 803
teeing(), 802
toCollection(), 468, 469
toList(), 468, 469
toUnmodifiableList(), 484
toUnmodifiableMap(), 484
toUnmodifiableSet(), 484
Command, 1077
CommandLineParser
parse(), 567
Commons CLI, 566
Commons Configuration, 579
Compact Strings, 270
Comparable<T>, 359, 360, 402
compareTo(), 360
Comparator<T>, 200–202, 360, 363, 401–403, 457
compare(), 364
compareTo(), 401, 403
comparing(), 402
comparingDouble(), 402
comparingInt(), 402, 403
comparingLong(), 402
Hintereinanderschaltung, 403
naturalOrder(), 404
null-Werte, 405
nullsFirst(), 405
nullsLast(), 405
primitive Typen, 403
reversed(), 404
reverseOrder(), 404
thenComparing(), 402, 403
thenComparingDouble(), 402
thenComparingInt(), 402
thenComparingLong(), 402
Comparators
naturalOrder(), 402
reversed(), 402
reverseOrder(), 402
compare(), 201, 202, 408
Compare-and-Swap-Operation, 621
compareTo(), 401, 403
comparing(), 402
comparingInt(), 403
CompletableFuture, 659, 666
CompletableFuture<T>, 659, 660
completeAsync(), 665
completeOnTimeout(), 665
exceptionally(), 660
failedFuture(), 665
get(), 660
orTimeout(), 665
supplyAsync(), 660, 662
thenAccept(), 665
thenApply(), 660
thenApplyAsync(), 662
thenCombine(), 662
CompletableFuture<t>
thenAccept(), 660
completeAsync(), 665
completeOnTimeout(), 665
Compound Key, 788
computeIfAbsent(), 384
computeIfPresent(), 384
ConcurrentHashMap<K,V>, 382, 636, 637, 1298
ConcurrentModificationException, 336, 412, 635
ConcurrentSkipListMap<K,V>, 329, 637, 1299
ConcurrentSkipListSet<E>, 329, 1299

Condition, 604
  await(), 618
  signal(), 618
  signalAll(), 618
Conditional-Operator, 1126
Constructor Injection, 1208
Consumer, 678
Consumer<T>, 209, 210
  accept(), 211
Container
  hashbasierte, 350
Containerklasse, 327
Continuous Integration, 65
Convenience-Methode, 891
Convention over Configuration, 65
Copy-Paste, 113, 890, 891, 930
  Probleme durch, 890
  Wiederverwendung, 91
Copy-Paste-Ansatz, 92
copyOf(), 391
CopyOnWriteArrayList<E>, 636
CopyOnWriteArraySet<E>, 636
count(), 457, 469
counting(), 474
Cp1252, 283
Cp850, 283
CPU-bound, 1275
CPU-bound-Optimierungen, 1338
Cross-Cutting Concern, 1045, 1046
currentTimeMillis(), 514, 809
CVS, 13
Cyclomatic Complexity, 1130

Daemon-Thread, 626
DAO, 1111
Data Access Object, 1111
Data Transfer Object, 160, 788
DataInputStream, 289
  readLong(), 289
  readUTF(), 289
DataOutputStream, 289
  writeLong(), 289
  writeUTF(), 289
Date, 497, 526
Date and Time API, 497
DateFormat, 752
Datenkapselung, 111, 979, 986
Datensicherung, 22
Datenstruktur, 1292
  Array, 330
  Baum, 1292
  Einsatz geeigneter, 1292
  Liste, 338
  Menge, 346
Datentyp, 87
  primitiver, 87
DateTimeFormatter, 516, 577, 579
DayOfWeek, 498, 500
dayOfWeekInMonth(), 519
Deadlock, 582, 602, 603
Debugger, 46
Debugging, 46
  Java Debug Wire Protocol, 51
  JDWP, 51
Debuggingzwecke, 456
Decoder, 283
Decorator, 1049
Deep Copy, 148
default, 21, 31
Defaultmethode, 208
  API-Erweiterung, 208
  forEach(), 209, 334
  Rückwärtskompatibilität, 208
  sort(), 209
  Standardverhalten, 211
DefaultMutableTreeNode, 1055
  hasChildren(), 1055
  isLeaf(), 1055
Definition, 87
Deklaration, 87
Delayed, 640
Delayed Exception, 928
DelayQueue<E>, 640
Delegation, 114, 1048
Dependency Injection, 177, 1082, 1208
Dependency Inversion Principle, 167, 175, 1011
Deployment, 55
Deque<E>, 329, 447, 450
Deserialisierung, 708
Design by Contract, 115, 1118
Designpattern, 1023
Designproblem, 1128
Diamond Operator, 186, 275, 321
Dictionary, 369
Differenzmenge, 333
DIP, 167, 175, 1011
distinct(), 457, 465
DIT, 1131
divide and conquer, 652
Domain, 1145
Double, 240

doubleToLongBits(), 254
longBitsToDouble(), 254
parseDouble(), 249
Double Checked Locking, 1039
DoubleStream, 455
boxed(), 455
DoubleSummaryStatistics, 470
Down Cast, 96, 146, 887
dropWhile(), 481, 482
DRY-Prinzip, 993, 1014
DTO, 788, 1301
Hilfsmethoden, 161
Dummy, 1204
Duplikatbehandlung, 389
Duplikaten
Besonderheiten bei, 388
Duplikatfreiheit, 389
Duplikatsprüfung, 389
Duration, 502, 504, 505, 510
between(), 504
ofDays(), 504
Dynamic Binding, 96
dynamische Analyse, 1128
dynamisches Binden, 938

EclEmma, 1252
Eclipse, 13, 14
Refactoring
Encapsulate Field, 980
Extract Interface, 989
Extract Local Variable, 941
Extract Method, 993
effectively final, 219, 274, 308
Eiffel, 115
Eigenschaften
orthogonale, 139
unabhängige, 139
Einfügereihenfolge, 387
Einschwing-Effekt, 810
Empfänger, 670
empty(), 433, 440
Encoder, 283
Entropie, 1103
Entwicklungsumgebung
integrierte, 13
Entwurfsmuster, 1023
Adapter, 1047
Builder, 1032
Command, 1077
Decorator, 139, 1049
Erbauer, 1032
Erzeugungsmethode, 934, 936, 1026
Fabrikmethode, 753, 913, 1029, 1296
Fassade, 1044
Iterator, 1056
Kompositum, 1052, 1292
Listener, 1086
Null-Objekt, 472, 1058, 1306
Observer, 1086
Prototyp, 1040
Proxy, 1084, 1307, 1309
Publisher/Subscriber, 1086
Schablonenmethode, 1061
Singleton, 572, 594, 1035
Strategie, 1065
Template-Methode, 1061
Value Object, 160
Entwurfsstil, 106
Enum
lokaler, 797
Enum-Muster, 154
EnumSet, 158
equals(), 408
Behandlung optionaler Attribute, 234
in Subklassen, 236
Kontrakt, 229
nullSafeEquals(), 235
StringBuffer, 263
StringBuilder, 263
Typische Fehler, 232
EqualsUtils
nullSafeEquals(), 235
equalTo(), 1238
Erbauer, 1032
Erreichbarkeit, 726
Erzeugungsmethode, 934, 936, 1026
Escapen, 268
Escaping Reference, 904
Event, 670
Exception, 298
Checked, 304, 305
IllegalArgumentException, 299
IllegalStateException, 299
in Threads, 627
NullPointerException, 299
printStackTrace(), 300
Unchecked, 304, 305
UnsupportedOperationException, 299
Exception Handling, 310
Besonderheiten, 306
Erleichterung, 920
Final Rethrow, 307

Multi Catch, 306
unspezifisches, 918
exceptionally(), 660
Execute-Around-Pattern, 731
Executor, 211, 642
Executors, 211
newCachedThreadPool(), 648
newFixedThreadPool(), 648
newScheduledThreadPool(), 648
newSingleThreadExecutor(), 648
ExecutorService, 647
submit(), 647
Explosion
kombinatorische, 138
Externalizable, 722
Extraktion, 487
Extreme Programming, 1147
Extremwerttest, 931

Fabrikmethode, 753, 913, 1029, 1296
fail-fast, 635
Fail-fast-Iterator, 336, 337, 340
failedFuture(), 665
Fall Through, 779
Fall-Through-Assertion, 317
Fallback-Strategie, 441
Fallstrick
Bottom-up-Migration, 867
var, 321
Fassade, 1044
FCFS, 447
Feedbackzyklus, 1174, 1177
Fehler, 1144
Fehlerbehandlung
offensive, 926
Fehlermaskierung, 1133, 1146
Fehlersuche, 47, 806
mit Debugger, 47
FIFO, 447
FIFO-Prinzip, 329
File, 276, 291
createNewFile(), 277
delete(), 277
exists(), 277
getAbsolutePath(), 276
getCanonicalPath(), 277
getName(), 276
isDirectory(), 277
isFile(), 277
list(), 277
listFiles(), 277
mkdir(), 277
mkdirs(), 277
FileFilter, 278
accept(), 278
FileInputStream, 280
FilenameFilter, 278
accept(), 278
FileOutputStream, 280
Files, 293
lines(), 271, 295
list(), 295
mismatch(), 801
readAllLines(), 295, 663
readString(), 296
write(), 295
writeString(), 296
filter(), 456, 458, 460, 472, 513
Filter-Map-Reduce, 485
im Einsatz, 488
Filter-Map-Reduce-Framework, 453
Filterbedingung, 663
final
effectively, 219
findAny(), 457, 471, 480, 481
findFirst(), 457, 471, 472, 481
First-Come-First-Serve, 447
First-In-First-Out, 447
firstDayOfMonth(), 518
firstDayOfNextMonth(), 518
firstDayOfNextYear(), 518
firstDayOfYear(), 518
firstInMonth(), 518
fixed(), 515
Flüchtigkeitsfehler, 389
flache Kopie, 148
flatMap(), 440, 456, 460, 475, 663
Float, 240
floatToIntBits(), 254
intBitsToFloat(), 254
parseFloat(), 249
Flow, 670, 671, 675
Fokussierung, 1163
for-Schleife, 484
forEach(), 209, 334, 457, 467, 480
forEachRemaining(), 337
FOREVER, 506
Fork-Join-Framework, 660
Format, 264, 754
Formatierung, 516
Ausnahmen, 1108
grundlegende Regeln, 1107

Formatter, 753
forRemoval, 318
freeMemory(), 1285
Fremdbibliothek
    Einbinden, 860
Füllgrad, 331, 357
Function<T,R>, 210, 218, 456, 459
    apply(), 210, 218, 345, 459
Functional Interface, 200
    besondere Methoden, 201
    Implementierung von, 201
FunctionalInterface
    Annotation, 201
Funktionalität
    veraltete, 318
Future, 659
    isDone(), 649

Gültigkeitsprüfung, 790
Garbage Collection, 89, 222, 681, 725
    alte Generation, 726
    Grundlagen zur, 725
    junge Generation, 726
Garbage Collector, 222, 342, 343, 725, 1120, 1354
    CMS, 727
    G1, 727
GC, 725
Geheimnisprinzip, 163
Generalisierung, 93, 1121
Generation
    alte, 726
    junge, 726
Generics, 85, 144, 184
Gesetz der Ähnlichkeit, 1103
Gesetz der Nähe, 1105
Gesetz von Demeter, 164
get(), 433, 660
getAvailableZoneIds(), 513
getOrDefault(), 383
getState(), 588
GMT, 513
Google Guava, 530, 860
Gradle, 13, 63
    Abhängigkeiten definieren, 76
    Builds mit, 70
    Code-Checker-Tool einbinden, 78
    eigene Tasks definieren, 77
    externe Abhängigkeit spezifizieren, 75
    IDE-Projekte erzeugen, 79
    Installation, 70
    JAR erstellen, 77
    Javadoc generieren, 77
    Multi-Project-Builds, 80
    Sourcen kompilieren, 73
    Unit Tests ausführen, 74
    verfügbare Tasks, 72
    Vorteile, 82
Graphen, 1055
Greenwich Mean Time, 513
GregorianCalendar, 526
groupingBy(), 474
Guava, 860

Hamcrest, 1237
    assertThat(), 1238
Happens-before, 624
    Antisymmetrisch, 625
    Irreflexiv, 625
    Transitiv, 625
Happens-before-Ordnung, 629, 630
has-a-Beziehung, 911
hashCode(), 356
    Kontrakt, 355
    Primzahl, 356
Hashcontainer
    Bucket, 351
    Füllgrad, 357
    Kollision, 357
    Load Factor, 357
HashMap<K,V>, 372, 636, 1297
HashSet<E>, 347, 348, 388, 1297
Hashtabelle, 352
Hashtable<K,V>, 573
HashUtils, 357
Hauptast, 21
HEAD, 21
HelpFormatter
    printHelp(), 568
heterogene Liste, 431
High-Level-Refactoring, 979
Hilfsmethoden, 161
Histogramm
    aufbereiten, 491
homogene Liste, 431
Hook, 1061
Hotspot-Optimierer, 811
HTTP/2-API, 737
HttpClient
    newHttpClient(), 738
HttpRequest
    newBuilder(), 738

HttpResponse
body(), 738
statusCode(), 738
Hyperassertions, 1234

I/O-bound, 1275
I/O-bound-Optimierungen, 1318
IDE, 13
Identität, 227
identity(), 218
Idiom
Null-sichere Vergleiche mit der Utility-Klasse Objects, 235
Performante Stringkonkatenation, 262
Prüfung auf inhaltliche Gleichheit, 263
Thread-sichere Iteration, 412
Traversierung von Collections mit dem Interface Iterator, 334
Warten auf eine Bedingung, 612
IEEE-754-Format, 254
ifPresent(), 433
ifPresentOrElse(), 434, 439
IllegalAccessException, 857
IllegalArgumentException, 1015
IllegalMonitorStateException, 599
unerwartete, 617
IllegalStateException, 337, 449, 463, 1013
Konfigurationsfehler, 925
IllegalThreadStateException, 585
Immutable Collections, 484
ImmutableCollections, 388, 389
Implementierungsphase, 1118
Implementierungsvererbung, 93, 114, 448, 570
indent(), 798
Indirektionen, 1046
Infinitest, 1247
Information Hiding, 89, 163
Initialisierung
korrekte, 1013
Initialisierungsprüfung
zentrale, 1014
Initializer
Instanz-, 907
Inkonsistenz
Vermeidung von, 389
innere Klasse, 272
innere Qualität, 1150
InputStream, 280, 281, 291
available(), 281
close(), 281
read(), 281, 282
readAllBytes(), 281
ready(), 282
transferTo(), 281
InputStreamReader, 283
Inspektion
von Verarbeitungsschritten, 463
Installation
Gradle, 70
instanceof, 229, 793
Instant, 497, 499, 501, 503–505, 509, 526
plus(), 501, 504
Instanz, 86
Instanz-Initializer, 907
Integer, 240
Cache, 251
parseInt(), 248, 249, 579, 960
Integrationstest, 1144, 1147
IntelliJ IDEA, 14
Interface, 86, 123
Angebot von Verhalten, 86, 123
Aufspalten eines, 990
Comparable<T>, 360
Einführen eines, 989
Erweiterung, 208
Kompatibilitätsproblem, 208
lokaler, 797
Namensgebung von, 126
Präfix, 126
Read-only-, 991, 992
Read-Write-, 992
statische Methoden, 211
Veröffentlichung, 208
Write-, 992
Interface Segregation Principle, 167, 173, 1011, 1122
Interleaving, 623
Intermediate Operation, 456
zustandsbehaftete, 465
zustandslose, 458
Internationalisierung, 403, 743
Interprozesskommunikation, 582
Interrupted-Exception, 592, 593
IntPredicate, 484
IntStream, 434, 454, 455, 481
asLongStream(), 455
average(), 434
boxed(), 455
chars(), 454
mapToObj(), 455
max(), 434

min(), 434
range(), 454
summaryStatistics(), 803
IntSummaryStatistics, 470
IntUnaryOperator, 484
Invariante, 115, 150
Invarianz, 179, 418, 419
IOException, 277
is-a-Beziehung, 93, 96, 125, 448, 911
isBlank(), 270
isLeap(), 501
ISO-8859-1, 284
ISP, 167, 173, 1011, 1122
Ist-Verhalten, 1143
isWhitespace(), 270
Iterable<T>, 209, 334, 1058
forEach(), 467
iterate(), 483, 484
Iteration, 214, 334
externe, 214
interne, 214, 215
Iterator, 214, 334, 1056
fail-fast-, 336, 337, 340, 635
weakly consistent, 638
Iterator<E>, 211, 334, 1058, 1301
ConcurrentModificationException, 336
forEachRemaining(), 337
hasNext(), 334
next(), 334–336
remove(), 211, 335, 337
UnsupportedOperationException, 335

JaCoCo, 1248
JAR, 55
erstellen, 77
modulares, 843
Java Debug Wire Protocol, 51
Java Virtual Machine, 1353
Java-Memory-Modell, 619
Javadoc, 1105, 1114
@author, 1115
@deprecated, 113, 533, 884, 1115
@inheritDoc, 1115
@param, 1115
@return, 1115
@see, 1115
@since, 1115
@throws, 1115
@version, 1115
Tag, 1114
Javadoc-Kommentar, 318
javap, 262
jdeps, 845
JDWP, 51
JIT-Compiler, 811
JLS, 317, 355
JMH, 808
dritter Benchmark, 817
erster Benchmark, 812
zweiter Benchmark, 815
JMM, 619
Atomarität, 619, 620
Reordering, 619, 622
Sichtbarkeit, 619
join(), 490
joining(), 474, 482
jshell, 521
JSR-310, 497
JUnit, 38, 1237
@AfterAll, 1161
@AfterEach, 1161
@BeforeAll, 1161
@BeforeEach, 1161
Annotation, 40
assertEquals(), 40, 1238
assertFalse(), 40, 955
Assertions, 40
assertNotNull(), 40
assertNotSame(), 40
assertNull(), 40
assertSame(), 40
assertThrows(), 41
assertTrue(), 40, 955
Extreme Programming, 1147
fail(), 41
TDD, 1147
Test-Driven Development, 1147
XP, 1147
JUnit-Framework, 40
Just-in-Time-Compiler, 1281
JVM, 1353

Kanten, 1055
Kapazität, 331, 352
Kapselung, 85, 89, 105, 111, 238, 986, 1131
bessere, 981
Keep It Clean, 1103
Keep It Human-Readable, 1102
Keep It Natural, 1102
Keep It Simple And Short, 1102
Key-Extractor, 402, 405

Klasse, 86
abgeleitete, 92
anonyme innere, 275
Basis-, 92
innere, 272
konkrete, 129
Ober-, 92
Sub-, 92
Super-, 92
Klassenhierarchie, 93, 139, 238
Klassenlänge
maximale, 1123
Klassenname
voll qualifizierter, 897
Knoten, 344, 1055
Kohäsion, 90, 105, 112, 1131, 1153
Kollaborateur, 1204
Kollision, 351, 352, 357, 1297
Kommentar
@TODO, 916
Überschrift-, 1113
Javadoc-, 318
Komparator, 363
Hintereinanderschaltung, 403
Kompatibilität, 860
Kompatibilitätsmodus, 860, 875
Kompiliertyp, 95, 938
Komponente, 826
Komponententest, 1144
Komposition, 91
Kompositum, 1052, 1292
Konsistenz, 229
Konstantensammlung, 154
Kontrakt
equals(), 229
hashCode(), 355
Kontravarianz, 179, 181, 422
Konzept
mathematisches, 388
Kopie
tiefe, 147, 151
Kopplung, 92, 1153
lose, 981
Kosteneinsparungen
durch Qualität, 1150
Kovarianz, 179, 418, 422
kritischer Bereich, 582, 595
Lock, 604
Synchronisationsobjekt, 597
synchronized, 597, 604
Kurzzeitgedächtnis, 1124
Lambda, 199, 201, 275, 576
als Parameter, 203
als Rückgabewert, 203
im Java-Typsystem, 200
Kurzschreibweisen, 202
Lambda-Ausdruck, 199
Last-In-First-Out, 447
lastDayOfMonth(), 518
lastDayOfYear(), 518
lastInMonth(), 518
Laufzeit, 487
Laufzeittyp, 95
Layout, 554
Lazy Init Proxy, 1085
Lazy Initialization, 880, 1037, 1302
Legacy-Code, 526, 1149
LIFO, 447
limit(), 457, 513
lines(), 271, 295
LinkedBlockingQueue<E>, 614, 639
LinkedHashMap<K,V>, 373
removeEldestEntry(), 374, 376
LinkedList<E>, 338, 343, 450, 1294, 1296
add(), 344
get(), 344
Größenanpassung, 344
Knoten, 343, 344
Node, 343
remove(), 344
Speicherverbrauch, 344
Liskov Substitution Principle, 167, 169
list(), 295
List<E>, 209, 328, 338, 345, 386
add(), 338, 448
copyOf(), 391
get(), 338, 448
indexOf(), 338, 448
isEmpty(), 448
lastIndexOf(), 338
listIterator(), 340
remove(), 338, 448
replaceAll(), 345
set(), 338
sort(), 209
subList(), 339
Liste, 338
heterogene, 431
homogene, 431
Listener, 1086
ListIterator<E>, 340
hasPrevious(), 340

nextIndex(), 340
previous(), 340
previousIndex(), 340
Literal, 154
Load Factor, 357
LOC, 1130
Local Variable Type Inference, 319, 322
LocalDate, 497, 499, 507, 508, 526
ofInstant(), 509
toEpochSecond(), 509
LocalDateTime, 499, 507, 511, 526
now(), 513
ofInstant(), 526
Locale, 745
getAvailableLocales(), 746
getCountry(), 746
getLanguage(), 746
LocalTime, 497, 499, 509, 526
Lock, 595, 604
Grundlagen, 604
lock(), 604
unlock(), 604
LoD, 164
Log-Level, 553
log4j, 552
Layout, 554
Log-Level, 553
Logger, 552
Logger, 552
Logging, 551
Logging-Framework, 551, 552, 1116
log4j, 552
Lokalität, 1276
Long, 240
Cache, 251
parseLong(), 249
LongStream, 455
boxed(), 455
LongSummaryStatistics, 470
Lookup-Map, 372
Lookup-Tabelle, 369
Lower Type Bound, 422
LRU-Cache, 375, 1326
LSP, 167, 169

Magic Number, 154, 563, 881, 1116, 1272
Magic String, 563, 571, 1116
Main-Branch, 26
main-Thread, 583, 626
Manifest-Datei, 58
map(), 456, 459, 461, 663
Map.Entry<K,V>, 369
Map<K,V>, 328, 329, 369, 381, 386
clear(), 370
computeIfAbsent(), 384
computeIfPresent(), 384
containsKey(), 370
containsValue(), 370
copyOf(), 391
entrySet(), 370
get(), 370
getOrDefault(), 383
größenbeschränkte, 374
isEmpty(), 370
keySet(), 370
merge(), 385
nach Wert sortieren, 492
put(), 370
putAll(), 370
putIfAbsent(), 383
remove(), 370
replace(), 383
size(), 370
values(), 370
mapToObj(), 455, 482
Marker-Interface, 153
Maschinenzeit, 497
master, 21, 31
Matcher, 1237
Maven, 13, 63
Vorteile, 82
Maven Central, 68
max(), 434, 457
McCabe, 1130
Membervariable, 86
Memoization, 1343
Memory Leak, 343, 1354
Memory-Analyse-Tool, 1286
Memory-bound, 1275
Memory-bound-Optimierungen, 1325
Menge, 346, 388
Mengenoperation, 333
Merge, 21
merge(), 385
Message, 670
MessageFormat, 752
Messaging, 582
Metadaten, 683
Metainformation, 683, 699
Method Injection, 1208
Methode, 86
abstrakte, 86, 125

Ausgangspunkt, 1117
Methodenlänge, 167
maximale, 1123
Methodenreferenz, 213
Metrik, 1101, 1129
CBO, 1131
Coupling Between Objects, 1131
Cyclomatic Complexity, 1130
DIT, 1131
Lines Of Code, 1130
LOC, 1130
McCabe, 1130
Method Lines Of Code, 1130
MLOC, 1130
NCSS, 1130
NOC, 1131
NOM, 1132
Non Commented Sourcecode Statements, 1130
Number Of Methods, 1132
Metriken, 1128
Microbenchmark, 808
eigener, 808
mit JMH, 811
Middleware, 582
Migration, 860
Automatic Module, 868
Bottom-up-, 865
Schwierigkeiten, 875
Migrationsszenarien, 863
min(), 434, 457
mismatch(), 408, 801
MLOC, 1130
Mock, 1206
Mockito, 1217
Modul, 826
Anforderungen an ein, 826
modulares JAR, 843
Modularisierung, 825
Grundlagen der, 826
Kompatibilität, 860
Migration, 860
Verzeichnislayout, 859
Vorteile, 827
Moduldeskriptor, 829
Monitor, 595, 596
Month, 498, 500
MonthDay, 500
MoreUnit, 1246
Multi Catch, 306, 920
Multi Map, 384
Multi-Module Build, 840
Multitasking, 581
Multithreading, 581
Murphy's Law, 1305
Mutator, 89

Nachbedingung, 115
Named Modules
Bottom-up-Migration, 865
Namen
aussagekräftige, 1110
Lesbarkeit, 1111
Semantik, 1111
sinnvoll gewählte, 1111
sinnvolle, konsistente, 1111
Namensgebung, 1110
grundlegende Regeln, 1107
sinnvolle, 1110
Namenskonsistenz, 1111
Namenskürzel
vermeide, 1110
Namensregeln, 1108
CamelCase-Schreibweise, 1108
nanoTime(), 809
Narrowing, 245
natürliche Ordnung, 359
naturalOrder(), 404
NavigableMap<K,V>, 377
ceilingKey(), 377
floorKey(), 377
higherKey(), 377
lowerKey(), 377
NCSS, 1130
Nebenläufigkeit, 581
negate(), 215, 216
NetBeans, 14
New Input Output, 295
newBuilder(), 738
newHttpClient(), 738
next(), 518
nextOrSame(), 518
Nightly Build, 65
NOC, 1131
NoClassDefFoundError, 872
Noise, 201
NOM, 1132
noneMatch(), 457
Normalfall, 1156
NoSuchElementException, 449
Null-Akzeptanz, 229, 233
Null-Objekt, 431, 472, 1058, 1306

null-Wert
Behandlung von, 405
IllegalArgumentException, 1015
NullPointerException, 1015
NullPointerException, 334, 405, 432, 436, 917, 945, 1015
Hilfe bei Fehlersuche, 806
unerwartete, 922, 925
Verbesserung bei, 805
nullsFirst(), 405
nullsLast(), 405
Number, 240
NumberFormat, 752
Format, 754
getCurrencyInstance(), 754
getInstance(), 754
getIntegerInstance(), 754
getNumberInstance(), 754
getPercentInstance(), 754
NumberFormatException, 247, 579

O-Notation, 1289
Object, 200, 221
equals(), 222, 227, 355
finalize(), 222
getClass(), 221, 224, 229
hashCode(), 222, 355, 356
notify(), 222, 591, 610, 617, 633, 940
notifyAll(), 222, 591, 610, 617, 940
toString(), 221, 223, 224
wait(), 222, 591, 610, 617, 633, 940
Objekt, 86
Objekt-Caching, 1325, 1326
Objekt-Pool, 1330
Objekt-Pooling, 1325
Objekt-Spaghetti, 1131
Objektgraph, 292
Objektintegrität, 1014
Probleme bei Remote Calls, 996
Probleme durch Zwischenzustände, 996
Objektmethoden
gebräuchlichste, 545
Objektzustand, 86, 114, 222, 227, 317, 907, 996, 1008, 1109, 1120, 1121
erwarteter, 1005
gültiger, 1018, 1118
konsistenter, 1002
ungültiger, 1005
Observer, 1086, 1087
als Anti-Pattern, 1095
OCP, 99, 110, 111, 167, 168, 1011
of(), 454, 481
ofDateAdjuster(), 519
ofDay(), 504
ofInstant(), 509, 526
ofNullable(), 483
ofString(), 738
Old Generation, 726
onComplete(), 671, 674
onError(), 671, 674
onNext(), 671, 673
onSubscribe(), 671
OO-Design
Aggregation, 91
Assoziation, 91
Attribut, 86
behaves-like-Beziehung, 87
Call-by-Reference, 97
Call-by-Value, 97
can-act-like-Beziehung, 87
Cast, 96
Datenkapselung, 111
Generalisierung, 93
Interface, 86, 123
is-a-Beziehung, 93
Kapselung, 89, 111
Kohäsion, 90, 112
Komposition, 91
Kopplung, 92
Membervariable, 86
Methode, 86
Objektzustand, 86
Overloading, 93, 94
Overriding, 93, 94
Polymorphie, 95
Realisierung, 87
Referenz, 88
Schnittstelle, 123
Spezialisierung, 93
Sub-Classing, 95
Sub-Typing, 95
Trennung von Zuständigkeiten, 111
Typ, 87
Type Cast, 96
Typkonformität, 87
Vererbung, 92
Vererbung und Wiederverwendbarkeit, 113
OO-Metrik, 1131
OO-Techniken
abstrakte Basisklasse, 128, 129

grundlegende, 123
Interface, 123
Marker-Interface, 153
Read-only-Interface, 142
Realisierung, 129
Schnittstelle, 123
Open Closed Principle, 99, 110, 111, 167, 168, 1011
OpenOption, 295
Operation
zustandsbehaftete, 479
Optimierung, 1274
80-zu-20-Regel, 1283
CPU-bound-, 1338
Grundlagen, 1274
I/O-bound-, 1318
Inlining, 1280
Loop-Unrolling, 1280
Memory-bound-, 1325
Peephole-, 619, 1281
Speicherverbrauch, 1325
von Stringoperationen, 1333
Vorher-nachher-Messungen, 1282
Optional<T>, 431, 439, 440, 472
empty(), 433, 440
get(), 433
ifPresent(), 433
ifPresentOrElse(), 434, 439
of(), 432
ofNullable(), 432
or(), 439
orElse(), 435
orElseGet(), 435
orElseThrow(), 435
stream(), 439–441
OptionalDouble, 434, 469
OptionalInt, 434
OptionalLong, 434
Options
addOption(), 566
or(), 215, 216, 439
Oracle-Verzeichnislayout
Schwachstellen des, 858
Ordnung
natürliche, 349, 359
orElse(), 435
Orthogonalität, 91, 487, 995, 1102, 1123
orTimeout(), 665
Out-of-Memory-Situation, 342, 1120
OutputStream, 280, 281
OutputStreamWriter, 283
Overloading, 93, 94, 232
Overriding, 93, 94, 232
invariantes, 181
kovariantes, 181

Paging, 466
Pair Programming, 1106, 1110, 1265, 1266, 1271
parallel(), 454, 478, 480, 481
Parallelisierbarkeit, 465, 477
parallelStream(), 480
Parallelverarbeitung, 454, 480
Wissenswertes zur, 476
Parameter Object, 160
Parameter Value Object, 160, 788
Parameterprüfung, 1118
Parsing, 516
partitioningBy(), 474
Path, 211
Paths, 211
Pattern Matching, 793
PatternSyntaxException, 269
peek(), 456, 463
Peephole-Optimierung, 619
Performance, 670
Optimierung der, 808
Period, 505, 510
Philosophenproblem, 603
plus(), 501, 504
PMD, 1135
POLA, 896, 1116
Polymorphie, 95, 110, 906
POM, 66
Port, 310
Post-Commit-Review, 1266
Post-Condition, 1118
Postdekrement, 893
Postinkrement, 893
Prädekrement, 893
Prädikat, 215
boolesche Bedingung, 215
komplexe Bedingung, 216
Präinkrement, 893
Pre-Commit-Review, 1266
Pre-Condition, 1118
Predicate<T>, 210, 215, 457, 461, 472, 483, 663
and(), 215, 216
negate(), 215, 216
or(), 215, 216
test(), 210, 215

previous(), 518
previousOrSame(), 518
primitiver Datentyp, 240
    boolean, 240
    byte, 240
    char, 240
    double, 240
    float, 240
    int, 240
    long, 240
    short, 240
Principle of Least Astonishment, 896, 1116
PrintStream, 265, 290
    print(), 290
    printf(), 265
    println(), 290
PrintWriter, 265
    printf(), 265
Prinzip des geringsten Erstaunens, 896
Prioritätswarteschlange, 447
PriorityBlockingQueue<E>, 639
PriorityQueue<E>, 447
Problem des Handlungsreisenden, 1290
Processor, 671, 679
Producer, 672
Producer-Consumer-Problem, 606
Profiling-Messung, 1288, 1296
Profiling-Tools, 1285, 1286
    JProfiler, 1287
    VisualVM, 1286
Programmierstil
    einheitlicher, 1106
Programmierung
    funktionale, 199
Project Jigsaw, 825
Project Object Model, 66
Properties, 561
    getProperty(), 569
    load(), 569
Prototyp, 1040
provides-Beziehung, 125
Proxy, 1084, 1307, 1309
Prozess, 582
Psychologie, 1124
Publisher, 670–675, 679
Publisher/Subscriber, 1086
Pull
    Beobachter, 1087
    Observer, 1087
Push
    Beobachter, 1087
    Observer, 1087
putIfAbsent(), 383

Qualified Export, 854
Qualität
    äußere, 1150
    innere, 1150
Qualitätssicherung, 1267
Qualitätssicherungsmaßnahme
    etablieren, 1150
    Mehraufwand durch, 1150
Queue<E>, 329, 447, 449, 639
    add(), 449
    element(), 449
    IllegalStateException, 449
    NoSuchElementException, 449
    offer(), 449
    peek(), 449
    poll(), 449
    remove(), 449
Quick Fix, 322

Rückgabewert
    komplexer, 788
Rückwärtskompatibilität, 318, 860, 862
Race Condition, 592, 594, 1036
Randfall, 1210
range(), 454
rangeClosed(), 481
Raw Type, 184, 1272
Reactive Programming, 670
Reactive Streams, 670, 671, 675
Reactor, 670
Read-only-Interface, 142, 991
Readability, 831
Readable, 291
readAllBytes(), 281
readAllLines(), 295
Reader, 280, 291
ReadWriteLock, 604, 605
Realisierung, 87
    konkrete, 129
Record, 786
    Compound Key, 788
    DTO, 788
    Gültigkeitsprüfung, 790
    Generics in, 791
    komplexer Rückgabewert, 788
    Parameter Value Object, 788
reduce(), 457, 473
reentrant, 602

ReentrantLock, 604, 605
ReentrantReadWriteLock, 604–606
Refactoring, 14, 953
    Encapsulate Field, 980
    Extract Interface, 989
    Extract Local Variable, 941
    Extract Method, 993
    High-Level, 979
    Standardvorgehen, 962
Reference Counting, 726
Referenzsemantik, 97, 149
Referenzvergleich, 227
Reflection, 221, 681, 706, 897
    Zugriffsschutz, 856
Reflexivität, 229, 233
Regel
    Keep It Clean, 1103
    Keep It Human-Readable, 1102
    Keep It Natural, 1102
    Keep It Simple And Short, 1102
Regressionstest, 45, 1152, 1237, 1247
regulärer Ausdruck, 266
    Bereichsangaben, 268
    Darstellung von Varianten, 267
    Spezialzeichen, 268
    Wiederholungen, 268
Rehashing, 358
Reihenfolge
    korrekte, 480
Release, 18, 63
Releasedatum, 504
Reliable Configuration, 825, 831, 863
Remote Debugging, 50
    Parametrierung der JVM, 51
Remote Proxy, 1085
Remote Repository, 29
remove(), 211
removeIf(), 333
Reordering, 619, 622
Reorderings, 624
repeat(), 271
replace(), 383
replaceAll(), 334, 345
Repository, 20, 23, 65
    lokales, 29
request(), 673, 674
Reverse Domain Name, 832
ReverseComparator, 400
    reverseOrder(), 400
reversed(), 404
reverseOrder(), 404
Rolle, 125, 138
Rückgabewert
    kovarianter, 182
Runnable, 200, 583, 630, 647, 659
    run(), 583
Runtime, 323, 1285
RuntimeException, 305
RxJava, 670

SAM-Typ, 200, 201, 275
Scanner, 290, 291
    ioException(), 292
    useDelimiter(), 291
Schablonenmethode, 1061
ScheduledExecutorService, 650
    scheduleAtFixedRate(), 651
    scheduleWithFixedDelay(), 651
ScheduledThreadPoolExecutor, 650
Scheduler, 582
Schnittmenge, 333
Schnittstelle, 123
Schönwettersoftware
    unzuverlässige, 300
Schulden
    technische, 1267
Sealed Types, 795
Security Update, 323
Seiteneffekt, 317, 1019, 1020, 1116
Selbstregulierung, 670
SELECT, 1301
Sender, 670
sequential(), 481
Serialisierung, 681, 708
Server, 123
Service Locator, 124
Set<E>, 328, 346, 386
    copyOf(), 391
Shallow Copy, 148
Short, 240
Short-circuiting Operations, 456
Shortcut-Return, 1117
Shut-down-Hook, 626, 899
Sicherheitsrisiko, 50
Sichtbarkeit, 619, 1122
    private, 89
    protected, 89
    public, 89
Signatur, 87
Silent Fail, 1119
SimpleDateFormat, 753
since, 318

Single Abstract Method, 275
Single Point of Failure, 34
Single Responsibility Principle, 167, 1122
Singleton, 572, 594, 1035
    als Anti-Pattern, 1039
    Lazy Initialization, 1037
    Race Condition, 1036
Skalierbarkeit, 670, 1288
skip(), 457
sleep(), 590
Socket
    close(), 310
Softwarebaustein
    deploybarer, 826
Softwarequalität, 1144
    Sichern der, 1144
SOLID-Prinzipien, 99, 110, 167, 1011, 1122
Soll-Verhalten, 1143
SonarQube, 1138
sort(), 209
sorted(), 457, 462, 465, 663
SortedMap<K,V>, 329, 377
SortedSet<E>, 329, 349
    first(), 349
    headSet(), 349
    last(), 349
    subSet(), 350
    tailSet(), 349
Sortierung
    ClassCastException, 348
    Comparable<T>, 359
    Comparator<T>, 360
    TreeMap<K,V>, 359
    TreeSet<E>, 359
Sourcecode-Checker, 1128
Sourcecode-Duplikation, 91
    Vermeiden von, 919
Space Efficiency, 388
SPDY, 737
Spezialisierung, 93
Split Package, 831
split(), 461
SRP, 167, 1122
Störeinfluss, 810
Stack<E>, 447, 448
    empty(), 448
    peek(), 448
    pop(), 448
    push(), 448
    search(), 448
StackOverflowError, 1055
Staging Area, 31
Stamm, 21
Standardverhalten
    vorgeben, 211
Start-Stopp-Messung
    einfache, 809
    unzuverlässige, 809
    Warm-up, 810
    wiederholte, 809
Starvation, 602, 603
statische Analyse, 1128
statusCode(), 738
Stellvertreterobjekt, 1201
Strategie, 1065
stream(), 439–441
Stream-API, 481
Stream-im-Stream, 460
Stream<T>, 440, 453, 481, 483
    allMatch(), 457, 470
    anyMatch(), 457, 470, 472
    collect(), 457, 468, 474, 482
    count(), 457, 469
    Create Operations, 454
    distinct(), 457, 465
    dropWhile(), 481, 482
    für primitive Typen, 455
    filter(), 456, 458, 460, 466, 472, 513
    findAny(), 457, 471, 480, 481
    findFirst(), 457, 471, 472, 481
    flatMap(), 440, 456, 460, 475, 663
    forEach(), 457, 480
    forEachOrdered(), 480
    Intermediate Operations, 456, 458, 465
    iterate(), 483, 484
    limit(), 457, 466, 513
    map(), 456, 459, 461, 663
    max(), 457
    min(), 457
    noneMatch(), 457
    of(), 454, 481
    ofNullable(), 483
    parallel(), 478, 480, 481
    peek(), 456, 463
    rangeClosed(), 481
    reduce(), 457, 473
    sequential(), 481
    skip(), 457, 466
    sorted(), 457, 462, 465, 663
    takeWhile(), 481, 482
    Terminal Operations, 456, 457, 467

toArray(), 457
String, 221, 256, 270, 283, 291, 490, 798
charAt(), 954
format(), 264
getBytes(), 283, 287
indent(), 798
isBlank(), 270
join(), 490
length(), 958
lines(), 271
mehrzeiliger, 774
repeat(), 271
split(), 265, 461
strip(), 270
stripLeading(), 270
stripTrailing(), 270
toLowerCase(), 278
transform(), 800
trim(), 315, 579
valueOf(), 224
StringBuffer, 226, 256, 261
delete(), 263
deleteCharAt(), 263
equals(), 263
insert(), 263
StringBuilder, 226, 256, 261
append(), 226
delete(), 263
deleteCharAt(), 263
equals(), 263
insert(), 263
StringIndexOutOfBoundsException, 954, 961
Stringliteral, 257
Stringliteral-Pool, 257, 1333
Stringobjekt, 257
StringTokenizer, 265, 315
nextToken(), 315
strip(), 270
stripLeading(), 270
stripTrailing(), 270
Stub, 1205
Sub-Classing, 95
Sub-Typing, 95
SubmissionPublisher, 672, 675
Subscriber, 670
Subscriber<T>, 671–675, 677–679
onComplete(), 671, 678
onError(), 671
onNext(), 671, 672, 678
onSubscribe(), 671
Subscription, 671–674, 677, 678
cancel(), 672
request(), 672
Substituierbarkeit, 93
Substitutionsprinzip, 93, 448
Subversion, 13
Suche
Binärsuche, 394
binarySearch(), 394
contains(), 393
containsAll(), 393
containsKey(), 393
containsValue(), 393
Die Rolle von equals(), 227
indexOf(), 393
lastIndexOf(), 393
summarizingInt(), 803
summaryStatistics(), 803
supplyAsync(), 660
switch
bisherige Schwachstellen, 780
Syntaxerweiterung, 780
unelegante Syntax, 783
Vollständigkeitsprüfung, 781
Switch Expressions, 779
Symmetrie, 229
Synchronisationsobjekt, 597
Synchronisierung, 582
synchronized, 595, 597, 599, 604, 621, 622, 633
SynchronousQueue<E>, 639
System
currentTimeMillis(), 514, 809
getenv(), 574
getProperty(), 574
nanoTime(), 809
System-Properties, 574
System.in, 280
System.out, 280
printf(), 264
Systemlast
Schwankungen der, 810
Systemtest, 1145

takeWhile(), 481, 482
Task, 642
TCP, 310
TDD, 1147, 1150
Technical Debt, 1267
technische Schulden, 1267
teeing(), 802

teile und herrsche, 652
Template-Methode, 1061
Temporal, 498, 500, 501, 517, 519
    with(), 519
Temporal Coupling, 968, 972, 1019, 1020
TemporalAccessor, 498, 500
TemporalAdjuster, 498, 500, 517
    adjustInto(), 517, 519
TemporalAdjusters, 508, 517, 521
    dayOfWeekInMonth(), 519
    firstDayOfMonth(), 518
    firstDayOfNextMonth(), 518
    firstDayOfNextYear(), 518
    firstDayOfYear(), 518
    firstInMonth(), 518
    lastDayOfMonth(), 518
    lastDayOfYear(), 518
    lastInMonth(), 518
    next(), 518
    nextOrSame(), 518
    ofDateAdjuster(), 519
    previous(), 518
    previousOrSame(), 518
TemporalAmount, 498, 510
TemporalField, 498
TemporalUnit, 498, 504
Terminal Operations, 456
test(), 210, 215
Test-Double, 1201
Test-Driven Development, 1147
Testabdeckung, 1248
Testen, 1143
Testfixture, 1156, 1160
Testklasse, 1108
    Postfix, 1108
TestNG, 1237
Testpyramide, 1145
Text Block, 774
    Erweiterungen in Java 14, 778
    Escaping, 775
    HTML, 776
    JavaScript, 776
    JSON, 777
    Platzhalter, 775
    Repositories, 777
    SQL, 777
    XML, 777
thenAccept(), 660, 665
thenAcceptAsync(), 660
thenApply(), 660
thenApplyAsync(), 662
thenCombine(), 662
Thread, 582, 583, 630
    Ableitung von der Klasse, 630
    Daemon-, 626
    getPriority(), 586
    getState(), 588
    getThreadGroup(), 587
    Gruppen, 587
    interrupt(), 591–593, 628, 631
    interrupted(), 631
    isAlive(), 585, 615
    isInterrupted(), 591, 592, 631
    join(), 615
    main-, 583, 626
    Priorität, 586
    run(), 583, 615
    setDaemon(), 626
    setDefaultUncaughtException-
        Handler(),
        628
    setPriority(), 586
    setUncaughtExceptionHandler(), 628
    sleep(), 590
    start(), 584, 615
    State, 588
    stop(), 586, 628
    ThreadUtils, 587
    User-, 626
    yield(), 590
Thread-Pool, 644
Thread-Scheduler, 588
ThreadGroup, 587
    activeCount(), 587
    enumerate(), 587
ThreadPoolExecutor, 649
Throwable, 314
Tiefe des Baums, 365
tiefe Kopie, 147, 148
TimeoutException, 665
toArray(), 457
toCollection(), 469
toEpochSecond(), 509
Token, 291
toList(), 468, 469
Tools
    Checkstyle, 1133
    PMD, 1135
    SonarQube, 1138
    Sourcecode-Checker, 1128
totalMemory(), 1285
toUnmodifiableList(), 484

toUnmodifiableMap(), 484
toUnmodifiableSet(), 484
transferTo(), 281
transform(), 800
Transformation
    in Lambda, 201
Transitivität, 229
Transmission Control Protocol, 310
Traveling-Salesman-Problem, 1290
Traversierung, 334
TreeMap<K,V>, 359, 377, 389, 1299
TreeSet<E>, 347, 349, 359, 389, 1299
Trennung von Zuständigkeiten, 85, 111, 238, 1002
trunk, 31
Typ, 87
Type Cast, 96
Type Erasure, 189
Type Inference, 202
Typhierarchie
    Collection<E>, 328
    Map<K,V>, 329
Typkonformität, 87
Typkürzel
    ?, 332
    ? extends E, 332
    E, 332
    K, 332
    T, 332
    V, 332
Typumwandlung, 886

Überschreiben
    invariantes, 181
    kovariantes, 181
Umgebungsvariable, 574
Umstellung
    schrittweise, 860
UnaryOperator<T>, 218, 459, 483
    identity(), 218
Unboxing, 247
UncaughtExceptionHandler, 627
Unchecked Exception, 304
Unicode, 257, 283
Unit Test, 37, 515, 525, 955, 962, 963, 1144, 1150, 1267
    als Regressionstest, 1237
    AssertJ, 1241
    EclEmma, 1252
    Hamcrest, 1237
    Infinitest, 1247
    JaCoCo, 1248
    Mockito, 1217
    MoreUnit, 1246
Universal Coordinated Time, 513
Unnamed Module, 863
Unordnung, 1103
UnsupportedOperationException, 140, 211, 335, 338, 379, 918, 1052, 1059
Unveränderlichkeit, 388, 389
Unvorhersagbarkeit, 480
Upper Type Bound, 422
Use-Beziehung, 123
User-Thread, 626
UTC, 513
UTF-16, 284
UTF-8, 270, 284
Utility-Klasse, 235
    HashUtils, 357
    SleepUtils, 584, 593

Value Object, 160, 1000
    Optimierung, 1337
var, 319
    Beschränkungen, 321
    für Lambdas, 322
    Fallstrick, 321
    Syntaxerweiterung, 319
Varargs, 159, 766
Varianz, 179
Vector<E>, 338, 340, 448, 1294, 1296
    add(), 341, 342
    get(), 341
    Größenanpassung, 342
    Kapazität, 340
    remove(), 342
    size(), 340
    Speicherverbrauch, 342
    trimToSize(), 343
Verarbeitung
    elementweise, 477, 478
    eventgetriebene, 670
    nicht blockierende, 670
Verarbeitungskette, 464, 660, 672, 679
Verarbeitungsschritt, 456, 476
Vererbung, 92, 1131
Vererbung und Wiederverwendbarkeit, 113
Vererbungshierarchie, 138, 910
    breite, 1128
    tiefe, 1128
Vergleich
    Referenz-, 227

Verhalten, 85, 227
Verhaltensweise, 125
Verklemmung, 582
Version, 323
Versionsbestandteil, 323
Versionsnummer, 323
Versionsschema, 323
Versionsverwaltung, 13
    auschecken, 24
    Branch, 26
    Check-in, 24
    Check-out, 24
    commit, 24
    dezentrale, 23
    einchecken, 24
    Konflikt, 25
    Main-Branch, 26
    Marke, 26
    Merge, 25
    Repository, 20, 23
    Stamm, 26
    Synchronize, 25
    Tag, 26
    Trunk, 26
    Update, 25
    zentrale, 23
Verzeichnisaufbau, 825
Verzeichnislayout, 859
    empfehlenswertes, 858
    empfohlenes, 859
    Oracle propagiertes, 858
Virtual Proxy, 1085
VisualVM, 1286, 1355
volatile, 620, 622, 630, 633
Vorbedingung, 115
Vorteile
    Maven und Gradle, 82

Wahrnehmungspsychologie, 1103
    Gesetz der Ähnlichkeit, 1103
    Gesetz der Nähe, 1105
Warm-up, 810
Wartungsalbtraum, 891
weakly consistent, 638
Whitebox-Test, 1146
Whitespace, 334
Widening, 210, 245, 252
Wiederverwendbarkeit, 85, 995, 1121, 1123
Wiederverwendung, 92
with(), 519
Working Copy, 24
Worthäufigkeitshistogramm, 475
Wrapper, 1049
Wrapper-Klassen, 240
write(), 295, 296
Writer, 280, 283
    close(), 283
    flush(), 283
    write(), 283
writeString(), 296
Wurzel, 365

XP, 1147, 1150

Year, 500
    isLeap(), 501
YearMonth, 500
yield
    mit Rückgabewert, 785
Young Generation, 726

Zeichencodierung, 270
Zeichensatz
    Einfluss, 802
Zeitangabe
    unvollständige, 507
ZonedDateTime, 499, 511–513, 526
    ofInstant(), 526
ZoneId, 499, 509, 512, 513
    getAvailableZoneIds(), 513
    of(), 512
ZoneOffset, 499, 509, 512, 513
Zuständigkeit
    beim Testen, 1147
Zustand, 85, 227
Zustandsinformation, 1116
Zustandsprüfung
    Einführen einer, 1013
    IllegalStateException, 1013
Zweierkomplement, 241, 254
Zyklen, 1055