

5.

Auflage



Michael Inden

# Der Weg zum Java-Profi

Konzepte und Techniken für die  
professionelle Java-Entwicklung

dpunkt.verlag

# Inhalt

**Cover**

**Über den Autor**

**Titel**

**Impressum**

**Inhaltsübersicht**

**Inhaltsverzeichnis**

**Vorwort**

Vorwort zur 5. Auflage

**1 Einleitung**

1.1 Über dieses Buch

1.1.1 Motivation

1.1.2 Was leistet dieses Buch und was nicht?

1.1.3 Wie und was soll mithilfe des Buchs gelernt werden?

1.1.4 Wer sollte dieses Buch lesen?

1.2 Aufbau des Buchs

1.3 Konventionen und ausführbare Programme

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

2 Professionelle Arbeitsumgebung

2.1 Vorteile von IDEs am Beispiel von Eclipse

2.2 Projektorganisation

2.2.1 Projektstruktur in Eclipse

2.2.2 Projektstruktur für Maven und Gradle

2.3 Einsatz von Versionsverwaltungen

2.3.1 Arbeiten mit zentralen Versionsverwaltungen

2.3.2 Dezentrale Versionsverwaltungen

- 2.3.3 VCS und DVCS im Vergleich
  - 2.4 Einsatz eines Unit-Test-Frameworks
    - 2.4.1 Das JUnit-Framework
    - 2.4.2 Parametrisierte Tests mit JUnit 5
    - 2.4.3 Vorteile von Unit Tests
  - 2.5 Debugging
    - 2.5.1 Fehlersuche mit einem Debugger
    - 2.5.2 Remote Debugging
  - 2.6 Deployment von Java-Applikationen
    - 2.6.1 Das JAR-Tool im Kurzüberblick
    - 2.6.2 JAR inspizieren und ändern, Inhalt extrahieren
    - 2.6.3 Metainformationen und das Manifest
    - 2.6.4 Inspizieren einer JAR-Datei
  - 2.7 Einsatz eines IDE-unabhängigen Build-Prozesses
    - 2.7.1 Maven im Überblick
    - 2.7.2 Builds mit Gradle
    - 2.7.3 Vorteile von Maven und Gradle
  - 2.8 Weiterführende Literatur
- 3 Objektorientiertes Design
  - 3.1 OO-Grundlagen
    - 3.1.1 Grundbegriffe
    - 3.1.2 Beispielentwurf: Ein Zähler
    - 3.1.3 Vom imperativen zum objektorientierten Entwurf
    - 3.1.4 Diskussion der OO-Grundgedanken
    - 3.1.5 Wissenswertes zum Objektzustand
  - 3.2 Grundlegende OO-Techniken
    - 3.2.1 Schnittstellen (Interfaces)
    - 3.2.2 Basisklassen und abstrakte Basisklassen

- 3.2.3 Interfaces und abstrakte Basisklassen
  - 3.3 Wissenswertes zu Vererbung
- 3.3.1 Probleme durch Vererbung
- 3.3.2 Delegation statt Vererbung
  - 3.4 Fortgeschrittenere OO-Techniken
- 3.4.1 Read-only-Interface
- 3.4.2 Immutable-Klasse
- 3.4.3 Marker-Interface
- 3.4.4 Konstantensammlungen und Aufzählungen
- 3.4.5 Value Object (Data Transfer Object)
  - 3.5 Prinzipien guten OO-Designs
- 3.5.1 Geheimnisprinzip nach Parnas
- 3.5.2 Law of Demeter
- 3.5.3 SOLID-Prinzipien
  - 3.6 Formen der Varianz
- 3.6.1 Grundlagen der Varianz
- 3.6.2 Kovariante Rückgabewerte
  - 3.7 Generische Typen (Generics)
- 3.7.1 Einführung
- 3.7.2 Generics und Auswirkungen der Type Erasure
  - 3.8 Weiterführende Literatur
- 4 Lambdas, Methodenreferenzen und Defaultmethoden
  - 4.1 Einstieg in Lambdas
- 4.1.1 Syntax von Lambdas
- 4.1.2 Functional Interfaces und SAM-Typen
- 4.1.3 Exceptions in Lambdas
  - 4.2 Syntaxerweiterungen in Interfaces
- 4.2.1 Defaultmethoden

- 4.2.2 Statische Methoden in Interfaces
  - 4.3 Methodenreferenzen
  - 4.4 Externe vs. interne Iteration
  - 4.5 Wichtige Functional Interfaces für Collections
- 4.5.1 Das Interface Predicate
- 4.5.2 Das Interface UnaryOperator
  - 4.6 Praxiswissen: Definition von Lambdas
- 5 Java-Grundlagen
  - 5.1 Die Klasse Object
    - 5.1.1 Die Methode toString()
    - 5.1.2 Die Methode equals()
  - 5.2 Primitive Typen und Wrapper-Klassen
    - 5.2.1 Grundlagen
    - 5.2.2 Konvertierung von Werten
    - 5.2.3 Wissenswertes zu Auto-Boxing und Auto-Unboxing
    - 5.2.4 Ausgabe und Verarbeitung von Zahlen
  - 5.3 Stringverarbeitung
    - 5.3.1 Die Klasse String
    - 5.3.2 Die Klassen StringBuffer und StringBuilder
    - 5.3.3 Ausgaben mit format() und printf()
    - 5.3.4 Die Methode split() und reguläre Ausdrücke
    - 5.3.5 Optimierung bei Strings in JDK 9
    - 5.3.6 Neue Methoden in der Klasse String in JDK 11
  - 5.4 Varianten innerer Klassen
  - 5.5 Ein- und Ausgabe (I/O)
    - 5.5.1 Dateibehandlung und die Klasse File
    - 5.5.2 Ein- und Ausgabestreams im Überblick
    - 5.5.3 Zeichencodierungen bei der Ein- und Ausgabe

- 5.5.4 Speichern und Laden von Daten und Objekten
- 5.5.5 Dateiverarbeitung mit dem NIO
- 5.5.6 Neue Hilfsmethoden in der Klasse Files in JDK 11

#### 5.6 Fehlerbehandlung

- 5.6.1 Einstieg in die Fehlerbehandlung
- 5.6.2 Checked Exceptions und Unchecked Exceptions
- 5.6.3 Besonderheiten beim Exception Handling
- 5.6.4 Exception Handling und Ressourcenfreigabe
- 5.6.5 Assertions

#### 5.7 Weitere Neuerungen in JDK 9, 10 und 11

- 5.7.1 Erweiterung der @Deprecated-Annotation in JDK 9
- 5.7.2 Syntaxerweiterung var in JDK 10 und 11
- 5.7.3 Versionsverarbeitung mit JDK 9 und 10

#### 5.8 Weiterführende Literatur

## **II Bausteine stabiler Java-Applikationen**

### 6 Das Collections-Framework

#### 6.1 Datenstrukturen und Containerklassen

- 6.1.1 Wahl einer geeigneten Datenstruktur
- 6.1.2 Arrays
- 6.1.3 Das Interface Collection
- 6.1.4 Das Interface Iterator
- 6.1.5 Listen und das Interface List
- 6.1.6 Mengen und das Interface Set
- 6.1.7 Grundlagen von hashbasierten Containern
- 6.1.8 Grundlagen automatisch sortierender Container
- 6.1.9 Die Methoden equals(), hashCode() und compareTo() im Zusammenspiel
- 6.1.10 Schlüssel-Wert-Abbildungen und das Interface Map
- 6.1.11 Erweiterungen am Beispiel der Klasse HashMap

- 6.1.12 Erweiterungen im Interface Map in JDK 8
- 6.1.13 Collection-Factory-Methoden in JDK 9
- 6.1.14 Unveränderliche Kopien von Collections mit Java 10
- 6.1.15 Entscheidungshilfe zur Wahl von Datenstrukturen
  - 6.2 Suchen und Sortieren
    - 6.2.1 Suchen
    - 6.2.2 Sortieren von Arrays und Listen
    - 6.2.3 Sortieren mit Komparatoren
    - 6.2.4 Erweiterungen im Interface Comparator mit JDK 8
  - 6.3 Utility-Klassen und Hilfsmethoden
    - 6.3.1 Nützliche Hilfsmethoden
    - 6.3.2 Dekorierer synchronized und unmodifiable
    - 6.3.3 Vordefinierte Algorithmen in der Klasse Collections
  - 6.4 Containerklassen: Generics und Varianz
  - 6.5 Die Klasse Optional
    - 6.5.1 Grundlagen zur Klasse Optional
    - 6.5.2 Weiterführendes Beispiel und Diskussion
    - 6.5.3 Verkettete Methodenaufrufe
    - 6.5.4 Erweiterungen in der Klasse Optional in JDK 9
    - 6.5.5 Erweiterung in Optional in JDK 10 und 11
  - 6.6 Fallstricke im Collections-Framework
    - 6.6.1 Wissenswertes zu Arrays
    - 6.6.2 Wissenswertes zu Stack, Queue und Deque
  - 6.7 Weiterführende Literatur
- 7 Das Stream-API
  - 7.1 Grundlagen zu Streams
    - 7.1.1 Streams erzeugen – Create Operations
    - 7.1.2 Intermediate und Terminal Operations im Überblick

- 7.1.3 Zustandslose Intermediate Operations
- 7.1.4 Zustandsbehaftete Intermediate Operations
- 7.1.5 Terminal Operations
- 7.1.6 Wissenswertes zur Parallelverarbeitung
- 7.1.7 Neuerungen im Stream-API in JDK 9
- 7.1.8 Neuerungen im Stream-API in JDK 10
- 7.2 Filter-Map-Reduce
  - 7.2.1 Herkömmliche Realisierung
  - 7.2.2 Filter-Map-Reduce mit JDK 8
- 7.3 Praxisbeispiele
  - 7.3.1 Aufbereiten von Gruppierungen und Histogrammen
  - 7.3.2 Maps nach Wert sortieren
- 8 Datumsverarbeitung seit JDK 8
  - 8.1 Überblick über die neu eingeführten Typen
    - 8.1.1 Neue Aufzählungen, Klassen und Interfaces
    - 8.1.2 Die Aufzählungen DayOfWeek und Month
    - 8.1.3 Die Klassen MonthDay, YearMonth und Year
    - 8.1.4 Die Klasse Instant
    - 8.1.5 Die Klasse Duration
    - 8.1.6 Die Aufzählung ChronoUnit
    - 8.1.7 Die Klassen LocalDate, LocalTime und LocalDateTime
    - 8.1.8 Die Klasse Period
    - 8.1.9 Die Klasse ZonedDateTime
    - 8.1.10 Zeitzonen und die Klassen ZoneId und ZoneOffset
    - 8.1.11 Die Klasse Clock
    - 8.1.12 Formatierung und Parsing
  - 8.2 Datumsarithmetik
    - 8.2.1 Einstieg in die Datumsarithmetik

- 8.2.2 Real-World-Example: Gehaltszahltag
  - 8.3 Interoperabilität mit Legacy-Code
- 9 Applikationsbausteine
  - 9.1 Einsatz von Bibliotheken
  - 9.2 Google Guava im Kurzüberblick
    - 9.2.1 String-Aktionen
    - 9.2.2 Stringkonkatenation und -extraktion
    - 9.2.3 Erweiterungen für Collections
    - 9.2.4 Weitere Utility-Funktionalitäten
      - 9.3 Wertebereichs- und Parameterprüfungen
      - 9.4 Logging-Frameworks
        - 9.4.1 Apache log4j2
        - 9.4.2 Tipps und Tricks zum Einsatz von Logging mit log4j2
          - 9.5 Konfigurationsparameter und -dateien
            - 9.5.1 Einlesen von Kommandozeilenparametern
            - 9.5.2 Verarbeitung von Properties
            - 9.5.3 Weitere Möglichkeiten zur Konfigurationsverwaltung
- 10 Multithreading-Grundlagen
  - 10.1 Threads und Runnables
    - 10.1.1 Definition der auszuführenden Aufgabe
    - 10.1.2 Start, Ausführung und Ende von Threads
    - 10.1.3 Lebenszyklus von Threads und Thread-Zustände
    - 10.1.4 Unterbrechungswünsche durch Aufruf von interrupt()
  - 10.2 Zusammenarbeit von Threads
    - 10.2.1 Konkurrierende Datenzugriffe
    - 10.2.2 Locks, Monitore und kritische Bereiche
    - 10.2.3 Deadlocks und Starvation
    - 10.2.4 Kritische Bereiche und das Interface Lock

## 10.3 Kommunikation von Threads

### 10.3.1 Kommunikation mit Synchronisation

### 10.3.2 Kommunikation über die Methoden wait(), notify() und notifyAll()

### 10.3.3 Abstimmung von Threads

### 10.3.4 Unerwartete IllegalMonitorStateExceptions

## 10.4 Das Java-Memory-Modell

### 10.4.1 Sichtbarkeit

### 10.4.2 Atomarität

### 10.4.3 Reorderings

## 10.5 Besonderheiten bei Threads

### 10.5.1 Verschiedene Arten von Threads

### 10.5.2 Exceptions in Threads

### 10.5.3 Sicheres Beenden von Threads

## 10.6 Weiterführende Literatur

## 11 Modern Concurrency

### 11.1 Concurrent Collections

#### 11.1.1 Thread-Sicherheit und Parallelität mit »normalen« Collections

#### 11.1.2 Parallelität mit den Concurrent Collections

#### 11.1.3 Blockierende Warteschlangen und das Interface Blocking-Queue

### 11.2 Das Executor-Framework

#### 11.2.1 Einführung

#### 11.2.2 Definition von Aufgaben

#### 11.2.3 Parallele Abarbeitung im ExecutorService

### 11.3 Das Fork-Join-Framework

#### 11.3.1 Einführendes Beispiel

#### 11.3.2 Real-World-Example: Merge Sort

### 11.4 Die Klasse CompletableFuture

#### 11.4.1 Einführung

11.4.2 Beispiel: Parallele Verarbeitung von Dateiinhalten

11.4.3 Erweiterungen in JDK 9

11.4.4 Beispiel: Von synchron zu multithreaded

11.5 Reactive Streams und die Klasse Flow

11.5.1 Schnelleinstieg Reactive Streams

11.5.2 Reactive Streams im JDK

11.5.3 Beispiel zur Klasse Flow

11.5.4 Fazit

11.6 Weiterführende Literatur

12 Fortgeschrittene Java-Themen

12.1 Crashkurs Reflection

12.1.1 Grundlagen

12.1.2 Zugriff auf Methoden und Attribute

12.1.3 Spezialfälle

12.1.4 Type Erasure und Typinformationen bei Generics

12.1.5 Fazit

12.2 Annotations

12.2.1 Einführung in Annotations

12.2.2 Standard-Annotations des JDKs

12.2.3 Definition eigener Annotations

12.2.4 Annotations zur Laufzeit auslesen

12.3 Serialisierung

12.3.1 Grundlagen der Serialisierung

12.3.2 Die Serialisierung anpassen

12.3.3 Versionsverwaltung der Serialisierung

12.3.4 Optimierung der Serialisierung

12.4 Garbage Collection

12.4.1 Grundlagen zur Garbage Collection

12.4.2 Der Garbage Collector »G1«

12.5 Dynamic Proxies

12.5.1 Statischer Proxy

12.5.2 Dynamischer Proxy

12.6 HTTP/2-API

12.6.1 Einführung

12.6.2 Real-World-Example: Wechselkurs mit REST

12.6.3 Fazit

12.7 Weiterführende Literatur

13 Basiswissen Internationalisierung

13.1 Internationalisierung im Überblick

13.1.1 Grundlagen und Normen

13.1.2 Die Klasse Locale

13.1.3 Die Klasse ResourceBundle

13.1.4 Formatierte Ein- und Ausgabe

13.1.5 Datumswerte und die Klasse DateFormat

13.1.6 Zahlen und die Klasse NumberFormat

13.1.7 Textmeldungen und die Klasse MessageFormat

13.1.8 Stringvergleiche mit der Klasse Collator

13.2 Programmbausteine zur Internationalisierung

13.2.1 Unterstützung mehrerer Datumsformate

13.2.2 Fazit und Ausblick

### **III Wichtige Neuerungen in Java 12 bis 15**

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

14.1 Syntaxneuerungen

14.1.1 Text Blocks

14.1.2 Switch Expressions

14.1.3 Records (Preview)

14.1.4 Pattern Matching bei instanceof (Preview)

14.1.5 Sealed Types (Preview)

14.1.6 Lokale Enums und Interfaces (Preview)

14.2 API-Neuerungen

14.2.1 Neue Methoden in der Klasse String

14.2.2 Neue Hilfsmethode in der Utility-Klasse Files

14.2.3 Der teeing()-Kollektor

14.3 JVM-Neuerungen

14.3.1 Verbesserung bei NullPointerExceptions

14.3.2 Entfernung der JavaScript-Engine

14.4 Microbenchmark Suite

14.4.1 Eigene Microbenchmarks und Varianten davon

14.4.2 Microbenchmarks mit JMH

14.4.3 Fazit zu JMH

14.5 Java 15 – notwendige Anpassungen für Build-Tools und IDEs

14.5.1 Java 15 mit Gradle

14.5.2 Java 15 mit Maven

14.5.3 Java 15 mit Eclipse

14.5.4 Java 15 mit IntelliJ

14.5.5 Java 15 mit JShell oder der Kommandozeile

14.6 Fazit

## **IV Modularisierung**

15 Modularisierung mit Project Jigsaw

15.1 Grundlagen

15.1.1 Begrifflichkeiten

15.1.2 Ziele von Project Jigsaw

15.2 Modularisierung im Überblick

15.2.1 Grundlagen zu Project Jigsaw

15.2.2 Beispiel mit zwei Modulen

15.2.3 Packaging

15.2.4 Abhängigkeiten und Modulgraphen

15.2.5 Module des JDKs einbinden

15.2.6 Arten von Modulen

15.3 Sichtbarkeiten und Zugriffsschutz

15.3.1 Sichtbarkeiten

15.3.2 Zugriffsschutz und Reflection

15.4 Empfehlenswertes Verzeichnislayout für Module

15.5 Kompatibilität und Migration

15.5.1 Kompatibilitätsmodus

15.5.2 Migrationsszenarien

15.5.3 Fallstrick bei der Bottom-up-Migration

15.5.4 Beispiel: Migration mit Automatic Modules

15.5.5 Beispiel: Automatic und Unnamed Module

15.5.6 Beispiel: Abwandlung mit zwei Automatic Modules

15.5.7 Fazit

15.6 Zusammenfassung

## **V Fallstricke und Lösungen im Praxisalltag**

16 Bad Smells

16.1 Programmdesign

16.1.1 Bad Smell: Verwenden von Magic Numbers

16.1.2 Bad Smell: Konstanten in Interfaces definieren

16.1.3 Bad Smell: Zusammengehörende Konstanten nicht als Typ definiert

16.1.4 Bad Smell: Casts auf unbekannte Subtypen

16.1.5 Bad Smell: Programmcode im Logging-Code

16.1.6 Bad Smell: Dominanter Logging-Code

16.1.7 Bad Smell: Unvollständige Änderungen nach Copy-Paste

- 16.1.8 Bad Smell: Unvollständige Betrachtung aller Alternativen
- 16.1.9 Bad Smell: Prä-/Postinkrement in komplexeren Statements
- 16.1.10 Bad Smell: Mehrere aufeinanderfolgende Parameter gleichen Typs
- 16.1.11 Bad Smell: Grundloser Einsatz von Reflection
- 16.1.12 Bad Smell: System.exit() mitten im Programm
- 16.1.13 Bad Smell: Variablendeklaration nicht im kleinstmöglichen Sichtbarkeitsbereich

## 16.2 Klassendesign

- 16.2.1 Bad Smell: Unnötigerweise veränderliche Attribute
- 16.2.2 Bad Smell: Herausgabe von this im Konstruktor
- 16.2.3 Bad Smell: Aufruf abstrakter Methoden im Konstruktor
- 16.2.4 Bad Smell: Mix abstrakter und konkreter Basisklassen
- 16.2.5 Bad Smell: Referenzierung von Subklassen in Basisklassen
- 16.2.6 Bad Smell: Öffentlicher Defaultkonstruktor lediglich zum Zugriff auf Hilfsmethoden

## 16.3 Fehlerbehandlung und Exception Handling

- 16.3.1 Bad Smell: Unbehandelte Exception
- 16.3.2 Bad Smell: Unpassender Exception-Typ
- 16.3.3 Bad Smell: Fangen der allgemeinsten Exception
- 16.3.4 Bad Smell: Exceptions zur Steuerung des Kontrollflusses
- 16.3.5 Bad Smell: Unbedachte Rückgabe von null
- 16.3.6 Bad Smell: Rückgabe von null statt Exception im Fehlerfall
- 16.3.7 Bad Smell: Sonderbehandlung von Randfällen
- 16.3.8 Bad Smell: Keine Gültigkeitsprüfung von Eingabeparametern
- 16.3.9 Bad Smell: Fehlerhafte Fehlerbehandlung
- 16.3.10 Bad Smell: I/O ohne finally oder ARM
- 16.3.11 Bad Smell: Resource Leaks durch Exceptions im Konstruktor

## 16.4 Häufige Fallstricke

## 16.5 Weiterführende Literatur

## 17 Refactorings

### 17.1 Refactorings am Beispiel

### 17.2 Das Standardvorgehen

### 17.3 Kombination von Basis-Refactorings

#### 17.3.1 Refactoring-Beispiel: Ausgangslage und Ziel

#### 17.3.2 Auflösen der Abhängigkeiten

#### 17.3.3 Vereinfachungen

#### 17.3.4 Verlagern von Funktionalität

### 17.4 Der Refactoring-Katalog

#### 17.4.1 Reduziere die Sichtbarkeit von Attributen

#### 17.4.2 Minimiere veränderliche Attribute

#### 17.4.3 Reduziere die Sichtbarkeit von Methoden

#### 17.4.4 Ersetze Mutator- durch Business-Methode

#### 17.4.5 Minimiere Zustandsänderungen

#### 17.4.6 Führe ein Interface ein

#### 17.4.7 Spalte ein Interface auf

#### 17.4.8 Führe ein Read-only-Interface ein

#### 17.4.9 Führe ein Read-Write-Interface ein

#### 17.4.10 Lagere Funktionalität in Hilfsmethoden aus

#### 17.4.11 Trenne Informationsbeschaffung und -verarbeitung

#### 17.4.12 Wandle Konstantensammlung in enum um

#### 17.4.13 Entferne Exceptions zur Steuerung des Kontrollflusses

#### 17.4.14 Wandle in Utility-Klasse mit statischen Hilfsmethoden um

#### 17.4.15 Löse if-else / instanceof durch Polymorphie auf

### 17.5 Defensives Programmieren

#### 17.5.1 Führe eine Zustandsprüfung ein

#### 17.5.2 Überprüfe Eingabeparameter

### 17.6 Fallstricke bei Refactorings

## 17.7 Weiterführende Literatur

## 18 Entwurfsmuster

### 18.1 Erzeugungsmuster

18.1.1 Erzeugungsmethode

18.1.2 Fabrikmethode (Factory Method)

18.1.3 Erbauer (Builder)

18.1.4 Singleton

18.1.5 Prototyp (Prototype)

### 18.2 Strukturmuster

18.2.1 Fassade (Façade)

18.2.2 Adapter

18.2.3 Dekorierer (Decorator)

18.2.4 Kompositum (Composite)

### 18.3 Verhaltensmuster

18.3.1 Iterator

18.3.2 Null-Objekt (Null Object)

18.3.3 Schablonenmethode (Template Method)

18.3.4 Strategie (Strategy)

18.3.5 Befehl (Command)

18.3.6 Proxy

18.3.7 Beobachter (Observer)

18.3.8 MVC-Architektur

### 18.4 Weiterführende Literatur

## **VI Qualitätssicherungsmaßnahmen**

### 19 Programmierstil und Coding Conventions

#### 19.1 Grundregeln eines guten Programmierstils

19.1.1 Keep It Human-Readable

19.1.2 Keep It Simple And Short (KISS)

19.1.3 Keep It Natural

19.1.4 Keep It Clean

19.2 Die Psychologie beim Sourcecode-Layout

19.2.1 Gesetz der Ähnlichkeit

19.2.2 Gesetz der Nähe

19.3 Coding Conventions

19.3.1 Grundlegende Namens- und Formatierungsregeln

19.3.2 Namensgebung

19.3.3 Dokumentation

19.3.4 Programmdesign

19.3.5 Klassendesign

19.3.6 Parameterlisten

19.3.7 Logik und Kontrollfluss

19.4 Sourcecode-Prüfung mit Tools

19.4.1 Metriken

19.4.2 Sourcecode-Prüfung im Build-Prozess

20 Unit Tests

20.1 Testen im Überblick

20.1.1 Was versteht man unter Testen?

20.1.2 Testarten im Überblick

20.1.3 Zuständigkeiten beim Testen

20.1.4 Testen und Qualität

20.2 Wissenswertes zu Testfällen

20.2.1 Testfälle mit JUnit definieren

20.2.2 Problem der Kombinatorik beim Bestimmen von Testfällen

20.3 Besondere Assertions und Annotations

20.4 Parametrisierte Tests mit JUnit 5

20.4.1 Einstieg

- 20.4.2 Verbesserung des Tests der Rabattberechnung
- 20.4.3 Praxisbeispiel: Berechnung in Testfall vereinfachen
- 20.4.4 Praxis-Trickkiste

## 20.5 Fortgeschrittene Unit-Test-Techniken

- 20.5.1 Stellvertreterobjekte / Test-Doubles
- 20.5.2 Vorarbeiten für das Testen mit Stubs und Mocks
- 20.5.3 Die Technik EXTRACT AND OVERRIDE
- 20.5.4 Einstieg in das Testen mit Mocks und Mockito
- 20.5.5 Abhängigkeiten mit Mockito auflösen
- 20.5.6 Unit Tests von privaten Methoden

## 20.6 Test Smells

- 20.6.1 Test Smell: Unangebrachtes assertTrue() und assertFalse()
- 20.6.2 Test Smell: Zu viele Asserts im Testfall
- 20.6.3 Test Smell: Asserts ohne Hinweis
- 20.6.4 Test Smell: Einsatz von toString() in assertEquals()
- 20.6.5 Test Smell: Unit Tests zur Prüfung von Laufzeiten

## 20.7 Nützliche Tools für Unit Tests

- 20.7.1 Hamcrest
- 20.7.2 AssertJ
- 20.7.3 MoreUnit
- 20.7.4 Infinitest
- 20.7.5 JaCoCo
- 20.7.6 EclEmma

## 20.8 Umstieg von JUnit 4 auf JUnit 5

- 20.8.1 Erweiterung im Gradle-Build für JUnit-5-Tests
- 20.8.2 Veränderungen in den Annotations
- 20.8.3 Alternativen für JUnit Rules
- 20.8.4 Veränderungen bei parametrisierten Tests

## 20.8.5 Alternative zur Hamcrest-Integration in JUnit 4

### 20.9 Fazit

### 20.10 Weiterführende Literatur

## 21 Codereviews

### 21.1 Definition

### 21.2 Probleme und Tipps zur Durchführung

### 21.3 Vorteile von Codereviews

### 21.4 Codereview-Checkliste

## 22 Optimierungen

### 22.1 Grundlagen

#### 22.1.1 Optimierungsebenen und Einflussfaktoren

#### 22.1.2 Optimierungstechniken

#### 22.1.3 CPU-bound-Optimierungsebenen am Beispiel

#### 22.1.4 Messungen – Erkennen kritischer Bereiche

#### 22.1.5 Abschätzungen mit der O-Notation

### 22.2 Einsatz geeigneter Datenstrukturen

#### 22.2.1 Einfluss von Arrays und Listen

#### 22.2.2 Optimierungen für Set und Map

#### 22.2.3 Design eines Zugriffsinterface

### 22.3 Lazy Initialization

#### 22.3.1 Konsequenzen des Einsatzes der Lazy Initialization

#### 22.3.2 Lazy Initialization mithilfe des PROXY-Musters

### 22.4 Optimierungen am Beispiel

### 22.5 I/O-bound-Optimierungen

#### 22.5.1 Technik – Wahl passender Strategien

#### 22.5.2 Technik – Caching und Pooling

#### 22.5.3 Technik – Vermeidung unnötiger Aktionen

### 22.6 Memory-bound-Optimierungen

22.6.1 Technik – Wahl passender Strategien

22.6.2 Technik – Caching und Pooling

22.6.3 Optimierungen der Stringverarbeitung

22.6.4 Technik – Vermeidung unnötiger Aktionen

22.7 CPU-bound-Optimierungen

22.7.1 Technik – Wahl passender Strategien

22.7.2 Technik – Caching und Pooling

22.7.3 Technik – Vermeidung unnötiger Aktionen

22.8 Weiterführende Literatur

23 Schlussgedanken

## **VII Anhang**

A Grundlagen zur Java Virtual Machine

A.1 Wissenswertes rund um die Java Virtual Machine

A.1.1 Ausführung eines Java-Programms

A.1.2 Speicherverwaltung und Classloading

## **Literaturverzeichnis**

## **Index**

## 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.<sup>1</sup>

## 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                                @FunctionalInterface

public interface Runnable                            public interface
Comparator<T>

{

    public abstract void run();                    int compare(T o1, T
o2);

}                                                    boolean
equals(Object obj);

                                                    }
}
```

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 Rückgabotyp 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.<sup>2</sup> 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 `IOExceptions` 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;

    }

    // 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 `RuntimeExceptions` 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 Informationen

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 <http://www.javaspecialists.eu/archive/Issue221.html> nützlich:

## 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	obj -> obj.toString()
	String::compareTo	(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<sup>3</sup> zeigt den sequenziellen Charakter weniger klar.<sup>4</sup> In allen drei Fällen spricht man von **externer Iteration**, weil die **Traversierung im Applikationscode programmiert** wird.

### 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;  
  
}
```

# 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
- Entwurfstil, 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

groupBy(), 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
- Infinittest, 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
- InterruptedException, 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