

LEHRBUCH

Dietmar Abts

# Grundkurs JAVA

Von den Grundlagen bis zu  
Datenbank- und Netzanwendungen

*12. Auflage*



Springer Vieweg

---

# **Grundkurs JAVA**

---

Dietmar Abts

# Grundkurs JAVA

Von den Grundlagen bis zu  
Datenbank- und Netzanwendungen

12. Auflage



Springer Vieweg

Dietmar Abts  
Ratingen, Deutschland

ISBN 978-3-658-43573-8                    ISBN 978-3-658-43574-5 (eBook)  
<https://doi.org/10.1007/978-3-658-43574-5>

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

© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert an Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2015, 2016, 2018, 2020, 2024

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung/Lektorat: Leonardo Milla  
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.  
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Das Papier dieses Produkts ist recyclebar.

# Vorwort zur 12. Auflage

Diese Auflage stellt wesentliche Aspekte der Programmiersprache Java vor. Gegenüber der vorherigen Auflage wurden neue Features aufgenommen.

Das vorliegende Buch basiert auf der zum Zeitpunkt der Drucklegung aktuellen *Java-Version 21*. Es bietet einen breiten Einstieg in die Java-Programmierung und führt Sie schrittweise durch die verschiedenen Themen, von den elementaren Grundlagen über Konzepte der Objektorientierung und grafische Benutzungsoberflächen bis zum Arbeiten mit Datenbanken und einfachen Client/Server-Anwendungen.

Ein Fallbeispiel fasst die Programmierung einer grafischen Oberfläche, Datenbankzugriffe und Netzwerkkommunikation in einem etwas umfangreicherem Programm zusammen.

Die 34 Kapitel dieses Grundkurses wurden in folgende Teile gegliedert:

- Grundlagen
- Objektorientierung
- Nützliche Klassen der Standardbibliothek
- Generische Programmierung
- Funktionale Programmierung
- Datenströme
- Nebenläufige Programmierung
- Grafische Oberflächen
- Datenbank- und Netzanwendungen
- Designvorgaben und Modularisierung
- Allgemeines

Die folgenden wichtigen Features von *Java 21* wurden in die passenden Kapitel integriert:

- *Pattern Matching bei instanceof und switch* (Kapitel 8.4)

Hiermit können komplexe Fallunterscheidungen durch Musterabgleich effizient implementiert werden.

- *Record Patterns* (Kapitel 8.4)

Mit Record Patterns können Records in ihre Bestandteile zerlegt werden, sodass beim Pattern Matching direkt mit den Komponenten weitergearbeitet werden kann.

- *Sequenced Collections* (Kapitel 20.6)

Sequenced Collections bieten einige neue Methoden für Collections, deren Elemente in einer wohldefinierten Reihenfolge geordnet sind.

- *Virtuelle Threads* (Kapitel 26.11)

Gegenüber den klassischen Plattform-Threads können mit virtuellen Threads viel mehr Threads in einem Prozess gestartet werden ohne den Speicher der JVM zu überlasten.

## Programmbeispiele und Aufgaben

Rund 250 Übungen unterstützen Sie dabei, die einzelnen Themen zu verstehen und zu vertiefen.

Der vollständige Quellcode der im Buch behandelten Programme und alle Lösungen zu den Aufgaben sind auf der Webseite zu diesem Buch verfügbar:

<https://www.springer.com>

sowie unter

<https://www.dietmar-abts.de> oder <https://github.com/abtsd/GKJava>

## Zielgruppen

Zielgruppen dieses Grundkurses sind:

- *Studierende* der Informatik und Wirtschaftsinformatik,
- *Beschäftigte* und *Auszubildende* in der IT-Branche, die bisher wenig Erfahrung mit der Entwicklung von Programmen haben, sowie
- *Umsteiger*, die bereits grundlegende Kenntnisse in einer anderen Programmiersprache haben.

Zum Erlernen einer Programmiersprache ist zum einen die Fähigkeit zum logisch-analytischen Denken, zum anderen aber auch der Wille zum selbstständigen Arbeiten und Ausprobieren am Computer unbedingt erforderlich.

Danken möchte ich meinen Leserinnen und Lesern für die konstruktive Kritik sowie Frau Bettina Saglio und Herrn Leonardo Milla vom Springer-Verlag für die gute Zusammenarbeit.

Nun wünsche ich Ihnen viel Spaß beim Lesen und Erarbeiten der Themen.

Ratingen, Dezember 2023

Dietmar Abts  
[dietmar.abts@t-online.de](mailto:dietmar.abts@t-online.de)

# Inhaltsverzeichnis

## I GRUNDLAGEN

<b>1 Einleitung.....</b>	1
1.1 Java.....	1
1.2 Vom Quellcode zum ausführbaren Programm.....	3
1.3 Entwicklungsumgebung.....	6
1.4 Zielsetzung und Aufbau des Buches.....	8
1.5 Programm- und Aufgabensammlung.....	10
1.6 Elementare Regeln und Konventionen.....	10

<b>2 Datentypen und Variablen.....</b>	13
2.1 Einfache Datentypen.....	13
2.2 Variablen.....	16
2.3 Aufgaben.....	18

<b>3 Operatoren.....</b>	19
3.1 Arithmetische Operatoren.....	19
3.2 Relationale Operatoren.....	22
3.3 Logische Operatoren.....	22
3.4 Bitoperatoren.....	24
3.5 Zuweisungsoperatoren.....	25
3.6 Bedingungsoperator.....	26
3.7 Cast-Operator.....	27
3.8 Aufgaben.....	28

<b>4 Kontrollstrukturen.....</b>	31
4.1 Verzweigungen.....	31
4.2 Schleifen.....	35
4.3 Sprunganweisungen.....	38
4.4 Die Java-Shell.....	40
4.5 Aufgaben.....	40

<b>5 Arrays.....</b>	45
5.1 Definition und Initialisierung.....	45
5.2 Zugriff auf Array-Elemente.....	46
5.3 Kommandozeilen-Parameter.....	49
5.4 Aufgaben.....	50

## II OBJEKTOIENTIERUNG

<b>6 Klassen, Objekte und Methoden.....</b>	51
6.1 Klassen und Objekte.....	51

6.2 Methoden.....	56
6.3 Konstruktoren.....	60
6.4 Klassenvariablen und Klassenmethoden.....	63
6.5 Varargs.....	65
6.6 Iterative und rekursive Problemlösungen.....	66
6.7 Records.....	68
6.8 Aufgaben.....	69
 <b>7 Pakete</b> .....	75
7.1 Pakete erzeugen und nutzen.....	75
7.2 Eigene Java-Bibliotheken erzeugen.....	78
7.3 Aufgaben.....	80
 <b>8 Vererbung</b> .....	81
8.1 Klassen erweitern.....	81
8.2 Konstruktoren und Vererbung.....	84
8.3 Methodenauswahl zur Laufzeit.....	87
8.4 Pattern Matching.....	88
8.5 Abstrakte Klassen.....	91
8.6 Modifizierer.....	93
8.7 Versiegelte Klassen.....	95
8.8 Aufgaben.....	95
 <b>9 Interfaces</b> .....	99
9.1 Interfaces definieren und einsetzen.....	99
9.2 Default-Methoden.....	105
9.3 Statische und private Methoden in Interfaces.....	110
9.4 Aufgaben.....	111
 <b>10 Vererbung vs. Delegation</b> .....	115
10.1 Warum Vererbung problematisch sein kann.....	115
10.2 Delegation als Alternative.....	117
10.3 Aufgaben.....	119
 <b>11 Geschachtelte Klassen</b> .....	121
11.1 Statische Klasse.....	121
11.2 Instanzklasse.....	122
11.3 Lokale Klasse.....	124
11.4 Anonyme Klasse.....	126
11.5 Aufgaben.....	128
 <b>12 Konstanten und enum-Aufzählungen</b> .....	129
12.1 Verwendung von int-Konstanten.....	129
12.2 Einfache enum-Aufzählung.....	131

12.3 enum-Aufzählung mit Attributen und Methoden.....	132
12.4 Konstantenspezifische Implementierung von Methoden.....	134
12.5 Singleton.....	134
12.6 Aufgaben.....	136
<b>13 Ausnahmebehandlung.....</b>	<b>137</b>
13.1 Ausnahmetypen.....	137
13.2 Auslösung und Weitergabe von Ausnahmen.....	140
13.3 Auffangen von Ausnahmen.....	143
13.4 Verkettung von Ausnahmen.....	146
13.5 Aufgaben.....	148
 III. NÜTZLICHE KLASSEN DER STANDARDBIBLIOTHEK	
<b>14 Zeichenketten.....</b>	<b>151</b>
14.1 Die Klasse String.....	151
14.2 Mehrzeilige Texte.....	160
14.3 Die Klassen StringBuilder und StringBuffer.....	161
14.4 Die Klasse StringTokenizer.....	163
14.5 Aufgaben.....	164
<b>15 Ausgewählte Klassen.....</b>	<b>167</b>
15.1 Wrapper-Klassen für einfache Datentypen.....	167
15.2 Die Klasse Object.....	173
15.3 Die Klassen Vector, Hashtable und Properties.....	180
15.4 Die Klasse System.....	188
15.5 Die Klasse Class.....	191
15.6 Die Klasse Arrays.....	196
15.7 Mathematische Funktionen.....	199
15.8 Datum und Uhrzeit.....	204
15.9 Aufgaben.....	214
<b>16 Internationalisierung.....</b>	<b>221</b>
16.1 Die Klasse Locale.....	221
16.2 Zeit und Zahlen darstellen.....	224
16.3 Sprachspezifische Sortierung.....	227
16.4 Ressourcenbündel.....	228
16.5 Aufgaben.....	229
<b>17 Services.....</b>	<b>231</b>
17.1 Service Provider.....	231
17.2 Service Consumer.....	234
17.3 Aufgaben.....	234

<b>18 Javadoc</b> .....	237
18.1 javadoc-Syntax.....	237
18.2 Das Werkzeug javadoc.....	239
18.3 Aufgaben.....	240
 IV. GENERISCHE PROGRAMMIERUNG	
<b>19 Generische Typen und Methoden</b> .....	241
19.1 Einführung.....	241
19.2 Typeinschränkungen.....	244
19.3 Raw Types.....	248
19.4 Wildcards.....	249
19.5 Generische Methoden.....	254
19.6 Grenzen des Generics-Konzepts.....	255
19.7 Aufgaben.....	256
<b>20 Collections</b> .....	259
20.1 Die Interfaces Collection, Iterable und Iterator.....	259
20.2 Listen.....	261
20.3 Mengen.....	264
20.4 Schlüsseltabellen.....	265
20.5 Erzeugung von Collections für vordefinierte Werte.....	268
20.6 Sequenced Collections.....	269
20.7 Aufgaben.....	270
 V. FUNKTIONALE PROGRAMMIERUNG	
<b>21 Lambda-Ausdrücke</b> .....	275
21.1 Funktionsinterfaces.....	276
21.2 Lambdas.....	278
21.3 Methodenreferenzen.....	282
21.4 Weitere Beispiele.....	285
21.5 Aufgaben.....	293
<b>22 Streams</b> .....	295
22.1 Das Stream-Konzept.....	295
22.2 Erzeugung von Streams.....	296
22.3 Transformation und Auswertung.....	297
22.4 Aufgaben.....	301
<b>23 Optionale Werte</b> .....	307
23.1 Motivation.....	307
23.2 Die Klasse Optional.....	308
23.3 Weitere Beispiele.....	313

---

23.4 Aufgaben.....	315
--------------------	-----

## VI. DATENSTRÖME

<b>24 Dateien und Datenströme.....</b>	317
24.1 Dateien und Verzeichnisse bearbeiten.....	317
24.2 Zeichensätze und Codierungen.....	325
24.3 Datenströme.....	328
24.4 Daten lesen und schreiben.....	334
24.5 Texte lesen und schreiben.....	341
24.6 Datenströme filtern.....	347
24.7 Wahlfreier Dateizugriff.....	349
24.8 Datenkomprimierung.....	352
24.9 Aufgaben.....	357
<b>25 Serialisierung.....</b>	363
25.1 Serialisieren und Deserialisieren.....	363
25.2 Sonderbehandlung bei Serialisierung.....	366
25.3 Das Datenformat JSON.....	369
25.4 Aufgaben.....	376

## VII. NEBENLÄUFIGE PROGRAMMIERUNG

<b>26 Nebenläufige Programmierung.....</b>	379
26.1 Threads erzeugen, starten und beenden.....	379
26.2 Lebenszyklus eines Threads.....	388
26.3 Synchronisation.....	388
26.4 Fallbeispiel Summenberechnung.....	394
26.5 Thread-Sicherheit.....	397
26.6 Kommunikation zwischen Threads.....	403
26.7 Shutdown-Threads.....	413
26.8 Thread-Pools und Executors.....	415
26.9 Asynchrone Verarbeitung mit CompletableFuture.....	418
26.10 Das Process-API.....	421
26.11 Virtuelle Threads.....	423
26.12 Aufgaben.....	426

## VIII. GRAFISCHE OBERFLÄCHEN

<b>27 GUI-Programmierung mit Swing.....</b>	435
27.1 Ein Fenster erzeugen mit JFrame.....	437
27.2 Ereignisbehandlung.....	440
27.3 Layouts.....	446
27.4 Buttons und Labels.....	458
27.5 Spezielle Container.....	463

27.6 Textkomponenten.....	467
27.7 Auswahlkomponenten.....	472
27.8 Menüs und Symbolleisten.....	476
27.9 Dialoge.....	482
27.10 Tabellen.....	488
27.11 Event-Queue und Event-Dispatcher.....	493
27.12 Aufgaben.....	499
 <b>28 Einführung in JavaFX.....</b>	 505
28.1 Installation und Konfiguration.....	505
28.2 Ein erstes Beispiel.....	507
28.3 Beispiel Brutto-Rechner.....	510
28.4 Asynchrone Verarbeitung.....	519
28.5 Diagramme.....	527
28.6 Tabellen.....	533
28.7 Aufgaben.....	536
 <b>IX. DATENBANK- UND NETZANWENDUNGEN</b>	
 <b>29 Datenbankzugriffe mit JDBC.....</b>	 541
29.1 Voraussetzungen.....	541
29.2 Datenbank erstellen.....	543
29.3 Daten einfügen.....	545
29.4 Daten ändern und löschen.....	547
29.5 Daten abrufen.....	549
29.6 Daten in einer Tabelle anzeigen.....	550
29.7 Daten in einem Diagramm präsentieren.....	557
29.8 Aufgaben.....	561
 <b>30 Netzwerkkommunikation.....</b>	 565
30.1 Dateien aus dem Netz laden.....	565
30.2 Eine einfache Client/Server-Anwendung.....	567
30.3 HTTP.....	571
30.4 Simple Web Server.....	577
30.5 Das HTTP-Client-API.....	579
30.6 Aufgaben.....	582
 <b>31 Fallbeispiel.....</b>	 585
31.1 Die Anwendung.....	585
31.2 Variante 1: Lokale Anwendung.....	588
31.3 Variante 2: Client/Server-Anwendung.....	601
31.4 Aufgaben.....	614

**X. DESIGNVORGABEN UND MODULARISIERUNG**

<b>32 Prinzipien objektorientierten Designs.....</b>	615
32.1 Single-Responsibility-Prinzip.....	615
32.2 Open-Closed-Prinzip.....	618
32.3 Liskovsches Substitutionsprinzip.....	621
32.4 Interface-Segregation-Prinzip.....	624
32.5 Dependency-Inversion-Prinzip.....	627
32.6 Zusammenfassung.....	630
32.7 Aufgaben.....	631

<b>33 Einführung in das Modulsystem.....</b>	635
----------------------------------------------	-----

33.1 Motivation.....	635
33.2 Grundlagen.....	636
33.3 Abhängigkeiten und Zugriffsschutz.....	637
33.4 Transitive Abhängigkeiten.....	643
33.5 Abhängigkeit von JDK-Modulen und anderen Modulen.....	643
33.6 Trennung von Schnittstelle und Implementierung.....	647
33.7 Modularisierung und Services.....	650
33.8 Einbindung nicht-modularer Bibliotheken.....	653
33.9 Modulkategorien.....	656
33.10 Aufgaben.....	658

**XI. ALLGEMEINES**

<b>34 IntelliJ IDEA, Debugging und Testen mit JUnit.....</b>	661
--------------------------------------------------------------	-----

34.1 Hinweise zu IntelliJ IDEA.....	661
34.2 Debugging.....	666
34.3 Testen mit JUnit.....	668

Literaturhinweise.....	673
------------------------	-----

Sachwortverzeichnis.....	675
--------------------------	-----



# 1 Einleitung

Die Programmiersprache *Java*, erschienen 1995, ist mittlerweile schon über 25 Jahre alt. Sie gehört noch immer zu den populärsten Programmiersprachen und liegt in den Rankings stets auf einem der obersten Plätze.<sup>1</sup>

In den letzten Jahren sind neue Programmiersprachen entstanden, die von Java vieles gelernt haben und zum Teil Sprach-Features verbessert sowie neue hinzugefügt haben, um den Komfort und die Sicherheit bei der Entwicklung zu erhöhen (z. B. die Programmiersprache *Kotlin*).

Auch Java hat sich weiterentwickelt, ist aber trotz neuer Konzepte in der Anwendung einfach geblieben und für den Anfänger gut zu erlernen.

## 1.1 Java

*Java* ist eine Programmiersprache, bezeichnet aber auch eine Plattform zur Ausführung von Programmen unabhängig vom zugrunde liegenden Betriebssystem.

Zur *Java-Technologie* gehören

- die Programmiersprache *Java*,
- das Werkzeug *Java Development Kit* (JDK) mit Compiler und Bibliotheken und
- die Java-Laufzeitumgebung JRE (*Java Runtime Environment*) zur Ausführung der Programme.

### Merkmale

Insbesondere die folgenden Eigenschaften zeichnen die Sprache Java aus. Diese Merkmale sind hier nur kurz zusammengestellt. Sie können vom Leser, der noch keine Programmiererfahrung hat, erst im Laufe der Beschäftigung mit den verschiedenen Themenbereichen dieses Buches ausreichend verstanden und eingeordnet werden.

- Java ist eine *objektorientierte Sprache*. Sie unterstützt alle zentralen Aspekte der Objektorientierung wie Klassen, Objekte und Vererbung.
- Die Sprache ist bewusst einfach gehalten. Im Unterschied zu C++ gibt es in Java z. B. keine expliziten Zeiger und keine Mehrfachvererbung.

---

<sup>1</sup> TIOBE-Index: <https://www.tiobe.com/tiobe-index>  
PYPL-Index: <https://pypl.github.io/PYPL.html>

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_1](https://doi.org/10.1007/978-3-658-43574-5_1).

- Java ist eine *typisierte Sprache*. Bei der Übersetzung in den *Bytecode* werden Überprüfungen der verwendeten Datentypen ausgeführt und Inkonsistenzen erkannt.
- Die in einem Programm benutzten Klassen können an unterschiedlichen Orten liegen. Sie werden erst zur Laufzeit des Programms bei Bedarf geladen.
- Das Speichermanagement erfolgt automatisch. Während der Laufzeit eines Programms kann der Speicherplatz für nicht mehr benötigte Objekte vom Laufzeitsystem automatisch freigegeben werden (*Garbage Collection*).
- In Java gibt es eine strukturierte Behandlung von Laufzeit-Fehlern, die während der Abarbeitung eines Programms auftreten können. So wird z. B. während der Laufzeit die Einhaltung von Indexgrenzen beim Zugriff auf Arrays überwacht.
- Java unterstützt den parallelen Ablauf von eigenständigen Programmabschnitten (*Multithreading*) und die Synchronisation bei konkurrierenden Datenzugriffen.
- Die Java-Standardbibliothek bietet eine Vielzahl von Möglichkeiten für die Netzwerkkommunikation auf Basis des Protokolls *TCP/IP*.
- Die Java-Pakete stellen darüber hinaus eine Vielzahl nützlicher APIs (*Application Programming Interfaces*) in Form von Klassen und Interfaces für die Anwendungsentwicklung zur Verfügung.

## Einsatzgebiete

Java wird als universelle Programmiersprache für eine Vielzahl von Anwendungen eingesetzt:

- Desktop-Anwendungen,
- Server-seitige Dienste im Internet (Application Server, Web Services),
- Entwicklung von Unternehmenssoftware,
- mobile Anwendungen (Apps auf der Basis des Betriebssystems Android),
- in technische Systeme (Geräte der Unterhaltungselektronik, der Medizintechnik usw.) eingebettete Anwendungen.

## Releasezyklus

Alle sechs Monate erscheint eine neue Java-Version. Die bis zu diesem Zeitpunkt fertiggestellten Features werden veröffentlicht. Dabei können neue Features zunächst als *Preview* zum Ausprobieren vorgestellt und bis zur endgültigen Veröffentlichung verbessert werden.

*LTS-Versionen* (LTS = Long-Term Support), wie z. B. Java 17 und Java 21, werden über einen längeren Zeitraum mit Sicherheits-Updates und Fehlerbehebungen versorgt und sind damit für den produktiven Einsatz besonders geeignet.

## Distributionen

Es existieren verschiedene Distributionen, die sich in ihrer Support- und Lizenzpolitik unterscheiden.

Beispiele sind:

- *Oracle JDK*<sup>2</sup>,
- *OpenJDK*<sup>3</sup> und
- *Eclipse Temurin*<sup>4</sup>.

Zumindest sind alle in Entwicklungs- und Testumgebungen kostenfrei nutzbar.

Die Distributionen liegen unter den in den Fußnoten aufgeführten Webadressen zum Herunterladen bereit. Beachten Sie die Betriebssystem-spezifischen Installationshinweise.

Welche Version auf Ihrem System installiert ist, können Sie jederzeit in der Eingabeaufforderung bzw. im Terminal wie folgt herausfinden:

```
javac -version bzw.  
java -version
```

## 1.2 Vom Quellcode zum ausführbaren Programm

Plattformunabhängigkeit ist eine wichtige Eigenschaften von Java, zusammengefasst in dem Slogan: *Write once – run anywhere*.

Das vom Java-Compiler aus dem Quellcode erzeugte Programm, der sogenannte *Bytecode*, ist unabhängig von der Rechnerarchitektur und läuft auf jedem Rechner, auf dem eine spezielle Software, die Java Virtual Machine (JVM), existiert. Diese JVM ist für jedes gängige Betriebssystem verfügbar.

### Java Virtual Machine

Die *virtuelle Maschine* JVM stellt eine Schicht zwischen dem Bytecode und der zu Grunde liegenden Rechnerplattform dar. Ein Interpreter übersetzt die Bytecode-Befehle in plattformspezifische Prozessorbefehle. Die JVM kann die Ausführung

---

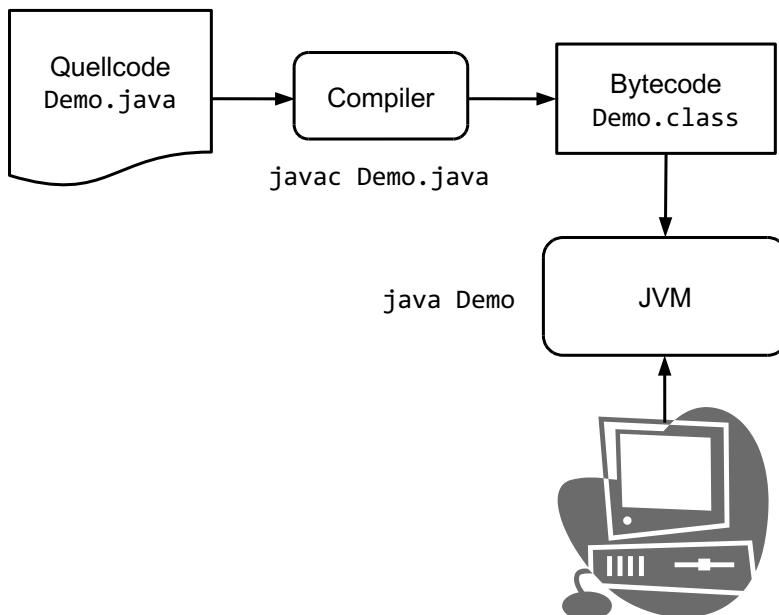
2 <https://www.oracle.com/java/technologies/downloads>

3 <https://openjdk.org>

4 <https://adoptium.net>

der Java-Programme überwachen und verhindern, dass Befehle ausgeführt werden, die die Sicherheit des Systems gefährden.

Zur Leistungssteigerung von wiederholt auszuführenden Programmteilen wird ein Compiler eingesetzt, der zur Laufzeit Performance-kritische Teile des Bytecodes vor der Programmausführung in Prozessorbefehle übersetzt. Neben Java können auch andere Sprachen (wie z. B. Kotlin) auf einer JVM laufen.



**Abbildung 1-1:** Compilierung und Ausführung

Wir wollen die einzelnen Schritte vom Quellcode bis zur Ausführung des Programms anhand eines kleinen Beispiels demonstrieren. Dazu reicht die Nutzung eines einfachen Texteditors und Terminalfensters zur Eingabe auf Kommandozeilenebene. Der Quellcode wird im Anschluss vorgestellt.

### Die einzelnen Schritte

1. Quellcode mit einem Editor erstellen (`Demo.java`).
2. Quellcode compilieren (`javac Demo.java`). Sofern kein Fehler gemeldet wird, liegt nun der Bytecode in `Demo.class` vor.
3. Bytecode ausführen (`java Demo`).

Die Datei *Demo.java* hat den folgenden Inhalt:

```
public class Demo {  
    public static void main(String[] args) {  
        System.out.println("Viel Erfolg mit dem Grundkurs Java");  
    }  
}
```

Um ein lauffähiges Programm zu erhalten, müssen hier bereits Sprachelemente (beispielsweise `class`) benutzt werden, die erst später ausführlich behandelt werden können.

Programmieren kann man leider nicht linear erlernen. Immer muss am Anfang etwas vorausgesetzt werden. Die Verständlichkeit wird allerdings hierdurch nicht beeinträchtigt.

Die Beispiele in den ersten Kapiteln dieses Buches nutzen alle den folgenden Programmrahmen.

```
public class Klasse {  
    public static void main(String[] args) {  
  
        // Hier stehen Anweisungen  
  
    }  
}
```

In Java ist der Quellcode hauptsächlich in sogenannten Klassen strukturiert. Auf `class` folgt der Name der Klasse (hier der Programmname). Die Klasse ist `public`, d. h. öffentlich, von außen zugänglich.

Code-Blöcke sind mit geschweiften Klammern `{` und `}` begrenzt.

Die Methode `main` enthält die auszuführenden Anweisungen. Die Methode ist `public`, also von außen zugänglich, sie ist `static` (statisch), d. h. der Code der Methode kann ausgeführt werden, ohne dass ein Objekt (eine Instanz) der Klasse erzeugt werden muss. Die Methode liefert kein Ergebnis (`void`).

Die Methode wird mit einem Parameter, der hier `args` genannt ist, aufgerufen. An diesen Parameter können mehrere, durch Leerzeichen getrennte Zeichenketten übergeben werden (siehe Kapitel 5.3). Der Typ `String[]` steht für ein Array (Reihung) von sogenannten Strings (Zeichenketten).

Der Name der Datei, in der sich der Quellcode befindet, muss dem Klassennamen, ergänzt um die Endung `.java`, entsprechen: `Klasse.java`

In der Klasse `Demo` gibt der Aufruf der Methode `System.out.println` eine Zeichenkette am Bildschirm aus.



```
Datei Bearbeiten Ansicht Suchen Terminal Hilfe
$ javac Demo.java
$ java Demo
Viel Erfolg mit dem Grundkurs Java
$ █
```

**Abbildung 1-2:** Compilieren und Ausführen auf Kommandozeilenebene

### Ausführung von Java-Programmen, die aus nur einer einzigen Datei bestehen

Wie Sie später sehen werden, bestehen größere Programme aus mehreren Dateien. In der Regel wird jede Klasse in einer eigenen Datei codiert.

Programme, die *nur aus einer Datei bestehen*, können direkt mit

```
java Quellcode.java
```

gestartet werden. Es findet eine *In-memory-Compilierung* statt, *Quellcode.class* wird also nicht dauerhaft gespeichert.

Beispiel:

```
java Demo.java
```

Auf diese einfache Weise können auch die Programme aus Kapitel 2 bis 5 aufgerufen werden, was den Einstieg in Java erleichtert, zumal hier noch keine Entwicklungsumgebung nötig ist.

## 1.3 Entwicklungsumgebung

Für Java werden verschiedene integrierte Entwicklungsumgebungen (*Integrated Development Environment*, IDE) zur Unterstützung von Projekten mit zahlreichen Werkzeugen angeboten, beispielsweise *IntelliJ IDEA* von JetBrains, *Eclipse IDE*, *Apache NetBeans* und *Visual Studio Code* von Microsoft.

### IntelliJ IDEA

IntelliJ IDEA ist derzeit die populärste Entwicklungsumgebung für Java. IntelliJ bietet eine Vielzahl von Funktionen nicht nur für Profis, sondern auch für Anfänger, z. B. Codevervollständigung, Korrekturvorschläge, Generierung von Code.

Zur Entwicklung der Programmbeispiele und Lösungen zu den Aufgaben wurde die *Community Edition* von *IntelliJ IDEA*<sup>5</sup> verwendet. Die Programme im Begleitmaterial sind gemäß den Projekt-Konventionen dieser IDE abgelegt. Die Quell-

---

5 <https://www.jetbrains.com/idea>

codes können natürlich auch für eine andere Entwicklungsumgebung übernommen werden.

Die *Community Edition* von IntelliJ ist frei verfügbar.<sup>6</sup> Installationsanleitungen stehen für Windows, macOS und Linux zur Verfügung.

Kapitel 34 dieses Buches enthält einige Hinweise zur Konfiguration der IDE und zum Einrichten der Projekte.

Ein IntelliJ-Projekt kann in sogenannte Module zerlegt werden. Hierbei handelt es sich um Unterprojekte, deren Code gemeinsam compiliert werden kann. Ein IntelliJ-Modul hat nichts mit dem Begriff Modul im Kapitel 33 (Einführung in das Modulsystem) zu tun. IntelliJ-Module werden aber in IntelliJ verwendet, um Java-Module (im Sinne von Kapitel 33) zu erstellen.

Die Programme und die Lösungen zu den Aufgaben wurden mit der Distribution *OpenJDK* entwickelt. Es wurde die zur Zeit der Drucklegung dieses Buches aktuelle Java-Version 21 verwendet.

### Eingabe von Kommandos im Terminal

Im Buch werden Kommandos zur Ausführung von Programmen etc. für das Betriebssystem *Linux* dargestellt.

Unterschiede zu anderen Betriebssystemen bestehen bei der Verwendung von mehrzeiligen Befehlen, Umgebungsvariablen und Trennern für Dateipfade und Klassenpfade.

- *Mehrzeilige Befehle*

Längere Befehle können auf mehrere Zeilen verteilt werden. Hierzu wird am Ende einer Zeile ein Leerzeichen (Blank) sowie ein Sonderzeichen eingegeben, für Windows ^ , für macOS und Linux \ .

- *Umgebungsvariablen*

Für Windows:

Setzen mit `set variable=wert`

Lesen mit `%variable%`

Für macOS und Linux:

Setzen mit `export variable=wert`

Lesen mit `$variable`

- *Trenner für Dateipfade*

Als Trenner für Dateipfade wird bei Windows der Backslash \ und bei macOS und Linux der normale Schrägstrich / verwendet.

---

6 <https://www.jetbrains.com/idea/download>

- *Trenner für Klassenpfade*

Sollen mehrere Pfade für den Klassenpfad (`CLASSPATH`) gesetzt werden, werden diese bei Windows durch ein Semikolon ; und bei macOS und Linux durch einen Doppelpunkt : getrennt.

## 1.4 Zielsetzung und Aufbau des Buches

Der *Grundkurs Java* bietet eine strukturierte und mit zahlreichen Beispielen und Aufgaben versehene Einführung in verschiedene Aspekte der Java-Programmierung. Das Buch kann nicht die gesamte Java-Klassenbibliothek vorstellen. Die für das Grundverständnis wichtigen Klassen und Methoden werden behandelt. Eine vollständige Beschreibung aller Klassen findet man in einschlägigen Referenzhandbüchern und in der Online-Dokumentation<sup>7</sup> zur *Java Standard Edition*.

Obwohl dieser Kurs keine Erfahrung in der Programmierung voraussetzt, erleichtern Grundkenntnisse in einer anderen Programmiersprache den Einstieg.

Die Kapitel 1 bis 33 des Buches bauen aufeinander auf und sollten deshalb idealerweise in der durch die Gliederung vorgegebenen Reihenfolge erarbeitet werden.

Das Buch besteht aus 11 Teilen, die zusammengehörende Aspekte der Programmierung und Anwendung vereinen.

### I. GRUNDLAGEN

Kapitel 2 bis 5 beschäftigen sich mit den *imperativen* (nicht-objektorientierten) Sprachkonzepten, wie einfache Datentypen, Variablen, Operatoren, Kontrollstrukturen und Arrays.

### II. OBJEKTOIENTIERUNG

Kapitel 6 bis 13 führen die *objektorientierten* Sprachkonzepte (Klassen, Records, Objekte, Methoden, Konstruktoren, Vererbung, abstrakte Klassen, Interfaces, innere Klassen und Aufzählungstypen) ein, deren Verständnis grundlegend für alles Weitere ist. Hier werden auch die Möglichkeiten zur Behandlung von Fehlern bzw. Ausnahmesituationen (*Exceptions*), die während der Programmausführung auftreten können, behandelt.

### III. NÜTZLICHE KLASSEN DER STANDARDBIBLIOTHEK

Kapitel 14 bis 18 stellen Zeichenketten (*Strings*) sowie zahlreiche nützliche Klassen der Klassenbibliothek vor, die in Anwendungen häufig verwendet werden. Hier werden u. a. die Auslagerung sprachabhängiger Texte (Internationalisierung) und der Service-Provider-Mechanismus vorgestellt, der die Abhängigkeit von konkreten Implementierungen von Interfaces reduziert.

---

7 <https://docs.oracle.com/en/java/javase/nn> (nn = Java-Version)

**IV. GENERISCHE PROGRAMMIERUNG**

Kapitel 19 und 20 führen in das Thema *Generics* (generische Typen und Methoden) ein und präsentieren einige oft benutzte Interfaces und Klassen des *Collection Frameworks*.

**V. FUNKTIONALE PROGRAMMIERUNG**

Kapitel 21 bis 23 behandeln *Lambda*-Ausdrücke, die im gewissen Sinne eine "funktionale Programmierung" ermöglichen, *Streams* zur Verarbeitung von Daten "am Fließband" und die Klasse *Optional* zur besseren Behandlung optionaler Werte.

**VI. DATENSTRÖME**

Kapitel 24 bis 25 enthalten die für das Lesen und Schreiben von Dateien (Datenströme) und die Verarbeitung von Dateien und Verzeichnissen wichtigen Klassen und Methoden.

**VII. NEBENLÄUFIGE PROGRAMMIERUNG**

Kapitel 26 bietet eine Einführung in die Programmierung mehrerer gleichzeitig laufender Anweisungsfolgen (*Threads*) innerhalb eines Programms, zeigt wie der Einsatz der Klasse *CompletableFuture* die Programmierung asynchroner Abläufe vereinfachen kann und wie Prozesse des Betriebssystems kontrolliert werden können.

**VIII. GRAFISCHE OBERFLÄCHEN**

Kapitel 27 und 28 befassen sich mit der Entwicklung von grafischen Oberflächen mit *Swing* sowie der Behandlung von Ereignissen. Zudem wird das Java-Framework *JavaFX* zur Realisierung von GUIs vorgestellt und gezeigt, wie mit einer auf XML basierenden Sprache (*FXML*), einem Design-Tool und CSS (Cascading Style Sheets) moderne User Interfaces erstellt werden können.

**IX. DATENBANK- UND NETZANWENDUNGEN**

Kapitel 29 enthält eine Einführung in die Programmierschnittstelle *JDBC* für den Zugriff auf relationale Datenbanken mit Hilfe von SQL.

Kapitel 30 beschäftigt sich mit der Programmierung von Client/Server-Anwendungen auf der Basis von TCP/IP und der Socket-Schnittstelle.

Kapitel 31 enthält ein etwas umfangreicheres Fallbeispiel. Dabei wird eine zunächst lokale Anwendung schrittweise in eine Client/Server-Anwendung überführt.

**X. DESIGNVORGABEN UND MODULARISIERUNG**

Kapitel 32 stellt fünf wichtige Prinzipien (die SOLID-Prinzipien) zum Entwurf und zur Entwicklung von Programmen vor. Kapitel 33 enthält abschließend eine Einführung in das mit Java 9 eingeführte Modulsystem.

## XI. ALLGEMEINES

Kapitel 34 enthält Hinweise zum Umgang mit der Entwicklungsumgebung *IntelliJ IDEA*, zum Debugging und zum Testen mit dem Framework JUnit.

Literaturhinweise bieten Quellen zur Vertiefung einzelner Themen.

## 1.5 Programm- und Aufgabensammlung

Zahlreiche Programmbeispiele helfen bei der Umsetzung der Konzepte in lauffähige Anwendungen. Aufgaben am Ende eines Kapitels ermöglichen, den behandelten Stoff einzüben und ggf. zu vertiefen.

Alle Programme wurden mit dem JDK für die zur Zeit der Drucklegung aktuelle Java-Version 21 unter Linux getestet.

Sämtliche Programme und Lösungen zu den Aufgaben stehen zum Download zur Verfügung. Hinweise zu weiteren Tools und Bibliotheken erfolgen in den Kapiteln, in denen sie erstmalig benutzt werden.

Den Zugang zum Begleitmaterial finden Sie auf der Webseite zu diesem Buch:

<https://www.springer.com>

sowie unter

<https://www.dietmar-abts.de> oder <https://github.com/abtsd/GKJava>

Extrahieren Sie nach dem Download alle Dateien des ZIP-Archivs in ein von Ihnen gewähltes Verzeichnis Ihres Rechners.

Die Programme (Klassen, Ressourcen und sonstige Artefakte) wurden in Form von Projekten für die Entwicklungsumgebung *IntelliJ IDEA* geordnet nach Kapitelnummern organisiert.

Kapitel 34 enthält kurze Hinweise zum Umgang mit *IntelliJ IDEA* für die hier verwendeten Projekte. Obwohl die Projekte gemäß den Konventionen und Erfordernissen von *IntelliJ IDEA* eingerichtet sind, können sie auch für andere Entwicklungsumgebungen nach leichter Anpassung genutzt werden.

## 1.6 Elementare Regeln und Konventionen

Wie jede Sprache hat auch Java ihre eigenen Regeln und Konventionen, die festlegen, wie Ausdrücke, Anweisungen, Namen für Klassen, Methoden usw. gebildet werden. In diesem Abschnitt kommen auch einige Begriffe vor, die erst in den folgenden Kapiteln näher erläutert werden können.

## Syntaxregeln

- Code wird durch geschweifte Klammern { und } in Blöcken strukturiert (siehe auch die weiter oben aufgeführten Programmbeispiele). Hierbei sollte der Code auch mit Einrückungen versehen sein, um die Lesbarkeit zu erhöhen.
- Java-Anweisungen müssen mit einem Semikolon ; abgeschlossen werden. Anweisungen dürfen sich über mehrere Zeilen erstrecken.
- Zeichenketten werden in doppelte Anführungszeichen " eingeschlossen.

## Bezeichner

- *Bezeichner* sind Namen für vom Programmierer definierte Elemente wie Variablen, Marken, Klassen, Records, Interfaces, Aufzählungstypen (enum), Methoden und Pakete. Sie können aus beliebig vielen Unicode-Buchstaben und Ziffern bestehen, müssen aber mit einem Unicode-Buchstaben beginnen. Der Unterstrich \_ und das Dollar-Zeichen \$ sind als erstes Zeichen eines Bezeichners zulässig. Es wird zwischen Groß- und Kleinschreibung der Namen unterschieden. Die Bezeichner dürfen nicht mit den Schlüsselwörtern der Sprache und den sogenannten Literalen true, false und null übereinstimmen.
- Einige Wörter sind als *Schlüsselwörter* für Java reserviert.<sup>8</sup> Sie dürfen nicht als Bezeichner verwendet werden. Das sind Wörter, die in Java selbst schon eine Bedeutung haben, z. B. if, else, for, while.
- Zur Erläuterung: Ein *Literal* ist eine Zeichenfolge, die den Wert eines einfachen Datentyps direkt darstellt.

## Namenskonventionen

*Namenskonventionen* erhöhen die Lesbarkeit von Programmen. Sie werden nicht von Java erzwungen, gehören aber zu einem guten Programmierstil. Folgende Regeln haben sich durchgesetzt:

- Namen werden mit gemischten Groß- und Kleinbuchstaben geschrieben, wobei in zusammengesetzten Wörtern die einzelnen Wortanfänge durch große Anfangsbuchstaben kenntlich gemacht werden (*CamelCase*).
- *Paketnamen* enthalten nur Kleinbuchstaben und Ziffern.<sup>9</sup>
- *Namen für Klassen, Records, Interfaces und Aufzählungstypen (enum)* beginnen mit einem Großbuchstaben, z. B. MyFirstExample. Da Klassennamen als Teil des Namens der Datei auftauchen, die die entsprechende

<sup>8</sup> [https://en.wikipedia.org/wiki/List\\_of\\_Java\\_keywords](https://en.wikipedia.org/wiki/List_of_Java_keywords)

<sup>9</sup> Abweichend hiervon benutzen wir in den Projekten auch den Unterstrich \_ .

Klasse im Quell- bzw. Bytecode enthält, unterliegen diese auch den Regeln des jeweiligen Dateisystems.

- *Variablennamen* beginnen mit einem Kleinbuchstaben, z. B. `myAccount`. Namen von *Konstanten* (Variablen mit unveränderbarem Wert) bestehen aus Großbuchstaben, wobei einzelne Wörter durch `_` getrennt werden, beispielsweise `KEY_FIRST`.
- *Methoden* werden in der Regel nach Verben benannt, ihre Namen beginnen ebenfalls mit einem Kleinbuchstaben, z. B. `compute`, `drawFigure`.

## Kommentare

*Kommentare* im Quellcode sind frei formulierte Texte, die dem Leser hilfreiche Hinweise geben können. Sie werden vom Compiler ignoriert.

Java kennt drei Arten von Kommentaren:

- *Einzeiliger Kommentar*  
Dieser beginnt mit den Zeichen `//` und endet am Ende der aktuellen Zeile.  
Beispiel:  

```
int z; // Anzahl gelesener Zeilen
```
- *Mehrzeiliger Kommentar*  
Dieser beginnt mit `/*`, endet mit `*/` und kann sich über mehrere Zeilen erstrecken.  
Beispiel:  

```
/* Diese Zeilen stellen
   einen mehrzeiligen Kommentar dar
*/
```

Die Positionierung im Quelltext ist vollkommen frei, `/*` und `*/` müssen nicht direkt am Zeilenanfang stehen.

- *Dokumentationskommentar*  
Dieser beginnt mit `/**`, endet mit `*/` und kann sich ebenfalls über mehrere Zeilen erstrecken. Er wird vom JDK-Tool `javadoc` zur automatischen Generierung von Programmdokumentation verwendet (siehe Kapitel 18).



## 2 Datentypen und Variablen

Damit ein Programm Berechnungen und andere Verarbeitungen vornehmen kann, muss es vorübergehend Zahlen, Zeichenketten und andere Daten speichern. Dazu dienen Variablen.

### Lernziele

In diesem Kapitel lernen Sie

- welche Datentypen zur Speicherung von Zeichen und Zahlen in Java existieren und
- wie Variablen definiert und initialisiert werden können.

### 2.1 Einfache Datentypen

Daten werden in Programmen durch *Variablen* repräsentiert. Einer Variablen entspricht ein Speicherplatz im Arbeitsspeicher, in dem der aktuelle Wert der Variablen abgelegt ist. Variablen können nur ganz bestimmte Werte aufnehmen. Dies wird durch den *Datentyp* festgelegt.

Der Datentyp legt also fest, welche Werte erlaubt sind und welche Operationen mit Werten dieses Typs möglich sind.

Java kennt acht sogenannte *einfache (primitive) Datentypen*:

**Tabelle 2-1:** Einfache Datentypen

Datentyp	Größe in Byte	Wertebereich
boolean	1	false, true
char	2	0 ... 65.535
byte	1	-128 ... 127
short	2	-32.768 ... 32.767
int	4	-2.147.483.648 ... 2.147.483.647
long	8	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
float	4	Absolutbetrag $1,4 \cdot 10^{-45}$ ... $3,4028235 \cdot 10^{38}$
double	8	Absolutbetrag $4,9 \cdot 10^{-324}$ ... $1,7976931348623157 \cdot 10^{308}$

Die genaue Größe für boolean ist abhängig von der jeweiligen JVM.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_2](https://doi.org/10.1007/978-3-658-43574-5_2).

## Literale

Ein *Literal* ist ein direkt angegebener Wert, wie z. B. 12.345.

## Wahrheitswerte

Der *logische Typ boolean* kennt zwei verschiedene Literale: `true` und `false`. Dieser Datentyp wird dort verwendet, wo eine Entscheidung erforderlich ist (z. B. bei Bedingungen in Fallunterscheidungen und Schleifen). Eine Umwandlung dieser Wahrheitswerte in ganzzahlige Werte ist nicht möglich.

## Zeichen

Der *Zeichentyp char* dient dazu, einzelne Zeichen des Unicode-Zeichensatzes in zwei Byte zu speichern. Literale werden in einfache Anführungszeichen eingeschlossen (z. B. `'a'`) und als Unicode-Zeichen oder als sogenannte *Escape-Sequenzen* angegeben (siehe Tabelle 2-2).

### Kurze Erläuterung zum Unicode:

Unicode ist eine standardisierte Mehrbyte-Codierung, mit der die Schriftzeichen aller gängigen Sprachen dargestellt werden können. Die Zeichen vom Typ `char` sind jeweils 16 Bit lang, was demnach 65.536 verschiedene Zeichen ermöglicht. Die ersten 128 Zeichen sind die üblichen 7-Bit-ASCII-Zeichen.

Java unterstützt auch neuere Unicode-Versionen, die vier Bytes für die Codierung eines Zeichens benötigen. Dafür sind dann zwei `char`-Elemente erforderlich.

**Tabelle 2-2:** Escape-Sequenzen

Escape-Sequenz	Bedeutung
\b	Backspace
\t	Tabulator
\n	neue Zeile (Line Feed, Newline)
\f	Seitenvorschub (Form Feed)
\r	Wagenrücklauf (Carriage Return)
\"	doppeltes Anführungszeichen "
\'	einfaches Anführungszeichen '
\\\	Backslash \
\ddd	ASCII-codiertes Zeichen in Oktal-Schreibweise
\udddd	Unicode-Zeichen in Hexadezimal-Schreibweise (z. B. \u0020 für das Leerzeichen)

Für die deutschen Umlaute und ß gelten die folgenden *Unicode-Escapes*:

Ä	\u00c4	ä	\u00e4	ß	\u00df
Ö	\u00d6	ö	\u00f6		
Ü	\u00dc	ü	\u00fc		

## Ganze Zahlen

Die *ganzzahligen Typen* byte, short, int, long sind vorzeichenbehaftet. Literale können in Dezimal-, Binär-, Oktal- oder Hexadezimalform notiert werden.

Binäre Literale werden mit `0b` oder `0B` eingeleitet. Ein *oktaler Wert* beginnt mit dem Präfix `0`, ein *hexadezimaler Wert* mit dem Präfix `0x` oder `0X`.

Gültige Ziffern sind bei dezimalen Literalen 0 bis 9, bei binären Literalen 0 und 1, bei oktalnen Literalen 0 bis 7 und bei hexadezimalen Literalen 0 bis 9, a bis f und A bis F.

Negative Zahlen werden durch Voranstellen des Minuszeichens - dargestellt. Ganzzahlige Literale sind vom Typ `int`, wenn nicht der Buchstabe l oder L angehängt ist. Im letzten Fall sind sie vom Typ `long`.

Der Typ `char` stellt auch einen ganzzahligen Typ dar und ist `short` gleichgestellt.

Wenn bei Berechnungen der zulässige Zahlenbereich überschritten wird, kommt es zu falschen Ergebnissen. Es findet also keine Überlaufkontrolle statt.

## Fließkommazahlen / Gleitkommazahlen

Literale der *Fließkommatypen* float und double werden in Dezimalschreibweise notiert. Sie bestehen aus einem Vorkomma-Teil, einem Dezimalpunkt, einem Nachkomma-Teil, einem Exponenten und einem Suffix.

Es muss mindestens der Dezimalpunkt, der Exponent oder das Suffix vorhanden sein, damit die Zahl von einer ganzzahligen Konstanten unterschieden werden kann.

Falls ein Dezimalpunkt vorkommt, muss vor oder nach ihm eine Ziffernfolge stehen. Entweder der Vorkomma-Teil oder der Nachkomma-Teil darf wegfallen. Dem Vorkomma-Teil und dem Exponenten kann ein Vorzeichen + oder - vorangestellt werden. Der Exponent, der durch e oder E eingeleitet wird, und das Suffix sind optional.

Das Suffix f oder F kennzeichnet ein `float`-Literal, das Suffix d oder D ein `double`-Literal. Falls kein Suffix angegeben ist, handelt es sich um ein `double`-Literal.

Folgende Literale stehen für dieselbe Fließkommazahl:

18. 1.8e1 .18e2

Unterstriche können in allen numerischen Literalen zur Erhöhung der Lesbarkeit eingefügt werden, z. B.

```
int x = 1_000_000;
```

Der Unterstrich darf nicht am Anfang und nicht am Ende des Literals stehen, ebenso nicht vor und nach einem Dezimalpunkt und nicht vor dem Suffix f, F, l oder L.

Variablen vom Typ float oder double können gebrochene Zahlen meist nur näherungsweise speichern.

Bei Überlauf lautet das Ergebnis `Infinity` bzw. `-Infinity`.

## Zeichenketten

*Zeichenketten* als Literale erscheinen in doppelten Anführungszeichen.

Beispiel:

```
"Das ist eine Zeichenkette"
```

Einen einfachen Datentyp für Zeichenketten gibt es allerdings nicht. Zeichenketten sind Objekte der Klasse `String`. Genaueres hierzu finden Sie im Kapitel 14. Mit dem Operator + können Zeichenketten aneinandergehängt werden.

## 2.2 Variablen

Die Definition einer *Variablen* erfolgt in der Form

```
Typname Variablenname;
```

Hierdurch wird Speicherplatz eingerichtet. Der Variablen kann durch eine explizite *Initialisierung* ein Wert zugewiesen werden.

Beispiel:

```
int nummer = 10;
```

Mehrere Variablen des gleichen Typs können in einer Liste, in der die Variablennamen dem Datentyp folgen und durch Kommas getrennt sind, definiert werden.

Beispiel:

```
int alter, groesse, nummer;
```

```
public class Variablen {
    public static void main(String[] args) {
        boolean booleanVar = true;
        char charVar = 'a';
        byte byteVar = 100;
        short shortVar = 32000;
        int intVar = 0b1000_0000;
        long longVar = 123_456_789;
        float floatVar = 0.12345f;
        double doubleVar = 0.12345e1;
```

```

        System.out.println("booleanVar: " + booleanVar);
        System.out.println("charVar: " + charVar);
        System.out.println("byteVar: " + byteVar);
        System.out.println("shortVar: " + shortVar);
        System.out.println("intVar: " + intVar);
        System.out.println("longVar: " + longVar);
        System.out.println("floatVar: " + floatVar);
        System.out.println("doubleVar: " + doubleVar);
    }
}

```

Ausgabe des Programms:

```

booleanVar: true
charVar: a
byteVar: 100
shortVar: 32000
intVar: 128
longVar: 123456789
floatVar: 0.12345
doubleVar: 1.2345

```

Im oben aufgeführten Programm werden Variablen der verschiedenen einfachen Datentypen definiert und initialisiert. Hier ist der Wert der `int`-Variablen in Binärform angegeben: Die Dualzahl 10000000 entspricht der Dezimalzahl 128.

Zur Ausgabe wird eine Zeichenkette mit dem Wert einer Variablen durch `+` verknüpft und das Ergebnis auf dem Bildschirm ausgegeben. Dabei wird vorher der Variablenwert automatisch in eine Zeichenkette umgewandelt.

### Typinferenz für lokale Variablen

Innerhalb von Methoden oder Blöcken kann der Datentyp bei der Definition einer Variablen auf der linken Seite auch weggelassen werden. Statt der Angabe des Typs wird einfach `var` hingeschrieben. `var` dient hier als Platzhalter für den Datentyp.

Beispiel:

```
var age = 42;
```

Der Compiler leitet den Typ vom Initialisierungsausdruck auf der rechten Seite ab. Im Beispiel handelt es sich um ein `int`-Literal. Also ist die Variable `age` vom Typ `int`. Mit der Verwendung von `var` kann der Quellcode deutlich kürzer und auch lesbarer sein.

```

public class VarTest {
    public static void main(String[] args) {
        var age = 42;
        var pi = 3.14159;
        var hello = "Hallo!";
        var notTrue = false;

        System.out.println(age + " " + pi + " " + hello + " " + notTrue);
    }
}

```

Die Variable `age` ist vom Typ `int`, `pi` vom Typ `double`, `hello` vom Typ `String` und `notTrue` vom Typ `boolean`.

## 2.3 Aufgaben

### Aufgabe 1

Welche der folgenden Bezeichner sind ungültig? Schauen Sie sich hierzu die elementaren Regeln in Kapitel 1 an.

<code>Hallo_Welt</code>	<code>\$Test</code>	<code>_abc</code>	<code>2test</code>
<code>#hallo</code>	<code>te?st</code>	<code>Girokonto</code>	<code>const</code>

*Lösung: Bezeichner*

### Aufgabe 2

Warum führt der folgende Code bei der Übersetzung zu einem Fehler?

```
int x = 0;
long y = 1000;
x = y;
```

*Lösung: Fehler*

### Aufgabe 3

Schreiben Sie ein Programm, das mit einem einzigen Aufruf der Java-Anweisung `System.out.print` eine Zeichenkette in mehreren Zeilen auf dem Bildschirm ausgibt.

*Lösung: Zeilen*

### Aufgabe 4

Speichern Sie die hexadezimale Zahl `AA00` und die binäre Zahl `10101010` in zwei `int`-Variablen und geben Sie die Werte aus.

*Lösung: Zahlen*

### Aufgabe 5

Codieren Sie die Lösung zu Aufgabe 4 mit Hilfe des Typ-Platzhalters `var`.

*Lösung: Var*



## 3 Operatoren

Mit *Operatoren* können Zuweisungen und Berechnungen vorgenommen und Bedingungen formuliert und geprüft werden. Operatoren sind Bestandteile von Ausdrücken.

### Ausdruck

Ein *Ausdruck* ist alles, was zu einem Wert ausgewertet werden kann. Er besteht im Allgemeinen aus Operatoren, Operanden, auf die ein Operator angewandt wird, und Klammern, die zusammen eine Auswertungsvorschrift beschreiben. Operanden können Variablen und Literale (konkrete Werte), aber auch Methodenaufrufe sein.

Ein Ausdruck wird zur Laufzeit ausgewertet und liefert dann einen Ergebniswert, dessen Typ sich aus den Typen der Operanden und der Art des Operators bestimmt. Einzige Ausnahme ist der Aufruf einer Methode mit Rückgabetyp `void`, dieser Ausdruck hat keinen Wert. Vorrangregeln legen die Reihenfolge der Auswertung fest, wenn mehrere Operatoren im Ausdruck vorkommen.

In den folgenden Tabellen, die die verschiedenen Operatoren aufführen, wird die Rangfolge durch Zahlen notiert. Priorität 1 kennzeichnet den höchsten Rang.

Durch Setzen von runden Klammern lässt sich eine bestimmte Auswertungsreihenfolge erzwingen:

`2 + 3 * 4` hat den Wert 14

`(2 + 3) * 4` hat den Wert 20

Literale, Variablen, Methodenaufrufe, Zugriffe auf Elemente eines Arrays u. Ä. bilden jeweils für sich elementare Ausdrücke.

Bei den Operatoren unterscheidet man arithmetische, relationale, logische, Bit-, Zuweisungs- und sonstige Operatoren.

### Lernziele

In diesem Kapitel lernen Sie

- mit welchen Operatoren Berechnungen, Vergleiche und Bedingungen formuliert werden können und
- welche Besonderheiten hierbei zu berücksichtigen sind.

## 3.1 Arithmetische Operatoren

Die *arithmetischen Operatoren* haben numerische Operanden und liefern einen numerischen Wert.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_3](https://doi.org/10.1007/978-3-658-43574-5_3).

Haben die Operanden unterschiedliche Datentypen, so wird automatisch eine *Typumwandlung* "nach oben" durchgeführt, d. h.:

Der "kleinere" Typ der beiden Operanden wird in den Typ des "größeren" umgewandelt (z. B. Umwandlung von `short` in `int`, von `int` in `double`).

Der Ergebnistyp entspricht dem größeren der beiden Operanden, ist aber mindestens vom Typ `int`.

Das bedeutet, dass z. B. die Summe zweier `byte`-Werte `b1` und `b2` vom Typ `int` ist und demnach nicht einer `byte`-Variablen zugewiesen werden kann.

Man beachte, dass bei Division von ganzen Zahlen der Nachkommateil abgeschnitten wird:

`13 / 5` hat den Wert 2

Hingegen hat `13 / 5.` den Wert 2.6, da 5. ein `double`-Literal ist und somit 13 nach `double` konvertiert wird. Man beachte den Dezimalpunkt hinter der Ziffer 5.

**Tabelle 3-1:** Arithmetische Operatoren

Operator	Bezeichnung	Priorität
+	positives Vorzeichen	1
-	negatives Vorzeichen	1
++	Inkrementierung	1
--	Dekrementierung	1
*	Multiplikation	2
/	Division	2
%	Rest	2
+	Addition	3
-	Subtraktion	3

Beim einstelligen *Inkrementierungs-* und *Dekrementierungsoperator*, der sich nur auf Variablen anwenden lässt, wird zwischen *Präfix-* und *Postfixform* unterschieden, je nachdem, ob der Operator vor oder hinter dem Operanden steht:

- `++a` hat den Wert von `a + 1`, `a` wird um 1 erhöht,
- `--a` hat den Wert von `a - 1`, `a` wird um 1 verringert,
- `a++` hat den Wert von `a`, `a` wird um 1 erhöht,
- `a--` hat den Wert von `a`, `a` wird um 1 verringert.

Bei der Präfixform liefert der Ausdruck den Wert nach der Veränderung, bei der Postfixform den Wert vor der Veränderung.

Der *Rest-Operator* `%` berechnet bei ganzzahligen Operanden den Rest  $r = a \% b$  einer ganzzahligen Division von  $a$  durch  $b$  so, dass gilt:  $a = (a / b) * b + r$ .

Beispiel:  $13 \% 5$  hat den Wert  $3$ ,  $-13 \% 5$  hat den Wert  $-3$ .

Der Rest-Operator kann auch auf Fließkommazahlen angewandt werden.

Beispiel:  $12. \% 2.5$  hat den Wert  $2.0$ , denn  $12. = 4 * 2.5 + 2.0$ .

```
public class ArithmOp {  
    public static void main(String[] args) {  
        System.out.println(13 / 5);  
        System.out.println(13 % 5);  
        System.out.println();  
  
        System.out.println(12. / 2.5);  
        System.out.println(12. % 2.5);  
        System.out.println();  
  
        int a = 1;  
        System.out.println(++a + "\t" + a);  
        System.out.println(a++ + "\t" + a);  
        System.out.println();  
  
        int b = 2;  
        System.out.println(--b + "\t" + b);  
        System.out.println(b-- + "\t" + b);  
        System.out.println();  
  
        double x = 0.7 + 0.1;  
        double y = 0.9 - 0.1;  
        System.out.println(y - x);  
    }  
}
```

Ausgabe des Programms:

2  
3

4.8  
2.0

2        2  
2        3

1        1  
1        0

1.1102230246251565E-16

Das Programm zeigt auch, dass das Rechnen mit Fließkommazahlen sogar in sehr einfachen Fällen ungenau sein kann, wenn auch die Ungenauigkeit verschwindend gering ist.

## 3.2 Relationale Operatoren

*Relationale Operatoren* vergleichen Ausdrücke mit numerischem Wert miteinander. Das Ergebnis ist vom Typ `boolean`.

Bei Fließkomma-Werten sollte die Prüfung auf exakte Gleichheit oder Ungleichheit vermieden werden, da es bei Berechnungen zu Rundungsfehlern kommen kann und die erwartete Gleichheit oder Ungleichheit daher nicht zutrifft. Stattdessen sollte mit den Operatoren `<` und `>` gearbeitet werden, um die Übereinstimmung der Werte bis auf einen relativen Fehler zu prüfen.

**Tabelle 3-2:** Relationale Operatoren

Operator	Bezeichnung	Priorität
<code>&lt;</code>	kleiner	5
<code>&lt;=</code>	kleiner oder gleich	5
<code>&gt;</code>	größer	5
<code>&gt;=</code>	größer oder gleich	5
<code>==</code>	gleich	6
<code>!=</code>	ungleich	6

## 3.3 Logische Operatoren

*Logische Operatoren* verknüpfen Wahrheitswerte miteinander. Java stellt die Operationen UND, ODER, NICHT und das exklusive ODER zur Verfügung.

**Tabelle 3-3:** Logische Operatoren

Operator	Bezeichnung	Priorität
<code>!</code>	NICHT	1
<code>&amp;</code>	UND mit vollständiger Auswertung	7
<code>^</code>	exklusives ODER (XOR)	8
<code> </code>	ODER mit vollständiger Auswertung	9
<code>&amp;&amp;</code>	UND mit kurzer Auswertung	10
<code>  </code>	ODER mit kurzer Auswertung	11

UND und ODER gibt es in zwei Varianten. Bei der sogenannten kurzen Variante (*short circuit*) wird der zweite Operand nicht mehr ausgewertet, wenn das Ergebnis des Gesamtausdrucks schon feststeht.

Beispielsweise ist  $A \text{ UND } B$  falsch, wenn  $A$  falsch ist, unabhängig vom Wahrheitswert von  $B$ .

Soll der zweite Operand auf jeden Fall ausgewertet werden, weil er z. B. eine unbedingt auszuführende Inkrementierung (als Seiteneffekt) enthält, muss die *vollständige Auswertung* mit `&` bzw. `|` genutzt werden.

`!a` ergibt `false`, wenn `a` den Wert `true` hat, und `true`, wenn `a` den Wert `false` hat.

Die folgende Tabelle zeigt die Ergebnisse der übrigen Operationen:

**Tabelle 3-4:** Verknüpfung von Wahrheitswerten

a	b	a & b a && b	a ^ b	a   b a    b
true	true	true	false	true
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

```
public class LogOp {
    public static void main(String[] args) {
        int a = 2, b = 3;

        System.out.println(a == 2 && b < 8);
        System.out.println(a != 2 || !(b < 2));
        System.out.println(a == 2 ^ b > 0);

        System.out.println(a == 0 && b++ == 3);
        System.out.println(b);

        System.out.println(a == 0 & b++ == 3);
        System.out.println(b);

        System.out.println(a == 2 || b++ == 3);
        System.out.println(b);

        System.out.println(a == 2 | b++ == 3);
        System.out.println(b);
    }
}
```

Ausgabe des Programms:

```
true
true
false
false
3
false
4
```

```
true
4
true
5
```

## 3.4 Bitoperatoren

*Bitoperatoren* arbeiten auf der Binärdarstellung ganzzahliger Operanden, also mit 8 (byte), 16 (short, char), 32 (int) oder 64 Bit (long).<sup>1</sup>

$\sim a$  entsteht aus  $a$ , indem alle Bit von  $a$  invertiert werden, d. h. 0 geht in 1 und 1 in 0 über.

Bei den *Schiebeoperatoren* werden alle Bit des ersten Operanden um so viele Stellen nach links bzw. rechts geschoben, wie im zweiten Operanden angegeben ist.

Beim *Linksschieben* werden von rechts Nullen nachgezogen. Beim *Rechtsschieben* werden von links Nullen nachgezogen, falls der erste Operand positiv ist. Ist er negativ, werden Einsen nachgezogen. Beim Operator  $>>>$  werden von links immer Nullen nachgezogen.

Beispiele:

$8 \ll 2$  hat den Wert 32

$8 >> 2$  hat den Wert 2

**Tabelle 3-5:** Bitoperatoren

Operator	Bezeichnung	Priorität
$\sim$	Bitkomplement	1
$\ll$	Linksschieben	4
$>>$	Rechtsschieben	4
$>>>$	Rechtsschieben mit Nachziehen von Nullen	4
$\&$	bitweises UND	7
$^$	bitweises exklusives ODER	8
$ $	bitweises ODER	9

Die folgende Tabelle zeigt die Ergebnisse der Verknüpfungen der jeweils korrespondierenden einzelnen Bit der beiden Operanden:

---

1 Informationen zum Binärsystem findet man unter  
<http://de.wikipedia.org/wiki/Dualsystem>

**Tabelle 3-6:** Bitweise Verknüpfungen

a	b	a & b	a ^ b	a   b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

## 3.5 Zuweisungsoperatoren

Neben der einfachen *Zuweisung* können arithmetische und bitweise Operatoren mit der Zuweisung kombiniert werden.

Bei der einfachen Zuweisung = wird der rechts stehende Ausdruck ausgewertet und der links stehenden Variablen zugewiesen.

Dabei müssen die Datentypen beider Seiten *kompatibel* sein, d. h. der Typ des Ausdrucks muss mit dem Typ der Variablen übereinstimmen oder in diesen umgewandelt werden können. Eine automatische Umwandlung der rechten Seite "nach oben" in den Typ der Variablen wird ggf. durchgeführt.

Folgende Zuweisungen (in Pfeilrichtung) sind möglich:

byte --> short, char --> int --> long --> float --> double

Zuweisungen haben als Ausdruck selbst einen Wert, nämlich den Wert des zugewiesenen Ausdrucks.

**Tabelle 3-7:** Zuweisungsoperatoren

Operator	Bezeichnung	Priorität
=	einfache Zuweisung	13
op=	kombinierte Zuweisung, dabei steht op für *, /, %, +, -, <<, >>, >>>, &, ^,	13

a op= b entspricht a = a op b, wobei für op der anzuwendende Operator einzusetzen ist, z. B. a += b.

Auch *Mehrfachzuweisungen* sind möglich, z. B.

a = b = 2

Das folgende Programm demonstriert die Wirkung der Bitoperatoren `&`, `|` und `^`. Das Bitmuster des zweiten Operanden `y` ist jeweils so gewählt, dass die Bits des ersten Operanden `x` in bestimmter Weise manipuliert werden:

Löschen von Bits (Bit auf `0` setzen), Setzen von Bits (Bit auf `1` setzen), Umschalten von Bits (aus `0` wird `1` und umgekehrt).

Die Kommentare im Quellcode zeigen die Bits des Ergebnisses. Ausgegeben werden aber die numerischen Werte des Ergebnisses.

```
public class BitOp {
    public static void main(String[] args) {
        int x, y;

        x = 0b10101010;
        y = 0b11110000;
        //      10100000
        x &= y;
        System.out.println(x);

        x = 0b10101010;
        y = 0b00001111;
        //      10101111
        x |= y;
        System.out.println(x);

        x = 0b10101010;
        y = 0b00001111;
        //      10100101
        x ^= y;
        System.out.println(x);
    }
}
```

Ausgabe des Programms:

160  
175  
165

## 3.6 Bedingungsoperator

Der Bedingungsoperator (*ternärer Operator*) benötigt drei Operanden:

*Bedingung* ? *Ausdruck1* : *Ausdruck2*

*Bedingung* muss vom Typ *boolean* sein. Falls *Bedingung* wahr ist, wird *Ausdruck1*, sonst *Ausdruck2* ausgewertet. Der Bedingungsoperator hat die Priorität 12.

Beispiele:

$a < 0 ? -a : a$

hat den Wert des Absolutbetrags von  $a$ , wenn  $a$  eine Zahl ist.

$a < b ? b : a$

ergibt das Maximum von  $a$  und  $b$ .

```
public class BedingungsOp {
    public static void main(String[] args) {
        int y = 1;
        int x = (y == 0) ? 0 : 100;
        System.out.println(x);
    }
}
```

Ausgabe des Programms:

100

## 3.7 Cast-Operator

Mit dem *Cast-Operator* wird eine explizite *Typumwandlung* vorgenommen.

Der Ausdruck

$(type) a$

wandelt den Ausdruck  $a$  in einen Ausdruck vom Typ  $type$  um, sofern eine solche Umwandlung möglich ist.

Der Cast-Operator darf nur auf der rechten Seite einer Zuweisung auftauchen. Er hat die Priorität 1.

Beispiele:

Um einer `int`-Variablen  $b$  einen `double`-Wert  $a$  zuzuweisen, ist folgende Typumwandlung erforderlich:

`b = (int) a;`

Hierbei wird der Nachkommateil ohne Rundung abgeschnitten.

Bei der Umwandlung einer `short`-Variablen  $a$  in einen `byte`-Wert  $b$  durch

`b = (byte) a;`

wird das höherwertige Byte von  $a$  abgeschnitten. Hat z. B.  $a$  den Wert 257, so hat  $b$  nach der Zuweisung den Wert 1, denn 257 ist `100000001` als Dualzahl.

Das folgende Beispielprogramm zeigt, wie eine *reelle Division* ganzer Zahlen erzwungen werden kann und wie durch Zuweisung eines `long`-Wertes an eine `double`-Variable Genauigkeit verloren gehen kann.

```
public class CastOp {
    public static void main(String[] args) {
        int x = 5, y = 3;
```

```
double z = x / y;
System.out.println(z);

z = (double) x / y;
System.out.println(z);

long a = 9123456789123456789L;
System.out.println(a);

double b = a;
long c = (long) b;
System.out.println(c);
}

}
```

Ausgabe des Programms:

```
1.0
1.6666666666666667
9123456789123456789
9123456789123457024
```

## 3.8 Aufgaben

### Aufgabe 1

Es sollen  $x$  Flaschen in Kartons verpackt werden. Ein Karton kann  $n$  Flaschen aufnehmen. Schreiben Sie ein Programm, das ermittelt, in wie viele Kartons eine bestimmte Anzahl Flaschen verpackt werden kann und wie viele Flaschen übrig bleiben.

*Lösung: Verpackung*

### Aufgabe 2

Zu vorgegebenen Zahlen  $x$  und  $y$  soll festgestellt werden, ob  $x$  durch  $y$  teilbar ist. Schreiben Sie hierzu ein Programm.

*Lösung: Teilen*

### Aufgabe 3

Jetzt ist es  $x$  Uhr (volle Stundenzahl). Wie viel Uhr ist es in  $n$  Stunden? Schreiben Sie hierzu ein Programm.

*Lösung: Uhrzeit*

### Aufgabe 4

Schreiben Sie ein Programm, das die Anzahl von Sekunden im Monat Januar berechnet.

*Lösung: Sekunden*

**Aufgabe 5**

Welche Werte haben die folgenden Ausdrücke und welche Werte haben die Variablen nach der Auswertung, wenn a den Anfangswert 1 und b den Anfangswert 7 hat?

- a)  $--a$       b)  $a--$       c)  $a++ + b$       d)  $b = ++a$   
e)  $a = b++$     f)  $-(a--) - -(--b)$     g)  $a++ + ++a + a++$

*Lösung: InkrementDekrement*

**Aufgabe 6**

Schreiben Sie ein Programm, das auf Basis eines vorgegebenen Radius den Durchmesser, den Umfang und die Fläche eines Kreises berechnet. Die Zahl  $\pi$  soll den angenäherten Wert 3.14159 haben.

*Lösung: Kreis*



## 4 Kontrollstrukturen

Anweisungen stellen die kleinsten ausführbaren Einheiten eines Programms dar.

Eine Anweisung kann

- eine Definition z. B. von Variablen enthalten,
- einen Ausdruck (Zuweisung, Inkrementierung, Dekrementierung, Methodenaufruf, Erzeugung eines Objekts) auswerten oder
- den Ablauf des Programms steuern.

Das Semikolon ; markiert das Ende einer Anweisung. Die *leere Anweisung* ; wird dort benutzt, wo syntaktisch eine Anweisung erforderlich ist, aber von der Programmlogik her nichts zu tun ist.

### Lernziele

In diesem Kapitel lernen Sie

- wie mit Hilfe von Kontrollstrukturen (das sind Verzweigungen und Schleifen) der Ablauf eines Programms gesteuert werden kann,
- wie wichtig die richtige Setzung geschweifter Klammern ist und
- wie nützlich Einrückungen des Codes sind, um Kontrollstrukturen übersichtlich zu gestalten.

### Block { ... }

Die geschweiften Klammern { und } fassen mehrere Anweisungen zu einem *Block* zusammen. Dieser kann auch keine oder nur eine Anweisung enthalten. Ein Block gilt wiederum als eine Anweisung und kann überall dort verwendet werden, wo eine elementare Anweisung erlaubt ist. Damit können Blöcke ineinander geschachtelt werden.

Variablen, die in einem Block definiert werden, sind nur dort gültig und sichtbar.

## 4.1 Verzweigungen

*Verzweigungen* erlauben es, abhängig von Bedingungen unterschiedliche Anweisungen auszuführen.

### if ... else

Die *if-Anweisung* tritt in zwei Varianten auf:

```
if (Ausdruck)
    Anweisung
```

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_4](https://doi.org/10.1007/978-3-658-43574-5_4).

oder

```
if (Ausdruck)
    Anweisung1
else
    Anweisung2
```

*Ausdruck* hat den Typ boolean.

Im ersten Fall wird *Anweisung* nur ausgeführt, wenn *Ausdruck* den Wert true hat. Im zweiten Fall wird *Anweisung1* ausgeführt, wenn *Ausdruck* den Wert true hat, andernfalls wird *Anweisung2* ausgeführt.

Der auszuführende Code kann aus einer einzelnen Anweisung oder aus einem Anweisungsblock bestehen. if-Anweisungen können geschachtelt werden. Ein else wird dem nächstmöglichen if zugeordnet. Das zeigt das folgende Beispielprogramm:

```
public class IfTest {
    public static void main(String[] args) {
        int zahl = 4;

        if (zahl == 6 || zahl == 8)
            System.out.println("Knapp daneben");
        else if (zahl == 7)
            System.out.println("Treffer");
        else
            System.out.println("Weit daneben");
    }
}
```

Ausgabe des Programms:

Weit daneben

## switch

Die switch-Anweisung führt je nach Wert des Ausdrucks unterschiedliche Anweisungen aus.

```
switch (Ausdruck) {
    case Konstante:
        Anweisungen
    ...
    default:
        Anweisungen
}
```

Der Typ des Ausdrucks *Ausdruck* muss char, byte, short, int, Character, Byte, Short, Integer, String oder ein Aufzählungstyp (enum) sein.<sup>1</sup>

---

<sup>1</sup> Character, Byte, Short, Integer und Aufzählungstypen werden in späteren Kapiteln behandelt (Kapitel 12 und 15).

*Konstante* muss ein konstanter Ausdruck passend zum Typ von *Ausdruck* sein. Diese konstanten Ausdrücke müssen paarweise verschiedene Werte haben.

In Abhängigkeit vom Wert von *Ausdruck* wird die Sprungmarke (case) angesprungen, deren Konstante mit dem Wert des Ausdrucks übereinstimmt. Dann werden *alle* dahinter stehenden Anweisungen, auch solche, die andere Sprungmarken haben, bis zum Ende der switch-Anweisung oder bis zum ersten break ausgeführt. Dieses Verhalten wird als *Durchfallen (Fall Through)* bezeichnet.

Die Anweisung break führt zum sofortigen Verlassen der switch-Anweisung. Die optionale Marke default wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird.

Anweisungen steht jeweils für keine, eine oder mehrere einzelne Anweisungen.

Das folgende Programm ist die switch-Variante zum obigen Programm mit if ... else:

```
public class SwitchTest {  
    public static void main(String[] args) {  
        int zahl = 4;  
  
        switch (zahl) {  
            case 6:  
            case 8:  
                System.out.println("Knapp daneben");  
                break;  
            case 7:  
                System.out.println("Treffer");  
                break;  
            default:  
                System.out.println("Weit daneben");  
        }  
    }  
}
```

Mit der Java-Version 14 wurde die switch-Syntax erweitert.

Es können nun mehrere durch Kommas getrennte Werte hinter case angegeben werden und switch kann auch als Ausdruck mit Ergebnis verwendet werden.

Das folgende Programm zeigt vier verschiedene Formen von switch.

Ist der Operator '\*' oder 'x', soll "Multiplikation" ausgegeben oder als Ergebnis zurückgegeben werden. Ist der Operator ':' oder '/', soll "Division" geliefert werden.

Die vier Formen werden im Anschluss erläutert.

```
public class SwitchTest2 {  
    public static void main(String[] args) {  
        char op = 'x';
```

```

// Beispiel 1
switch (op) {
    case '*':
    case 'x':
        System.out.println("Multiplikation");
        break;
    case ':':
    case '/':
        System.out.println("Division");
}
}

// Beispiel 2
switch (op) {
    case '*', 'x' -> System.out.println("Multiplikation");
    case ':', '/' -> {
        String s = "Division";
        System.out.println(s);
    }
}

// Beispiel 3
String result1 = switch (op) {
    case '*':
    case 'x':
        yield "Multiplikation";
    case ':', '/':
        String s = "Division";
        yield s;
    default:
        yield "Unbekannt";
};
System.out.println(result1);

// Beispiel 4
String result2 = switch (op) {
    case '*', 'x' -> "Multiplikation";
    case ':', '/' -> {
        String s = "Division";
        yield s;
    }
    default -> "Unbekannt";
};
System.out.println(result2);
}
}

```

### *Erläuterungen zum Programm*

#### *Beispiel 1*

Hier handelt es sich um die "klassische" `switch`-Form, wie sie oben erläutert wurde. Durchfallen (Fall Through) ist möglich, eine vollständige Abdeckung der Fälle ist nicht nötig (`default` darf fehlen).

### *Beispiel 2*

Hinter dem Pfeil -> steht entweder eine Anweisung, ein Block in geschweiften Klammern oder ein `throw`-Anweisung, die eine nicht kontrollierte Ausnahme auslöst (siehe Kapitel 13). Ein Durchfallen ist *nicht* möglich, eine vollständige Abdeckung der Fälle ist nicht erforderlich.

In den Beispielen 3 und 4 wird `switch` als Ausdruck mit Ergebnis verwendet. Hier ist eine vollständige Abdeckung der Fälle (`case`) erforderlich, da der `switch`-Ausdruck immer ein Ergebnis haben muss.

### *Beispiel 3*

`yield` zur Rückgabe oder eine nicht kontrollierte Ausnahme ist erforderlich. Ein Durchfallen ist möglich.

### *Beispiel 4*

Hinter dem Pfeil -> steht entweder ein Ausdruck, ein Block in geschweiften Klammern oder eine `throw`-Anweisung, die eine nicht kontrollierte Ausnahme auslöst (siehe Kapitel 13). In einem Block wird das Ergebnis mit `yield` zurückgegeben. Ein Durchfallen ist *nicht* möglich.

## Kurze Zusammenfassung

Wird : bei `case` genutzt, ist ein Durchfallen möglich, wird -> genutzt, ist Durchfallen nicht möglich.

Wird `switch` als Anweisung verwendet, ist eine Abdeckung aller Fälle nicht erforderlich, wird `switch` als Ausdruck mit Ergebnis genutzt, müssen alle Möglichkeiten abgedeckt sein.

## 4.2 Schleifen

*Schleifen* führen Anweisungen wiederholt aus, solange eine bestimmte Bedingung erfüllt ist.

### **while**

Die `while`-*Schleife* ist eine *abweisende* Schleife, d. h. die Ausführungsbedingung wird jeweils vor Eintritt in die Schleife überprüft.

`while (Ausdruck)`  
`Anweisung`

*Ausdruck* muss vom Typ boolean sein. Hat *Ausdruck* den Wert `true`, wird *Anweisung* (eine einzelne Anweisung oder ein Anweisungsblock) ausgeführt, andernfalls wird mit der Anweisung fortgefahren, die der Schleife folgt.

Wenn die Schleife jemals abbrechen soll, muss die Bedingung im Anweisungsblock beeinflusst werden und Ausdruck dann mit `false` ausgewertet werden.

### do ... while

Die **do-Schleife** ist eine *nicht abweisende* Schleife, d. h. *Anweisung* wird mindestens einmal ausgeführt. Die Ausführungsbedingung wird erst nach der Ausführung der Anweisung (eine einzelne Anweisung oder ein Anweisungsblock) geprüft.

```
do
    Anweisung
    while (Ausdruck);
```

Die Schleife wird beendet, wenn *Ausdruck*, der vom Typ `boolean` sein muss, den Wert `false` hat.

Das folgende Programm addiert die Zahlen von 1 bis zu einer vorgegebenen Zahl:

```
public class WhileTest {
    public static void main(String[] args) {
        int n = 100, summe = 0, i = 1;
        while (i <= n) {
            summe += i;
            i++;
        }
        System.out.println("Summe 1 bis " + n + ": " + summe);
    }
}
```

Ausgabe des Programms:

Summe 1 bis 100: 5050

### for

Die **for-Schleife** wiederholt eine Anweisung in Abhängigkeit von Kontrollausdrücken.

```
for (Init; Bedingung; Update)
    Anweisung
```

*Init* ist eine Liste von durch Kommas voneinander getrennten Anweisungen oder Variablendefinitionen des gleichen Typs. *Init* wird einmal vor dem Start der Schleife aufgerufen. Dieser Initialisierungsteil darf auch fehlen.

*Bedingung* ist ein Ausdruck vom Typ `boolean`. Die Bedingung wird zu Beginn jedes Schleifendurchgangs getestet. Fehlt *Bedingung*, wird als Ausdruck `true` angenommen.

*Anweisung* (eine einzelne Anweisung oder ein Anweisungsblock) wird nur ausgeführt, wenn *Bedingung* den Wert `true` hat.

*Update* ist eine Liste von durch Kommas voneinander getrennten Anweisungen. Sie kann auch leer sein. *Update* wird nach jedem Durchlauf der Schleife ausgewertet, bevor *Bedingung* das nächste Mal ausgewertet wird.

In den meisten Fällen dient *Update* dazu, den Schleifenzähler zu verändern und damit die Laufbedingung zu beeinflussen.

Der in Klammern gesetzte Teil hinter `for` muss immer genau zwei Semikolons enthalten.

Auch die folgende Schleife ist gültig, wenn auch eine Endlosschleife:

```
for (;;);
```

```
public class ForTest {  
    public static void main(String[] args) {  
        int n = 100, summe = 0;  
        for (int i = 1; i <= n; i++)  
            summe += i;  
        System.out.println("Summe 1 bis " + n + ": " + summe);  
    }  
}
```

### Genauigkeit beim Rechnen mit double-Zahlen

Das nächste Programm zeigt, dass beim Rechnen mit `double`-Werten Genauigkeit verloren gehen kann. Das Programm soll die folgende Aufgabe lösen:

Sie haben einen Euro und sehen ein Regal mit Bonbons, die 10 Cent, 20 Cent, 30 Cent usw. bis hinauf zu einem Euro kosten. Sie kaufen von jeder Sorte ein Bonbon, beginnend mit dem Bonbon für 10 Cent, bis Ihr Restgeld für ein weiteres Bonbon nicht mehr ausreicht.

Wie viele Bonbons kaufen Sie und welchen Geldbetrag erhalten Sie zurück? Natürlich 4 Bonbons und kein Restgeld.

Aber ...

```
public class Bonbons1 {  
    public static void main(String[] args) {  
        double budget = 1.;  
        int anzahl = 0;  
  
        for (double preis = 0.1; budget >= preis; preis += 0.1) {  
            budget -= preis;  
            anzahl++;  
        }  
  
        System.out.println(anzahl + " Bonbons gekauft.");  
        System.out.println("Restgeld: " + budget);  
    }  
}
```

Ausgabe des Programms:

```
3 Bonbons gekauft.
Restgeld: 0.3999999999999999
```

Vergleiche die Ausführungen zu Fließkomma-/Gleitkommazahlen in Kapitel 2 und die dort erwähnte mögliche Ungenauigkeit.

Die richtige Lösung erhält man, indem man mit ganzen Zahlen rechnet:

```
public class Bonbons2 {
    public static void main(String[] args) {
        int budget = 100;
        int anzahl = 0;

        for (int preis = 10; budget >= preis; preis += 10) {
            budget -= preis;
            anzahl++;
        }

        System.out.println(anzahl + " Bonbons gekauft.");
        System.out.println("Restgeld: " + budget);
    }
}
```

Ausgabe des Programms:

```
4 Bonbons gekauft.
Restgeld: 0
```

## **foreach**

Es gibt eine Variante der `for`-Schleife: die sogenannte `foreach`-Schleife. Mit ihr können in einfacher Form Elemente eines Arrays oder einer Collection (siehe Kapitel 20) durchlaufen werden. Diese Schleifenform wird an gegebener Stelle später erläutert.

## **4.3 Sprunganweisungen**

*Sprunganweisungen* werden hauptsächlich verwendet, um Schleifendurchgänge vorzeitig zu beenden.

### **break**

Die Anweisung `break` beendet eine `switch`-, `while`-, `do`- oder `for`-Anweisung, die die `break`-Anweisung unmittelbar umgibt.

Um aus geschachtelten Schleifen herauszuspringen, gibt es eine Variante:

```
break Marke;
```

*Marke* steht für einen selbst gewählten Bezeichner. Diese Anweisung verzweigt an das Ende der Anweisung, vor der diese Marke unmittelbar steht. Eine *markierte Anweisung* hat die Form:

*Marke: Anweisung*

break-Anweisungen mit Marke dürfen sogar in beliebigen markierten Blöcken genutzt werden.

### continue

Die Anweisung `continue` unterbricht den aktuellen Schleifendurchgang einer `while`-, `do`- oder `for`-Schleife und springt an die Wiederholungsbedingung der sie unmittelbar umgebenden Schleife. Wie bei der `break`-Anweisung gibt es auch hier die Variante mit Marke:

```
continue Marke;
```

Ein Beispiel:

```
public class SprungTest {  
    public static void main(String[] args) {  
        M1:  
        for (int i = 1; i < 10; i++) {  
            for (int j = 1; j < 10; j++) {  
                System.out.print(j + " ");  
                if (j == i) {  
                    System.out.println();  
                    continue M1;  
                }  
            }  
        }  
    }  
}
```

Ausgabe des Programms:

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5  
1 2 3 4 5 6  
1 2 3 4 5 6 7  
1 2 3 4 5 6 7 8  
1 2 3 4 5 6 7 8 9
```

## 4.4 Die Java-Shell

Java bietet einen Kommandointerpreter (Java-Shell), mit dem interaktiv und schnell kleinere Quellcode-Schnipsel ausprobiert werden können. Dabei können Anweisungen auch ohne Klassen- und Methodendefinition ausgeführt werden.

Die Shell wird durch Eingabe von `jshell` im Terminal gestartet.

Hierzu ein kleines Beispiel:

```
$ jshell
| Willkommen bei JShell - Version 21
| Geben Sie für eine Einführung Folgendes ein: /help intro

jshell> int n = 3
n ==> 3

jshell> for (int i = 0; i < n; i++) {
...>     System.out.println(i);
...> }
0
1
2

jshell> /exit
| Auf Wiedersehen
```

In vielen Fällen müssen einfache Anweisungen hier nicht mit einem Semikolon enden.

Es gibt eine Reihe von besonderen Kommandos der Shell, die mit einem Schrägstrich beginnen, z. B.

`/exit` um die Shell zu beenden,  
`/help` listet alle Möglichkeiten auf.

Eine ausführliche Dokumentation ist unter

<https://docs.oracle.com/en/java/javase/21/jshell/>  
zu finden.

## 4.5 Aufgaben

### Aufgabe 1

Was wird hier ausgegeben? Warum nicht die Zahlen von 0 bis 9?

```
int i;
for (i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Verbessern sie das Programm.

*Lösung: Preisfrage*

### Aufgabe 2

Erzeugen Sie die Ausgabe des Programms in Kapitel 4.3 eleganter, d. h. ohne Verwendung von `continue`.

*Lösung: Dreieck*

### Aufgabe 3

Schreiben Sie ein Programm, das eine von Ihnen vorgegebene Anzahl von Sternchen (\*) in Form eines Dreiecks auf dem Bildschirm ausgibt.

*Lösung: Sternchen*

### Aufgabe 4

Schreiben Sie ein Programm, das eine Tabelle mit dem kleinen Einmaleins (also  $1 * 1$  bis  $10 * 10$ ) angeordnet in zehn Zeilen mit je zehn Spalten ausgibt.

*Lösung: Einmaleins*

### Aufgabe 5

Eine Schleife soll den Wertebereich zwischen 0 und 100 in 5 Schritten gleicher Länge durchlaufen. Alle 5 Zahlen sollen ausgegeben werden.

*Lösung: Schritte*

### Aufgabe 6

Schreiben Sie ein Programm, das ermittelt, wie hoch ein Guthaben von 5000 Geldeinheiten bei 1,5 % Verzinsung nach Ablauf eines Jahres ist.

*Lösung: Verzinsung*

### Aufgabe 7

Schreiben Sie ein Programm, das den Buchwert in Höhe von 15000 Geldeinheiten mit einem Abschreibungssatz von 40 % und einem Restwert von 100 Geldeinheiten geometrisch degressiv abschreibt.

*Lösung: Abschreibung*

### Aufgabe 8

Berechnen Sie den kleinsten ganzzahligen Wert von  $n$ , sodass  $2^n$  größer oder gleich einer vorgegebenen ganzen Zahl  $x$  ist.

Beispiel: Für  $x = 15$  ist  $n = 4$ .

*Lösung: Log*

### Aufgabe 9

Zwei Fließkommazahlen sollen verglichen werden. Ist der Absolutbetrag der Differenz dieser beiden Zahlen kleiner als ein vorgegebener Wert  $z$ , soll  $0$  ausgegeben werden, sonst  $-1$  bzw.  $1$ , je nachdem  $x$  kleiner als  $y$  oder größer als  $y$  ist.

*Lösung: Signum*

### Aufgabe 10

Schreiben Sie ein Programm, das eine ganze Zahl vom Typ `int` in Binärdarstellung (32 Bit) ausgibt. Benutzen Sie hierzu die Bitoperatoren `&` und `<<`.

Tipp: Das Bit mit der Nummer  $i$  (Nummerierung beginnt bei  $0$ ) in der Binärdarstellung von `zahl` hat den Wert `1` genau dann, wenn der Ausdruck

`zahl & (1 << i)`

von  $0$  verschieden ist.

*Lösung: Binaer*

### Aufgabe 11

Es soll der größte gemeinsame Teiler von zwei positiven ganzen Zahlen  $p$  und  $q$  mit Hilfe des *Euklidischen Algorithmus* ermittelt werden. Dieses Verfahren kann wie folgt beschrieben werden:

- (1) Belege  $p$  mit einer positiven ganzen Zahl.
- (2) Belege  $q$  mit einer positiven ganzen Zahl.
- (3) Ist  $p < q$ , dann weiter mit (4), sonst mit (5).
- (4) Vertausche die Belegung von  $p$  und  $q$ .
- (5) Ist  $q = 0$ , dann weiter mit (9), sonst mit (6).
- (6) Belege  $r$  mit dem Rest der Division  $p$  durch  $q$ .
- (7) Belege  $p$  mit dem Wert von  $q$ .
- (8) Belege  $q$  mit dem Wert von  $r$ , dann weiter mit (5).
- (9) Notiere die Belegung von  $p$  als Ergebnis und beende.

Schreiben Sie hierzu ein Programm.

*Lösung: Euklid*

### Aufgabe 12

Schreiben Sie ein Programm, das zu einer Zahl  $n$  die Fakultät  $n!$  ermittelt. Es gilt bekanntlich:

$$n! = 1 * 2 * \dots * (n-1) * n \quad \text{sowie} \quad 0! = 1$$

Die größte Fakultät, die in einer long-Variablen dargestellt werden kann, ist 20!. Für größere Fakultäten muss eine double-Variable verwendet werden.

## *Lösung: Fakultät*

## Aufgabe 13

Schreiben Sie ein Programm, das eine Treppe aus  $h$  Stufen einer bestimmten Breite  $b$  in der folgenden Form zeichnet (Ausgabe von Leerzeichen und "\*" im Terminal):

In diesem Beispiel ist  $h = 10$  und  $b = 3$ . Nutzen Sie Schleifen. Nur durch Änderung von  $h$  und  $b$  soll die Treppenform angepasst werden können.

## *Lösung: Treppe*

## Aufgabe 14

Schreiben Sie ein Programm, das die Zerlegung einer Zahl  $n \geq 2$  in ihre Primfaktoren berechnet. Nutzen Sie das folgende Verfahren:

## Setze n

Setze t auf 2

Solange  $n > 1$  ist:

Ist  $n$  durch  $t$  teilbar?

Wenn ja:

Teile  $n$  durch  $t$

Setze  $n$  auf dieses Ergebnis

## Gib t aus

Wenn nein:

Erhöhe  $t$  um 1

## Lösung: Primfaktoren

## Aufgabe 15

Ein Schaltjahr ist ein Jahr, das eine Jahreszahl hat, die durch 4 teilbar ist.

denn, die Jahreszahl ist durch 400 teilbar. Schreiben Sie ein Programm, das für ein vorgegebenes Jahr ermittelt, ob es ein Schaltjahr ist.

*Lösung: Schaltjahr*

### Aufgabe 16

Berechnen Sie die Quersumme einer vorgegebenen ganzen Zahl. Die Quersumme einer Zahl ist die Summe aller ihrer Ziffern.

*Lösung: Quersumme*

### Aufgabe 17

Schreiben Sie ein Programm, das den *Body-Mass-Index* (BMI) für die Bewertung des Körpergewichts eines Menschen (in kg) in Relation zu seiner Körpergröße (in m) berechnet:

`bmi = gewicht / (groesse * groesse)`

Geben Sie zusätzlich die folgenden Texte aus:

Untergewicht	(falls <code>bmi &lt; 18.5</code> ),
Normalgewicht	(falls <code>bmi &gt;= 18.5 und bmi &lt; 25</code> ),
Übergewicht	(falls <code>bmi &gt;= 25 und bmi &lt; 30</code> ),
Adipositas	(falls <code>bmi &gt;= 30</code> ).

*Lösung: Bmi*

### Aufgabe 18

Für einen vorgegebenen Monat (als `int`-Zahl) soll die jeweilige Jahreszeit ermittelt werden. Nutzen Sie `switch` als Ausdruck mit Ergebnis.

*Lösung: Jahreszeit*



# 5 Arrays

Ein *Array* ist eine geordnete Sammlung von Elementen desselben Datentyps, die man unter einem gemeinsamen Namen ansprechen kann. Die Elemente eines Arrays enthalten alle entweder Werte desselben einfachen Datentyps oder Referenzen auf Objekte desselben Typs.

In diesem Kapitel werden nur Arrays vom einfachen Datentyp und *String*-Arrays betrachtet.

## Lernziele

In diesem Kapitel lernen Sie

- wie Arrays definiert und initialisiert werden können und
- wie man mit Arrays arbeitet.

## 5.1 Definition und Initialisierung

Die Definition der Array-Variablen erfolgt in der Form

*Typ[] Arrayname;*

*Typ* ist hierbei ein einfacher Datentyp (oder ein Referenztyp).

Zur Erzeugung eines Arrays wird die *new*-Anweisung benutzt:

*new Typ[Ausdruck]*

*Ausdruck* legt die Größe des Arrays fest und muss einen ganzzahligen Wert vom Typ *int* haben.

Beispiel:

```
int[] x;  
x = new int[10];
```

Hierdurch wird ein Array *x* vom Typ *int* erzeugt, das 10 Zahlen aufnehmen kann.

Steht zum Zeitpunkt der Definition bereits fest, wie viele Elemente das Array aufnehmen soll, können beide Anweisungen auch zusammengefasst werden.

Beispiel:

```
int[] x = new int[10];
```

Die Elemente eines Arrays werden bei ihrer Erzeugung mit *Standardwerten* vorbesetzt. Standardwerte sind:

0 für Zahlen, *false* für den Typ *boolean*, das Unicode-Zeichen \u0000 für *char*, bei *String* der Wert *null* (das ist eine Referenz, die auf nichts verweist).

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_5](https://doi.org/10.1007/978-3-658-43574-5_5).

Ein Array kann bei der Definition erzeugt und direkt *initialisiert* werden. Die Größe des Arrays ergibt sich aus der Anzahl der zugewiesenen Werte.

Beispiel:

```
int[] x = {1, 10, 4, 0}; // hat vier Elemente
oder auch
int[] x = new int[] {1, 10, 4, 0};
```

Die Größe eines Arrays kann erst zur Laufzeit festgelegt werden. Sie kann dann aber nicht mehr verändert werden.

Wir werden später sehen, dass Arrays spezielle Objekte sind. Die Anzahl der Elemente eines Arrays kann über das Attribut `length` abgefragt werden. Für das Array `x` im letzten Beispiel gilt also: `x.length` hat den Wert 4.

## 5.2 Zugriff auf Array-Elemente

Die Elemente eines Arrays der Größe `n` werden von `0` bis `n-1` durchnummeriert. Der Zugriff auf ein Element erfolgt über seinen Index:

`Arrayname[Ausdruck]`

`Ausdruck` muss den Ergebnistyp `int` haben.

Die Einhaltung der Array-Grenzen wird vom Laufzeitsystem geprüft. Bei Überschreiten der Grenzen wird die Exception `ArrayIndexOutOfBoundsException` ausgelöst (siehe Kapitel 13).

```
public class ArrayTest1 {
    public static void main(String[] args) {
        int[] zahlen = new int[10];

        for (int i = 0; i < zahlen.length; i++) {
            zahlen[i] = i * 100;
        }

        for (int i = 0; i < zahlen.length; i++) {
            System.out.print(zahlen[i] + " ");
        }

        System.out.println();
        String[] tage = { "Mo", "Di", "Mi", "Do", "Fr", "Sa", "So" };
        for (int i = 0; i < tage.length; i++) {
            System.out.print(tage[i] + " ");
        }
    }
}
```

Ausgabe des Programms:

```
0 100 200 300 400 500 600 700 800 900
Mo Di Mi Do Fr Sa So
```

Wenn beim Durchlaufen einer `for`-Schleife der Schleifenindex nicht benötigt wird, kann die `foreach`-Schleife eingesetzt werden:

```
for (int zahl : zahlen) {  
    System.out.print(zahl + " ");  
}
```

In diesem Beispiel wird der Variablen `zahl` der Reihe nach jedes Element des Arrays `zahlen` zugewiesen. Die Variable `zahl` ist nur im Schleifenkörper gültig. Veränderungen an der Schleifenvariablen wirken sich allerdings nicht auf die Elemente des Arrays aus.

```
public class ArrayTest2 {  
    public static void main(String[] args) {  
        int[] zahlen = new int[10];  
  
        for (int i = 0; i < zahlen.length; i++) {  
            zahlen[i] = i * 100;  
        }  
  
        for (int zahl : zahlen) {  
            System.out.print(zahl + " ");  
        }  
  
        System.out.println();  
        String[] tage = { "Mo", "Di", "Mi", "Do", "Fr", "Sa", "So" };  
        for (String tag : tage) {  
            System.out.print(tag + " ");  
        }  
    }  
}
```

## Mehrdimensionale Arrays

*Mehrdimensionale Arrays* werden als geschachtelte Arrays angelegt: Arrays von Arrays usw.

Beispiel:

```
int[][] x = new int[2][3];
```

erzeugt eine 2x3-Matrix, 2 Zeilen und 3 Spalten.

`x[0]` und `x[1]` verweisen jeweils auf ein Array aus 3 Elementen.

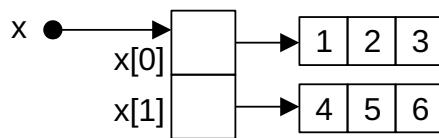
Der Zugriff auf ein Element erfolgt durch Angabe aller erforderlichen Indizes, z. B.

`x[1][0]` greift auf das Element in Zeile 1 und Spalte 0 zu.

Auch eine Initialisierung ist möglich. Werte einer Dimension werden durch geschweifte Klammern zusammengefasst.

```
int[][] x = {{1, 2, 3}, {4, 5, 6}};
```

`x.length` hat den Wert 2.



**Abbildung 5-1:** 2x3-Matrix

Die oben aufgeführte 2x3-Matrix kann auch wie folgt erzeugt werden:

```

int[][] x = new int[2][];
for (int i = 0; i < 2; i++)
    x[i] = new int[3];

public class Matrix {
    public static void main(String[] args) {
        int[][] x = {{1, 2, 3}, {4, 5, 6}};

        System.out.println(x.length);
        for (int[] ints : x) {
            System.out.println(ints.length);
        }

        for (int i = 0; i < x.length; i++) {
            for (int j = 0; j < x[i].length; j++) {
                System.out.print("(" + i + "," + j + "): " + x[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
  
```

Ausgabe des Programms:

```

2
3
3
(0,0): 1 (0,1): 2 (0,2): 3
(1,0): 4 (1,1): 5 (1,2): 6
  
```

Bei der Initialisierung können für jedes Element einer Dimension auch unterschiedlich viele Elemente initialisiert werden.

Das folgende Programm zeigt, wie "nicht rechteckige" Arrays erzeugt werden können:

```

public class Dreieck {
    public static void main(String[] args) {
        int[][] x = { { 1 }, { 1, 2 }, { 1, 2, 3 }, { 1, 2, 3, 4 },
                     { 1, 2, 3, 4, 5 } };
    }
}
  
```

```

        for (int i = 0; i < x.length; i++) {
            for (int j = 0; j < x[i].length; j++) {
                System.out.print(x[i][j]);
            }
            System.out.println();
        }

        // Variante mit foreach
        for (int[] ints : x) {
            for (int anInt : ints) {
                System.out.print(anInt);
            }
            System.out.println();
        }
    }
}

```

Ausgabe des Programms:

```

1
12
123
1234
12345
...

```

### 5.3 Kommandozeilen-Parameter

Beim Aufruf einer *Java-Applikation* kann man dem Programm in der Kommandozeile Parameter, die durch Leerzeichen voneinander getrennt sind, mitgeben:

```
java Programm param1 param2 ...
```

Diese *Kommandozeilen-Parameter* werden der Methode `main` übergeben:

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

`args` ist ein Array vom Typ `String` und enthält die Parameter der Kommandozeile `param1`, `param2` usw. als Elemente.

Soll eine Zeichenkette, die Leerzeichen enthält, als *ein* Parameter gelten, so muss diese Zeichenkette in doppelte Anführungszeichen gesetzt werden.

Das folgende Programm gibt die Kommandozeilen-Parameter auf dem Bildschirm aus.

```

public class Kommandozeile {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println((i + 1) + ". Parameter: " + args[i]);
        }
    }
}

```

Der Aufruf

```
java -cp out/production/kap05 Kommandozeile Dies ist ein "T e s t"
```

liefert die Ausgabe:

1. Parameter: Dies
2. Parameter: ist
3. Parameter: ein
4. Parameter: T e s t

## 5.4 Aufgaben

### Aufgabe 1

Schreiben Sie ein Programm, das für ein mit Zahlen initialisiertes int-Array den kleinsten und den größten Wert sowie den Mittelwert berechnet.

*Lösung: Statistik*

### Aufgabe 2

Schreiben Sie ein Programm, das ein int-Array von  $20 \times 20$  Elementen erzeugt und jedes Element mit dem Produkt seiner Indizes initialisiert.

*Lösung: Matrix*

### Aufgabe 3

Implementieren Sie die Matrixmultiplikation für zweidimensionale double-Arrays A und B. A ist eine  $1 \times m$ -Matrix, B eine  $m \times n$ -Matrix. Das Ergebnis C ist eine  $1 \times n$ -Matrix.

Zur Berechnung wird das Schema *Zeile mal Spalte* angewandt: Das Ergebnis des Elements der i-ten Zeile und der k-ten Spalte von C ergibt sich aus der Multiplikation der Werte der i-ten Zeile von A mit den Werten der k-ten Spalte von B, wobei die einzelnen Produkte aufaddiert werden.

*Lösung: Multiplikation*

### Aufgabe 4

Schreiben Sie ein Programm, das prüft, ob ein eindimensionales int-Array Doppelgänger hat.

*Lösung: Pruefen*



# 6 Klassen, Objekte und Methoden

Dieses Kapitel behandelt objektorientierte Sprachelemente von Java. Im Vordergrund stehen die Begriffe *Klasse*, *Objekt*, *Attribut* und *Methode*. Die objektorientierte Programmierung löst sich von dem Prinzip der Trennung von Daten und Funktionen, wie sie in den traditionellen (prozeduralen) Programmiersprachen üblich ist. Stattdessen werden Daten und Funktionen zu selbständigen Einheiten zusammengefasst.

## Lernziele

In diesem Kapitel lernen Sie

- wie Klassen definiert und Objekte erzeugt werden,
- den Umgang mit Instanzvariablen und -methoden sowie statischen Variablen und statischen Methoden,
- sowie den Umgang mit den speziellen Klassen *Records*.

## 6.1 Klassen und Objekte

Java-Programme bestehen aus Klassendefinitionen. Eine *Klasse* ist eine allgemeingültige Beschreibung von Dingen, die in verschiedenen Ausprägungen vorkommen können, aber alle

- eine *gemeinsame Struktur* und
- ein *gemeinsames Verhalten* haben.

Sie ist ein Bauplan für die Erzeugung von einzelnen konkreten Ausprägungen. Diese Ausprägungen bezeichnet man als *Objekte* oder *Instanzen* der Klasse.

## Zustand und Verhalten

Variablen, auch *Attribute* genannt, *Konstruktoren* und *Methoden* sind die hauptsächlichen Bestandteile einer Klasse. Diese werden zusammengefasst auch als *Member* einer Klasse bezeichnet.

- Die Werte von Attributen sagen etwas über den *Zustand* des Objekts aus.
- Methoden enthalten den ausführbaren Code einer Klasse und beschreiben damit das *Verhalten* des Objekts. Methoden können Werte von Attributen und damit den Zustand des Objekts ändern.
- Konstruktoren sind spezielle Methoden, die zur Objekterzeugung aufgerufen werden.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_6](https://doi.org/10.1007/978-3-658-43574-5_6).

## Klasse

Der grundlegende Aufbau einer *Klasse* sieht wie folgt aus:

```
[Modifizierer] class Klassenname [extends Basisklasse]
[implements Interface-Liste] {

    Attribute
    Konstruktoren
    Methoden

    ...
}
```

Die in eckigen Klammern angegebenen Sprachkonstrukte sind optional. Des Weiteren können im Rumpf der Klasse noch sogenannte Initialisierer, innere Klassen und Interfaces definiert werden. Die Optionen und Zusätze werden an späterer Stelle ausführlich erklärt.

Beispiel: Die Klasse Konto

```
public class Konto {
    private int kontonummer;
    private double saldo;

    public int getKontonummer() {
        return kontonummer;
    }

    public void setKontonummer(int nr) {
        kontonummer = nr;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double betrag) {
        saldo = betrag;
    }

    public void zahleEin(double betrag) {
        saldo += betrag;
    }

    public void zahleAus(double betrag) {
        saldo -= betrag;
    }

    public void info() {
        System.out.println("Kontonummer: " + kontonummer + " Saldo: " + saldo);
    }
}
```

Die Klasse `Konto` enthält die Attribute `kontonummer` und `saldo` und vier Methoden, die die Attribute mit Werten versehen bzw. die aktuellen Attributwerte zurückliefern (get-/set-Methoden), zwei Methoden, die den Saldo erhöhen bzw. vermindern, sowie eine Methode, die die Attributwerte ausgibt.

Die Klasse `Konto` ist mit den Modifizierern `public` und `private` ausgestattet. Auf mit `private` versehene Attribute und Methoden kann nur in der eigenen Klasse zugegriffen werden, nicht von anderen Klassen aus. Bei `public` gilt diese Einschränkung nicht.

### Quelldatei und public-Klasse

Eine Quelldatei sollte *nur eine* Klassendefinition enthalten. Sind mehrere Klassen in einer Datei definiert, so darf höchstens eine den Modifizierer `public` haben. `public` erlaubt jedem den Zugriff auf die Klasse. Enthält die Datei eine `public`-Klasse, so muss diese Datei den gleichen Namen wie die Klasse (abgesehen von der Endung `.java`) haben, wobei auf Groß- und Kleinschreibung zu achten ist.

Im obigen Beispiel hat die Datei, die die Definition der Klasse `Konto` enthält, den Namen `Konto.java`.

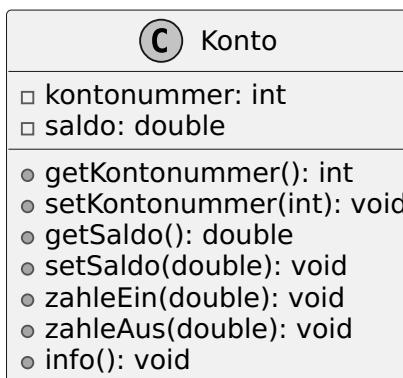


Abbildung 6-1: Klasse Konto

### Objekterzeugung mit new

Um von einer Klasse ein *Objekt* zu erzeugen, wird eine Variable vom Typ der Klasse definiert und anschließend ein mit Hilfe des Operators `new` neu angelegtes Objekt dieser Variablen zugewiesen.

Beispiel:

```
Konto meinKonto;  
meinKonto = new Konto();
```

oder zusammengefasst:

```
Konto meinKonto = new Konto();
```

Der Operator `new` erstellt mit Hilfe der speziellen Methode `Konto()`, die hier vom Compiler automatisch erzeugt wird, Speicherplatz für ein neues Objekt vom Typ `Konto`.

Das Erzeugen eines Objekts wird auch als *Instanziierung* bezeichnet. Objekte werden auch *Instanzen* der entsprechenden Klasse genannt.

## Identität

Objekte besitzen eine *Identität*, die von ihrem Zustand unabhängig ist. Zwei Objekte können sich also in allen Attributwerten gleichen ohne identisch zu sein.

## Referenzvariable

Die Variable `meinKonto` enthält nicht das Objekt selbst, sondern nur einen Adressverweis auf seinen Speicherplatz. Solche Variablen werden *Referenzvariablen* genannt. Klassen sind also *Referenztypen*.

Die vordefinierte Konstante `null` bezeichnet eine leere Referenz. Referenzvariablen mit Wert `null` verweisen nirgendwohin.

Beispiel:

```
Konto meinKonto = new Konto();
Konto k = meinKonto;
```

Die Variablen `meinKonto` und `k` referenzieren hier beide dasselbe Objekt.

Für zwei Referenzvariablen `x` und `y` hat `x == y` genau dann den Wert `true`, wenn beide dasselbe Objekt referenzieren.

## Initialisierung

Der Datentyp eines Attributs kann ein einfacher Datentyp oder ein Referenztyp sein. Nicht explizit initialisierte Attribute erhalten bei der Objekterzeugung automatisch einen *Standardwert*:

- für Zahlen `0`,
- für Variablen vom Typ `boolean` den Wert `false`,
- für `char` das Unicode-Zeichen `\u0000` und
- für Referenzvariablen `null`.

## Punktnotation

Um auf Attribute oder Methoden eines Objekts zugreifen zu können, wird die *Punktnotation* verwendet:

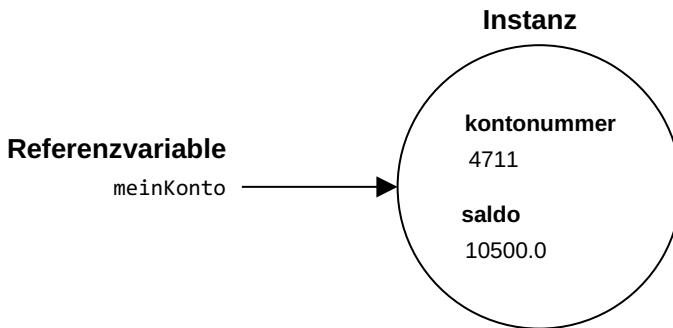
*Referenz.Attribut bzw. Referenz.Methode(...)*

```
public class KontoTest {
    public static void main(String[] args) {
        // Ein Objekt der Klasse Konto wird erzeugt.
        Konto meinKonto = new Konto();

        // Für dieses Objekt werden einige Methoden angewandt.
        meinKonto.setKontonummer(4711);
        meinKonto.setSaldo(500.);
        meinKonto.zahleEin(10000.);
        double saldo = meinKonto.getSaldo();
        System.out.println("Saldo: " + saldo);
    }
}
```

Ausgabe des Programms:

Saldo: 10500.0



**Abbildung 6-2:** Objektreferenz

Wären die Attribute der Klasse `Konto` vor dem Zugriff von außen durch `private` nicht geschützt, könnte man auch unter Umgehung der Methodenaufrufe die Attribute direkt ansprechen:

```
meinKonto.kontonummer = 4711;
meinKonto.saldo = 500.;
meinKonto.saldo += 10000.;
```

Da die Variablen im Allgemeinen zu den Implementierungsdetails einer Klasse gehören, sollten sie verborgen werden, sodass sie – wie im obigen Beispiel – nur von den öffentlichen Methoden der Klasse verändert werden können.

## 6.2 Methoden

Operationen auf Objekten werden mit *Methoden* realisiert. Innerhalb einer Methode können die Attribute und andere Methoden der eigenen Klasse ohne ausdrückliche Objektreferenz benutzt werden.

### Methode

Eine Methodendefinition besteht aus dem *Methodenkopf* und dem *Methodenrumpf*, der die Implementierung enthält.

```
[Modifizierer] Rueckgabetypr Methodename([Parameter-Liste])
[throws Exception-Liste] {
    Anweisungen
}
```

Die in eckigen Klammern angegebenen Sprachkonstrukte sind optional. *throws* wird in Kapitel 13 erklärt.

### Rückgabetypr

Eine Methode kann einen Wert zurückgeben. Der *Rückgabetypr* kann ein einfacher Datentyp oder ein Referenztyp sein. Eine Methode, die keinen Wert zurückgibt, muss den Typ *void* haben.

Werte werden mit Hilfe der Anweisung *return* zurückgegeben:

```
return Ausdruck;
```

Die *return*-Anweisung beendet die Methode. Bei *void*-Methoden fehlt *Ausdruck*. Der *Rückgabewert* kann an der Aufrufstelle weiterverwendet werden. *Ausdruck* muss *zuweisungskompatibel* zum Rückgabetypr der Methode sein.

### Parameter

Die in runden Klammern angegebenen *Parameter* (Typ und Name) spezifizieren die *Argumente*, die der Methode beim Aufruf übergeben werden. Mehrere Parameter werden durch Kommas getrennt.

### call by value

Für jeden Parameter wird innerhalb der Methode ein eigener Speicherplatz erzeugt, in den der Parameterwert kopiert wird (*call by value*). Dadurch bleiben die Variablen, deren Werte beim Aufruf der Methode als Argumente übergeben werden, unverändert.

Ist das Argument vom Referenztyp, verweist also auf ein Objekt, so kann die Methode dieses Objekt verändern, da die Referenzvariable und ihre Kopie auf dasselbe Objekt verweisen.

Das folgende Programm veranschaulicht den Parametermechanismus.

```
public class ParamTest {  
    public void test(double betrag, Konto kto) {  
        betrag += 100.;  
        kto.zahleEin(betrag);  
    }  
  
    public static void main(String[] args) {  
        ParamTest p = new ParamTest();  
  
        double wert = 1000.;  
        Konto konto = new Konto();  
  
        System.out.println("Vorher: wert=" + wert + " saldo=" + konto.getSaldo());  
        p.test(wert, konto);  
        System.out.println("Nachher: wert=" + wert + " saldo=" + konto.getSaldo());  
    }  
}
```

Das Programm startet mit der Ausführung der `main`-Methode. Vor dem Aufruf der Methode `test` für das Objekt `p` wird

Vorher: `wert=1000.0 saldo=0.0`

ausgegeben.

Beim Aufruf von `test` werden die Parameter `betrag` und `kto` als Variablen erzeugt. In `betrag` wird der Wert von `wert`, in `kto` die Referenz auf das `Konto`-Objekt `konto` (also die Adresse dieses Objekts) kopiert. Nach Ausführung der Methode enthält `wert` unverändert den Wert `1000.0`, der Saldo des `Konto`-Objekts `konto` beträgt nun `1100.0`.

Ausgabe nach dem Aufruf der Methode `test`:

Nachher: `wert=1000.0 saldo=1100.0`

## Lokale Variable

Variablen, die innerhalb einer Methode oder innerhalb eines Blocks einer Methode definiert werden, bezeichnet man als *lokale Variablen*. Sie werden angelegt, wenn die Ausführung der Methode bzw. des Blocks beginnt, und werden zerstört, wenn die Methode bzw. der Block verlassen wird. Variablen innerer Blöcke verdecken gleichnamige Variablen äußerer Blöcke, insbesondere gleichnamige Attribute der Klasse.

Die Parameter einer Methode sind als lokale Variablen des Methodenrumpfs aufzufassen.

Variablen, die im Initialisierungsteil des Kopfs einer `for`-Anweisung definiert werden, sind nur innerhalb der `for`-Anweisung gültig.

Innerhalb eines Blocks kann auf die Variablen der umgebenden Blöcke bzw. der umgebenden Methode sowie auf die Attribute der umschließenden Klasse zugegriffen werden. Es ist nicht erlaubt, eine bereits definierte lokale Variable in einem tiefer geschachtelten Block erneut mit dem gleichen Namen zu definieren. Allerdings darf ihr Name mit einem Attributnamen der Klasse übereinstimmen. Definitionen lokaler Variablen können mit anderen Anweisungen gemischt werden.

### Lokale `final`-Variable

Beginnt die Definition einer lokalen Variablen mit `final`, so handelt es sich um eine *Konstante*, die entweder sofort bei ihrer Definition mit einem Wert initialisiert wird oder der einmal, vor dem ersten Zugriff, ein Wert zugewiesen wird. Der Wert ist dann unveränderbar. Die Parameter einer Methode können `final` deklariert werden. Eine Zuweisung an sie im Methodenrumpf ist dann nicht möglich.

### Die Referenz `this`

Innerhalb einer Methode bezeichnet `this` einen Wert, der dasjenige Objekt referenziert, für das die Methode aufgerufen wurde. `this` bezeichnet also das gerade "handelnde" Objekt.

`this` wird benutzt, um auf "verdeckte" Attribute der eigenen Klasse zuzugreifen, die eigene Instanz als Wert zurückzugeben oder sie als Argument beim Aufruf einer anderen Methode zu verwenden.

Beispiel:

```
public void setKontonummer(int kontonummer) {  
    this.kontonummer = kontonummer;  
}
```

In diesem Beispiel verdeckt die lokale Variable das Attribut `kontonummer` der Klasse `Konto`. Mit `this` wird das Attribut trotz Namensgleichheit zugänglich.

### Signatur

Der Name einer Methode und ihre Parameterliste (Parametertypen in der gegebenen Reihenfolge) bilden gemeinsam die *Signatur* der Methode. Der Rückgabetypr gehört nicht zur Signatur.

Um eine Methode einer Klasse in einem Programm aufrufen zu können, muss nur ihre Signatur, ihr Rückgabetypr und ihre semantische Wirkung (z. B. in Form einer

---

umgangssprachlichen Beschreibung) bekannt sein. Wie die Methode im Methodenrumpf implementiert ist, interessiert nicht (*Black Box*).

## Überladen / Overloading

Es können mehrere Methoden mit gleichem Namen, aber unterschiedlichen Signaturen definiert werden. Der Compiler entscheidet beim Methodenaufruf, welche der definierten Methoden aufgerufen wird. Dabei werden auch mögliche implizite Typumwandlungen einbezogen und die Methode ausgewählt, die am genauesten passt.

Diese Technik nennt man *Überladen* (Overloading) von Methoden.

Überladene Methoden unterscheiden sich also in der Parameterliste. Sie sollten allerdings eine vergleichbare Funktionalität haben, um keine Verwirrung zu stiften.

Beispiel:

```
public int max(int a, int b) {  
    return a < b ? b : a;  
}  
  
public double max(double a, double b) {  
    return a < b ? b : a;  
}  
  
public int max(int a, int b, int c) {  
    return max(max(a, b), c);  
}
```

Die Signaturen dieser Methoden sind alle voneinander verschieden:

```
max int int  
max double double  
max int int int  
  
public class OverloadingTest {  
    public int max(int a, int b) {  
        System.out.println("Signatur: max int int");  
        return a < b ? b : a;  
    }  
  
    public double max(double a, double b) {  
        System.out.println("Signatur: max double double");  
        return a < b ? b : a;  
    }  
  
    public int max(int a, int b, int c) {  
        System.out.println("Signatur: max int int int");  
        return max(max(a, b), c);  
    }  
}
```

---

```

public static void main(String[] args) {
    OverloadingTest ot = new OverloadingTest();

    System.out.println("max(1, 3): " + ot.max(1, 3));
    System.out.println();
    System.out.println("max(1, 3, 2): " + ot.max(1, 3, 2));
    System.out.println();
    System.out.println("max(1., 3.): " + ot.max(1., 3.));
    System.out.println();
    System.out.println("max(1., 3): " + ot.max(1., 3));
}

}

```

Die Ausgabe des Programms ist:

```

Signatur: max int int
max(1, 3): 3

Signatur: max int int int
Signatur: max int int
Signatur: max int int
max(1, 3, 2): 3

Signatur: max double double
max(1., 3.): 3.0

Signatur: max double double
max(1., 3): 3.0

```

## 6.3 Konstruktoren

Konstruktoren sind spezielle Methoden, die bei der Erzeugung eines Objekts mit `new` aufgerufen werden. Sie werden häufig genutzt, um einen Anfangszustand für das Objekt herzustellen, der durch eine einfache Initialisierung nicht zu erzielen ist. Ein *Konstruktor* trägt den Namen der zugehörigen Klasse und hat keinen Rückgabetyp. Ansonsten wird er wie eine Methode definiert und kann auch überladen werden.

### Standardkonstruktor

Wenn für eine Klasse kein Konstruktor explizit deklariert ist, wird ein sogenannter *Standardkonstruktor* ohne Parameter vom Compiler selbst bereitgestellt. Ist ein Konstruktor mit oder ohne Parameter explizit definiert, so erzeugt der Compiler von sich aus keinen Standardkonstruktor mehr.

Beispiel:

Für die Klasse `Konto` werden vier Konstruktoren definiert. Beim zweiten Konstruktor kann eine Kontonummer, beim dritten Konstruktor zusätzlich ein Anfangssaldo mitgegeben werden. Der vierte Konstruktor erzeugt eine Kopie eines

---

bereits bestehenden Objekts derselben Klasse. Ein solcher Konstruktor wird auch als *Kopierkonstruktor* bezeichnet.

Da die folgende Klasse im selben Verzeichnis liegt wie die oben behandelte Klasse Konto, muss sie einen anderen Namen erhalten, hier Konto2:

```
public Konto2() { }

public Konto2(int kontonummer) {
    this.kontonummer = kontonummer;
}

public Konto2(int kontonummer, double saldo) {
    this.kontonummer = kontonummer;
    this.saldo = saldo;
}

public Konto2(Konto2 k) {
    kontonummer = k.kontonummer;
    saldo = k.saldo;
}
```

Test der Konstruktoren von Klasse Konto2:

```
public class KonstrTest {
    public static void main(String[] args) {
        Konto2 k1 = new Konto2();
        Konto2 k2 = new Konto2(4711);
        Konto2 k3 = new Konto2(1234, 1000.);
        Konto2 k4 = new Konto2(k3);

        k1.info();
        k2.info();
        k3.info();
        k4.info();

        new Konto2(5678, 2000.).info();
    }
}
```

Ausgabe des Programms:

```
Kontonummer: 0 Saldo: 0.0
Kontonummer: 4711 Saldo: 0.0
Kontonummer: 1234 Saldo: 1000.0
Kontonummer: 1234 Saldo: 1000.0
Kontonummer: 5678 Saldo: 2000.0
```

Dieses Programm zeigt auch, dass Objekte ohne eigene Referenzvariable erzeugt werden können (*anonyme Objekte*). Die Methode `info` wird hier direkt für das zuletzt erzeugte `Konto2`-Objekt aufgerufen. Danach ist das Objekt nicht mehr verfügbar.

### **this(...)**

Konstruktoren können sich gegenseitig aufrufen, sodass vorhandener Programmcode wiederverwendet werden kann. Der Aufruf eines Konstruktors muss dann *als erste Anweisung* mit dem Namen `this` erfolgen.

Beispiel:

Der zweite Konstruktor des vorigen Beispiels hätte auch so geschrieben werden können:

```
public Konto2(int kontonummer) {
    this(kontonummer, 0.);
}
```

### **Initialisierungsblock**

Ein *Initialisierungsblock*

```
{
    Anweisungen
}
```

ist ein Block von Anweisungen, der außerhalb aller Attribut-, Konstruktor- und Methodendefinitionen erscheint. Er wird beim Aufruf eines Konstruktors immer als Erstes ausgeführt. Mehrere Initialisierungsblöcke werden dabei in der aufgeschriebenen Reihenfolge durchlaufen.

### **Garbage Collector**

Die verschiedenen Elemente eines Programms werden in unterschiedlichen Bereichen des Speichers innerhalb einer JVM (Java Virtual Machine) verwaltet: verschiedene *Stack-Speicher* und ein gemeinsamer *Heap-Speicher*.

Im *Stack* werden lokale Variablen und Methodenparameter verwaltet. Im *Heap* liegen die erzeugten Objekte einer Klasse.

Der *Heap* wird vom *Garbage Collector* des Java-Laufzeitsystems automatisch freigegeben, wenn er nicht mehr gebraucht wird.

Wenn z. B. die Methode, in der ein Objekt erzeugt wurde, endet oder die Referenzvariable auf den Wert `null` gesetzt wurde, kann das Objekt nicht mehr benutzt werden, da keine Referenz mehr darauf verweist.

Wann die Speicherbereinigung ausgeführt wird, ist nicht festgelegt. Sie startet normalerweise nur dann, wenn Speicherplatz knapp wird und neue Objekte benötigt werden.

## 6.4 Klassenvariablen und Klassenmethoden

Es können Attribute und Methoden definiert werden, deren Nutzung nicht an die Existenz von Objekten gebunden ist.

### Klassenvariable / statische Variable

Ein Attribut einer Klasse, dessen Definition mit dem Schlüsselwort `static` versehen ist, nennt man *Klassenvariable*. Klassenvariablen (auch *statische Variablen* genannt) werden bereits beim Laden der Klasse erzeugt.

Es existiert nur ein Exemplar dieser Variablen, unabhängig von der Anzahl der Instanzen der Klasse, und ihre Lebensdauer erstreckt sich über das ganze Programm. Der Zugriff von außerhalb der Klasse erfolgt über den Klassennamen in der Form:

*Klassename.Variablename*

Der Zugriff über ein Objekt der Klasse ist ebenfalls möglich.

### Instanzvariable

Nicht-statische Attribute werden zur Unterscheidung auch *Instanzvariablen* genannt.

Mit Klassenvariablen können beispielsweise Instanzen-Zähler realisiert werden, wie das folgende Beispiel zeigt.

```
public class ZaehlerTest {  
    private static int zaehler;  
  
    public ZaehlerTest() {  
        zaehler++;  
    }  
  
    public static void main(String[] args) {  
        new ZaehlerTest();  
        new ZaehlerTest();  
        new ZaehlerTest();  
        System.out.println(ZaehlerTest.zaehler);  
    }  
}
```

Die Instanzen der Klasse `zaehlerTest` teilen sich die gemeinsame Variable `zaehler`. Das Programm gibt die Zahl 3 aus.

### Klassenmethode / statische Methode

Eine Methode, deren Definition mit dem Schlüsselwort `static` versehen ist, nennt man *Klassenmethode*. Klassenmethoden (auch *statische Methoden* genannt) können von außerhalb der Klasse über den Klassennamen aufgerufen werden:

`Klassenname.Methodename(...)`

Man braucht keine Instanz der Klasse, um sie aufrufen zu können.

### Instanzmethode

Nicht-statische Methoden werden zur Unterscheidung *Instanzmethoden* genannt.

Klassenmethoden dürfen nur auf Klassenvariablen und Klassenmethoden zugreifen. Klassenmethoden können wie bei Instanzmethoden überladen werden.

### main-Methode

Durch Hinzufügen der Methode

```
public static void main(String[] args) { ... }
```

wird eine beliebige Klasse zu einer *Java-Applikation*. Diese Klassenmethode ist der Startpunkt der Anwendung. Sie wird vom Java-Laufzeitsystem aufgerufen. `main` ist als Klassenmethode definiert, da zum Startzeitpunkt noch kein Objekt existiert. Ein Klasse, die eine `main`-Methode enthält, nennt man auch *ausführbare Klasse*.

```
public class MaxTest {
    public static int max(int a, int b) {
        return a < b ? b : a;
    }

    public static void main(String[] args) {
        System.out.println("Maximum: " + MaxTest.max(5, 3));
    }
}
```

### Statische Initialisierung

Ein *statischer Initialisierungsblock* ist ein Block von Anweisungen, der außerhalb aller Attribut-, Konstruktor- und Methodendefinitionen erscheint und mit dem Schlüsselwort `static` eingeleitet wird. Er kann nur auf Klassenvariablen zugreifen und Klassenmethoden aufrufen.

```
static {
    Anweisungen
}
```

Eine Klasse kann mehrere statische Initialisierungsblöcke haben. Sie werden in der aufgeschriebenen Reihenfolge ein einziges Mal ausgeführt, wenn die Klasse geladen wird.

## 6.5 Varargs

Mit *Varargs* (*variable length argument lists*) lassen sich Methoden mit einer beliebigen Anzahl von Parametern desselben Typs definieren. Hierzu wird nur ein einziger Parameter, der sogenannte *Vararg*-Parameter, benötigt. Zur Kennzeichnung werden dem Datentyp dieses Parameters drei Punkte angefügt.

Beispiel:

```
int sum(int... values)
int min(int firstValue, int... remainingValues)
```

Es darf nur ein einziger *Vararg*-Parameter in der Parameterliste der Methode vorkommen. Dieser muss der letzte in einer längeren Parameterliste sein. Im Methodenrumpf steht der *Vararg*-Parameter als Array zur Verfügung.

Im Folgenden werden die Methoden `sum` und `min` gezeigt, die mit beliebig vielen `int`-Argumenten aufgerufen werden können.

```
public class VarargTest {
    public static int sum(int... values) {
        int sum = 0;
        for (int v : values) {
            sum += v;
        }
        return sum;
    }

    public static int min(int firstValue, int... remainingValues) {
        int min = firstValue;
        for (int v : remainingValues) {
            if (v < min)
                min = v;
        }
        return min;
    }

    public static void main(String[] args) {
        System.out.println(sum(1, 2));
        System.out.println(sum(1, 10, 100, 1000));
        System.out.println(sum());
        System.out.println(min(3, 5, 2));
    }
}
```

Ausgabe des Programms:

```
3  
1111  
0  
2
```

## 6.6 Iterative und rekursive Problemlösungen

Meist können Probleme effizient *iterativ* gelöst werden. Dabei werden mehrfach auszuführende Schritte in Schleifen umgesetzt. Eine andere Lösungsstrategie ist die Rekursion. Wir wollen beide Lösungsarten (*Iteration* und *Rekursion*) anhand eines einfachen Beispiels vorstellen.

Beispiel:

Es soll die Summe der ersten n Zahlen ermittelt werden, also  $1 + 2 + \dots + n$ .

```
public class Iteration {  
    public static long sum(int n) {  
        long sum = 0;  
        for (int i = 1; i <= n; i++) {  
            sum += i;  
        }  
        return sum;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(sum(100));  
    }  
}
```

Ausgabe des Programms:

```
5050
```

Eine andere Möglichkeit besteht darin, eine Funktion zu definieren, die für mehrfach auszuführende Schritte sich selbst wiederholt aufruft, also *rekursiv* arbeitet.<sup>1</sup>

Für die Summe der Zahlen von 1 bis n besteht der folgende Zusammenhang:

```
sum(1) = 1  
sum(n) = sum(n-1) + n, falls n > 1
```

---

1 <https://de.wikipedia.org/wiki/Rekursion>

---

```

public class Rekursion {
    public static long sum(int n) {
        if (n == 1) {
            System.out.println("sum(1) returns 1");
            return 1;
        }

        System.out.println("sum(" + n + ") returns sum(" + (n - 1) + ") + " + n);
        return sum(n - 1) + n;
    }

    public static void main(String[] args) {
        System.out.println(sum(3));
    }
}

```

Wir haben zur Veranschaulichung das Programm mit Ausgaben instrumentiert:

```

sum(3) returns sum(2) + 3
sum(2) returns sum(1) + 2
sum(1) returns 1

```

sum wird also dreimal aufgerufen. Es gelten folgende Gleichungen:

```

sum(3) = sum(2) + 3
sum(2) = sum(1) + 2
sum(1) = 1

```

Nun können wir *rückwärts* die konkreten Ergebnisse einsetzen:

```

sum(3) = sum(2) + 3 = 3 + 3 = 6
sum(2) = sum(1) + 2 = 1 + 2 = 3
sum(1) = 1

```

Die rekursive Methode reduziert das Argument bei jedem Schritt. Hat das Argument einen bestimmten Wert erreicht (hier 1), bricht die Rekursion mit dem Ergebnis als Rückgabewert ab.

Iterative Verfahren sind meist effizienter und benötigen weniger Speicherplatz. Grund hierfür ist, dass beim rekursiven Verfahren die wiederholten Methodenaufrufe mit allen zwischengespeicherten Werten auf dem Stack abgelegt werden. Das kann zu einem Pufferüberlauf führen, da der Speicherplatz für einen Stack begrenzt ist.

Andererseits gibt es komplexe Problemstellungen, die elegant nur rekursiv mit wenigen Quellcodezeilen gelöst werden können, z. B. die *Türme von Hanoi*.<sup>2</sup>

---

<sup>2</sup> [https://de.wikipedia.org/wiki/Türme\\_von\\_Hanoi](https://de.wikipedia.org/wiki/Türme_von_Hanoi)

## 6.7 Records

Mit Java 16 wurde eine vereinfachte Form von Klassen eingeführt: *Records*.

Hierzu ein erstes Beispiel:

```
public record Point(int x, int y) { }
```

Die Attribute werden aus den Parametern hergeleitet. Der Compiler generiert einen *kanonischen Konstruktor*, der für jeden Parameter eine Instanzvariable vorsieht und die übergebenen Werte setzt. Die Instanzvariablen sind nach der Objekterzeugung *unveränderlich*.

Im Rumpf des Records können keine weiteren Instanzvariablen definiert werden, statische Variablen sind aber möglich.

Der Zugriff auf die Instanzvariablen erfolgt mittels gleichnamiger Methoden. Im Beispiel also mit: `x()` und `y()`. Die Attribute selbst sind *private*.

```
public class RecordTest {
    public static void main(String[] args) {
        var p1 = new Point(2, 3);
        System.out.println(p1.x() + ", " + p1.y());
    }
}
```

### Erweiterungen

Im Rumpf des Records können weitere Instanzmethoden und statische Methoden definiert werden.

Records können auch um zusätzliche Konstruktoren erweitert werden. Dabei müssen immer alle Instanzvariablen durch Delegation an den kanonischen Konstruktor gesetzt werden.

Um die Gültigkeit der übergebenen Parameter zu prüfen, kann der kanonische Konstruktor überschrieben werden.

```
public record Point2(int x, int y) {
    private static final int ZERO = 0;

    public Point2() {
        this(ZERO, ZERO);
    }

    public Point2(int value) {
        this(value, value);
    }

    public Point2(int x, int y) {
        this.x = x >= 0 ? x : 0;
        this.y = y >= 0 ? y : 0;
    }
}
```

```
public void display() {
    System.out.println("(" + this.x + ", " + this.y + ")");
}

public class RecordTest {
    public static void main(String[] args) {
        ...

        var p2 = new Point2();
        p2.display();
        var p3 = new Point2(1);
        p3.display();
        var p4 = new Point2(2, 3);
        p4.display();
        var p5 = new Point2(-2, 3);
        p5.display();
    }
}
```

Ausgabe:

```
(0, 0)
(1, 1)
(2, 3)
(0, 3)
```

Records stellen automatisch Implementierungen der Methoden

`equals()`, `hashCode()` und `toString()`

zur Verfügung. Damit können Objekte von Records verglichen und in lesbarer Form ausgegeben werden. Diese Methoden werden in Kapitel 14 und 15 näher erläutert.

### Records für Rückgabewerte von Methoden

Wenn eine Methode mehrere Werte zurückgeben soll, müssen bisher (vor Java 16) Arrays oder "echte" Klassen verwendet werden. Hier können nun Records als Rückgabetyp die Programmierung vereinfachen. Der folgende Abschnitt enthält eine Aufgabe zu diesem Thema.

## 6.8 Aufgaben

### Aufgabe 1

Wie kann eine Methode Daten von außen aufnehmen und nach außen weiterreichen?

*Lösung: Daten.txt*

**Aufgabe 2**

Besitzt folgende Klasse einen *Standardkonstruktor*?

```
public class A {  
    private int a;  
  
    public A(int i) {  
        a = i;  
    }  
}
```

*Lösung:* Konstruktor.txt

**Aufgabe 3**

Testen Sie den unter "Lokale Variable" in Kapitel 6.2 beschriebenen Sachverhalt.

*Lösung:* LocalTest

**Aufgabe 4**

Entwickeln Sie die Klasse Zaeher, die als Instanzvariable einen `int`-Wert enthält, diesen mit der Methode `hochzaehlen()` um 1 erhöhen, mit `zuruecksetzen()` auf 0 setzen und mit `getWert()` den Wert zurückgeben kann.

Testen Sie diese Funktionalität innerhalb der `main`-Methode.

*Lösung:* Zaehler

**Aufgabe 5**

Implementieren Sie die Klasse Sparbuch mit den Attributen `kontonummer`, `kapital` und `zinssatz` und den folgenden Methoden:

`zahleEin`

erhöht das Guthaben um einen bestimmten Betrag.

`hebeAb`

vermindert das Guthaben um einen bestimmten Betrag.

`getErtrag`

berechnet das Kapital mit Zins und Zinseszins nach einer vorgegebenen Laufzeit.

`verzinse`

erhöht das Guthaben um den Jahreszins.

`getKontonummer`

liefert die Kontonummer.

`getKapital`

liefert das Guthaben.

`getZinssatz`

liefert den Zinssatz.

Testen Sie die Methoden dieser Klasse.

*Lösung: Sparbuch*

### Aufgabe 6

Erstellen Sie eine Klasse `Abschreibung`, die den Anschaffungspreis, die Anzahl der Nutzungsjahre und den Abschreibungssatz enthält.

Vereinbaren Sie zwei Konstruktoren und die beiden Abschreibungsmethoden: lineare und geometrisch-degressive Abschreibung. Die Buchwerte für die einzelnen Jahre sollen am Bildschirm ausgegeben werden.

Testen Sie die Methoden dieser Klasse.

*Lösung: Abschreibung*

### Aufgabe 7

Erstellen Sie die Klasse `Beleg`, deren Objekte bei ihrer Erzeugung automatisch eine bei der Zahl 10000 beginnende laufende Belegnummer erhalten. Tipp: Verwenden Sie eine Klassenvariable.

*Lösung: Beleg*

### Aufgabe 8

Ein Stapel (Stack) ist eine Datenstruktur, in der Daten nach dem Prinzip "Last in, first out" (LIFO) verwaltet werden. Implementieren Sie einen Stapel auf der Basis eines Arrays mit den folgenden Methoden:

`void push(int e)`

legt eine Zahl oben auf den Stapel.

`int pop()`

entfernt das oberste Element des Stapels.

Ist das Array voll, soll automatisch ein neues Array mit doppelter Länge erzeugt werden, mit dem dann weiter gearbeitet werden kann. Die Elemente des alten Arrays müssen vorher in das neue Array kopiert werden. Kapazitätsprüfung, Erzeugung des neuen Arrays und Übernahme der bisherigen Werte sollen in der Methode `push` erfolgen.

*Lösung: Stapel*

### Aufgabe 9

Erstellen Sie die Klassen `Artikel` und `Auftrag`. In `Artikel` sollen Nummer (`int id`) und Preis (`double preis`) des Artikels gespeichert werden. In `Auftrag` ist eine Referenz auf den jeweils bestellten Artikel aufzunehmen sowie die bestellte Menge (`int menge`) dieses Artikels.

Entwickeln Sie für beide Klassen geeignete Konstruktoren und Methoden (get-/set-Methoden). Die Klasse `Auftrag` soll die folgende Klassenmethode enthalten:

```
public static double getGesamtwert(Auftrag... aufträge)
```

Diese soll die Summe aller einzelnen Auftragswerte (Menge x Artikelpreis) liefern.

*Lösung: AuftragTest*

### Aufgabe 10

Entwickeln Sie die Klasse `Rechteck` mit Instanzvariablen für Breite und Höhe und mit einem geeigneten Konstruktor.

Um Quadrate erzeugen zu können, soll ein weiterer Konstruktor mit nur einem Parameter (die Länge) definiert werden. Implementieren Sie Methoden zur Berechnung des Flächeninhalts und des Umfangs.

*Lösung: Rechteck*

### Aufgabe 11

Probieren Sie aus, wie sich das Laufzeitsystem bei einer Rekursion verhält, die nicht terminiert (Endlosrekursion).

*Lösung: Endlos*

### Aufgabe 12

Eine `double`-Zahl soll solange halbiert werden, bis ein bestimmter Wert erreicht oder unterschritten wurde. Lösen Sie das Problem rekursiv.

*Lösung: Halbieren*

### Aufgabe 13

Implementieren Sie eine rekursive Methode, die eine Zahl `n` bis auf 0 herunter zählt. Wenn `n` gleich 0 ist, soll das Wort "Zero" ausgegeben werden.

*Lösung: CountDown*

### Aufgabe 14

a) Realisieren Sie eine Klassenmethode, die die Verzinsung eines Anfangskapitals nach einer bestimmten Anzahl Jahre *iterativ* berechnet:

```
public static double zinsen(double kapital, double zinssatz, int jahre)
```

b) Innerhalb des Rumpfs einer Methode kann sich die Methode selbst aufrufen (Rekursion). Implementieren Sie die Methode aus a) *rekursiv*.

Bezeichnet `k(n)` das Gesamtkapital nach `n` Jahren (`n >= 0`), dann gilt:

```
k(0) = kapital  
k(n) = k(n-1) * (1 + zinssatz) für n > 0
```

Bei der Lösung wird eine Fallunterscheidung getroffen:

Hat `jahre` den Wert 0, wird `kapital` zurückgegeben. In allen anderen Fällen erfolgt ein Selbstaufruf der Methode, wobei jedoch der Wert des dritten Parameters um 1 vermindert ist. Zurückgegeben wird das Produkt aus `(1 + zinssatz)` und dem Rückgabewert der aufgerufenen Methode.

*Lösung: Verzinsung*

### Aufgabe 15

Implementieren Sie mit Hilfe eines Arrays einen *Ringpuffer*, in den ganze Zahlen geschrieben werden können. Der Ringpuffer hat eine feste Länge. Ist der Puffer voll, so soll der jeweils älteste Eintrag überschrieben werden. Ein Index gibt die Position im Array an, an der die nächste Schreiboperation stattfindet. Nach jedem Schreiben wird der Index um 1 erhöht und auf 0 gesetzt, wenn die obere Grenze des Arrays überschritten wurde. Es soll auch eine Methode geben, die den gesamten Inhalt des Puffers ausgibt.

*Lösung: Ringpuffer*

### Aufgabe 16

Die Methode

```
int tage(int jahr, int monat)
```

soll die Anzahl der Tage des angegebenen Monats im angegebenen Jahr zurückgeben. Bei ungültiger Monatszahl soll 0 zurückgegeben werden. Berücksichtigen Sie zur Ermittlung der Anzahl Tage des Monats Februar, ob das angegebene Jahr ein Schaltjahr ist (siehe Aufgaben in Kapitel 4).

*Lösung: Tage*

### Aufgabe 17

Schreiben Sie eine Klasse `Datum`, die die Attribute `tag`, `monat` und `jahr` hat, mit einem Konstruktor für diese drei Attribute. Implementieren Sie die Methode `display`, die ein gültiges Datum formatiert ausgibt. Nutzen Sie die Methode `tage` in Aufgabe 16, um zu prüfen, ob es sich um ein gültiges Datum handelt.

*Lösung: Datum*

### Aufgabe 18

Implementieren Sie eine Methode, die das Minimum und Maximum von `double`-Zahlen berechnet und als Instanz eines geeigneten Records zurückgibt:

```
public static MinMaxValues minMax(double a, double... args) { ... }
```

*Lösung: MinMax*

### Aufgabe 19

Implementieren Sie die Klasse `Flasche`

mit den Attributen: `id`, `inhalt` (in ml) und `fassungsVermoege` (in ml),  
den Konstruktoren:

```
Flasche(int id, int inhalt, int fassungsVermoege)  
Flasche(int id, int inhalt)  
Flasche(int id, double prozent, int fassungsVermoege)
```

Beim zweiten Konstruktor wird ein Standard-Fassungsvermögen (z. B. 500) unterstellt, beim dritten Konstruktor ist der Inhalt in Prozent vom Fassungsvermögen angegeben.

Realisieren Sie Methoden

- zum Setzen des Inhalts, dabei darf die Flasche nicht überlaufen,
- zum Nachfüllen,
- zum Verschütten einer gewissen Menge,
- zum Umfüllen in eine andere Flasche, die leer oder auch nicht leer sein kann,
- zum Feststellen, ob die Flasche leer oder voll ist,
- zur Ermittlung des Füllgrads in Prozent,
- zum Feststellen, welche die größere Flasche (gemäß Fassungsvermögen) von zwei Flaschen ist.

*Lösung: Flasche*



# 7 Pakete

Eine größere Anwendung besteht aus einer Vielzahl von Klassen, die aus Gründen der Zweckmäßigkeit (z. B. zur besseren Übersicht) zu Einheiten, den *Paketen*, zusammengefasst werden können. Dabei darf der Name einer Klasse in einem Paket mit dem einer Klasse in einem anderen Paket übereinstimmen.

## Lernziele

In diesem Kapitel lernen Sie

- wie Klassen in Paketen organisiert,
- wie Klassen anderer Pakete verwendet und
- wie Java-Bibliotheken (jar-Dateien) erzeugt werden können.

## 7.1 Pakete erzeugen und nutzen

Der *Name eines Pakets* besteht im Allgemeinen aus mehreren mit einem Punkt getrennten Teilen.

Beispiel:

```
admin.gui
```

### package

Um eine Klasse<sup>1</sup> einem bestimmten Paket zuzuordnen, muss als *erste* Anweisung im Quellcode die folgende Anweisung stehen:

```
package paketname;
```

Der Paketname wird so auf das Dateiverzeichnis abgebildet, dass jedem Namensteil ein Unterverzeichnis entspricht.

Beispielsweise muss der Bytecode zu einer Klasse mit der package-Klausel

```
package admin.gui;
```

im Unterverzeichnis `gui` des Verzeichnisses `admin` liegen:

```
admin/gui
```

Obwohl nicht vorgegeben, sollte der zugehörige Quellcode dann ebenfalls im Verzeichnis `admin/gui` liegen. Das erhöht die Übersichtlichkeit.

---

<sup>1</sup> Sämtliche Ausführungen gelten auch für Interfaces.

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_7](https://doi.org/10.1007/978-3-658-43574-5_7).

Also z. B. der Quellcode im Verzeichnis

*projektpfad/src/admin/gui*

und der Bytecode im Verzeichnis

*projektpfad/bin/admin/gui*

Innerhalb eines Paktes kann man auf alle anderen Klassen und Interfaces desselben Pakets direkt zugreifen. Um eine Klasse aus einem anderen Paket verwenden zu können, muss der Paketname mit angegeben werden:

*paketname.Klassenname*

## **import**

Eine andere Möglichkeit ist, die gewünschte Klasse am Anfang der Quelldatei (hinter einer möglichen package-Anweisung) bekannt zu machen:

*import paketname.Klassenname;*

Im darauf folgenden Quellcode kann dann die Klasse mit ihrem einfachen Namen angegeben werden, sofern es in allen anderen importierten Paketen keine Klasse mit dem gleichen Namen gibt.

Mit der folgenden Anweisung können *alle* Klassen eines Pakets zugänglich gemacht werden:

*import paketname.\*;*

Beispiel:

*import admin.\*;*

macht alle Paketinhalte zugänglich, die direkt in `admin` liegen, aber nicht die Inhalte, die in `gui` liegen. Hierzu ist dann beispielsweise

*import admin.gui.\*;*

nötig.

## **Wie wird der Bytecode vom Compiler bzw. der JVM gefunden?**

Eine Klasse wird vom Compiler bzw. Laufzeitsystem erst dann gesucht, wenn sie im Programm benötigt wird.

Beispiel:

*import admin.gui.MainFrame;*

`MainFrame.class` ist im Verzeichnis `admin/gui` gespeichert. Wo dieses Verzeichnis im Dateisystem genau liegt, kann in der Umgebungsvariablen `CLASSPATH` angegeben werden.

Bei *Windows* geschieht das in der Eingabeaufforderung mit Hilfe des `set`-Befehls beispielsweise wie folgt:

```
set CLASSPATH=.;C:\Users\emil\Projekt1
```

Hier wird nach dem Unterverzeichnis `admin` im aktuellen Verzeichnis `(.)` und im Verzeichnis `C:\Users\emil\Projekt1` gesucht.

Bei *Linux* wird der Klassenpfad wie folgt im Terminal festgelegt:

```
export CLASSPATH=.:~/home/emil/Projekt1
```

## Default-Paket

Klassen, in denen die `package`-Anweisung fehlt, gehören automatisch zu einem *unbenannten Paket* (Default-Paket). Sie können ohne explizite `import`-Anweisung gesucht werden.

Wichtige Pakete der Java-Entwicklungsumgebung sind z. B.

```
java.lang, java.io, java.util, javax.swing
```

`java.lang` enthält grundlegende Klassen und Interfaces.

Besonderheit:

Klassen aus `java.lang` werden ohne explizite `import`-Anweisung in jeden Code automatisch importiert.

## Namenskonvention für Pakete

Soll die entwickelte Software später ausgeliefert werden, ist es wichtig, dass die vollständigen Namen für Klassen nicht zufällig mit den Namen anderer Anwendungen kollidieren. In vielen Projekten werden Anwendungen auf Basis von Klassen fremder Projekte (z. B. Frameworks) entwickelt.

Folgende Konvention ist die Regel (*Reverse Domain Name Pattern*):

Entwickelt die Firma ABC mit der Webadresse `www.abc.de` eine Software `xyz`, so fangen alle Paketnamen dieser Software mit `de.abc.xyz` an. Die Eindeutigkeit des URL `www.abc.de` garantiert dann die Kollisionsfreiheit.

## Statische import-Klausel

Statische Methoden und statische Variablen werden über den Klassennamen angesprochen. Diese statischen Elemente einer Klasse können so importiert werden, dass sie ohne vorangestellten Klassennamen verwendet werden können:

```
import static paketname.Klassename.bezeichner;
```

`bezeichner` ist der Name einer statischen Methode oder einer statischen Variablen.

## Mit der Klausel

```
import static paketname.Klassenname.*;
```

werden alle statischen Elemente der Klasse zugänglich gemacht.

Solche *statischen Importe* können auch für Elemente einer `enum`-Aufzählung genutzt werden.

Beispiel:

```
import static java.lang.Math.*;
import static java.lang.System.out;

public class Beispiel {
    public static void main(String[] args) {
        out.println("sin(PI/4) = " + sin(PI / 4));
    }
}
```

`sin` ist eine Klassenmethode und `PI` eine Klassenvariable in der Klasse `Math`, `out` ist eine Klassenvariable in der Klasse `System`.

## 7.2 Eigene Java-Bibliotheken erzeugen

Der Umgang mit Paketen wird anhand des folgenden Beispiels erläutert.

Die Klasse `Konto` entspricht der gleichnamigen Klasse aus Kapitel 6.

```
package bank;

public class Konto {
    private int kontonummer;
    private double saldo;

    public Konto() {
    }

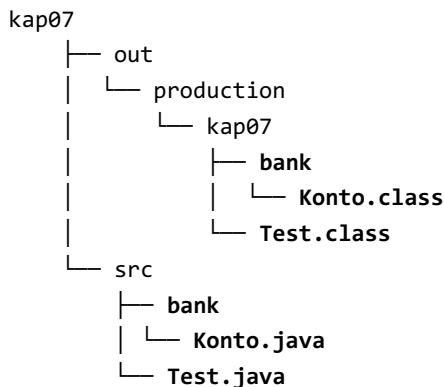
    ...
}

import bank.Konto;

public class Test {
    public static void main(String[] args) {
        Konto k = new Konto(1234, 1000.);
        k.info();
    }
}
```

Abbildung 7-1 zeigt, in welchen Verzeichnissen Quellcode und Bytecode abgelegt sind. Wir folgen hier der Ablagestruktur der Entwicklungsumgebung *IntelliJ IDEA*.

Um genau verfolgen zu können, was beim Compilieren geschieht, werden die einzelnen Schritte manuell im Terminal (beispielhaft für Linux) ausgeführt. Compilieren und Ausführen erfolgen im Verzeichnis *kap07*.



**Abbildung 7-1:** Ablagestruktur

### Schritt 1: Compilieren

```
javac -d out/production/kap07 src/bank/Konto.java
```

Mit der Option `-d` wird das Zielverzeichnis für die Ablage des Bytecode festgelegt.

Die Klasse `Test` importiert die Klasse `Konto` aus dem Paket `bank`. Deshalb muss der Klassenpfad angegeben werden, der den Suchpfad für `bank` angibt.

```
export CLASSPATH=out/production/kap07  
javac -d out/production/kap07 src/Test.java
```

Alternativ kann auch in einer Zeile eingegeben werden:

```
javac -d out/production/kap07 -cp out/production/kap07 src/Test.java
```

Das Setzen von `CLASSPATH` entfällt in diesem Fall.

### Schritt 2: Ausführen

```
export CLASSPATH=out/production/kap07  
java Test
```

oder

```
java -cp out/production/kap07 Test
```

## Java-Bibliotheken nutzen

Nach Schritt 1 kann auch eine Archivdatei im jar-Format (z. B. mit dem Namen `bank.jar`) erstellt werden, die alle Klassen eines oder mehrerer Pakete enthält:

```
jar --create --file bank.jar -C out/production/kap07 bank
```

Dann kann das Programm wie folgt aufgerufen werden:

```
java -cp out/production/kap07:bank.jar Test
```

Das Verzeichnis `bank` wird für die Ausführung nicht benötigt.

Sollen alle jar-Dateien aus einem bestimmten Verzeichnis (hier `lib`) verwendet werden, kann auch \* zum Compilieren bzw. Ausführen benutzt werden.

Beispiel:

```
java -cp .:lib/* MeineKlasse
```

## Pakete, Bibliotheken und Module

Wir wollen zum Schluss dieses Kapitels den Unterschied zwischen *Paketen*, *Bibliotheken* und *Modulen* erläutern.

- Ein *Paket* fasst Klassen und Interfaces zu einer Einheit unter einem Namen (Paketname) zusammen.
- Eine *Bibliothek* stellt ein oder mehrere Pakete in Form von jar-Dateien, die Bytecodes enthalten, zur Verfügung.
- Ein *Modul* im Sinne des Java-Modulsystems besteht aus ein oder mehreren Paketen, beschreibt die Abhängigkeit zu anderen Modulen und legt fest, welche eigenen Pakete für andere Module verwendbar sind. Module werden im Kapitel 33 behandelt.

## 7.3 Aufgaben

### Aufgabe 1

Erstellen Sie das Paket `de.example.utils`, das die Klassen `Kalender` und `Datum` enthält. `Kalender` beinhaltet die Methoden `tage` und `schaltjahr` aus Aufgabe 16 in Kapitel 6 und `Datum` ist die gleichnamige Klasse aus Aufgabe 17 in Kapitel 6.

Erzeugen Sie aus dem Paket `de.example.utils` die Java-Bibliothek `kalender.jar` und testen Sie die Funktionalität.

*Lösung:* `kap07-Aufgaben`



# 8 Vererbung

Neue Klassen können auf Basis bereits vorhandener Klassen definiert werden. *Vererbung* ist der Mechanismus, der dies ermöglicht.

## Lernziele

In diesem Kapitel lernen Sie

- wie von bestehenden Klassen neue Klassen abgeleitet werden können,
- wie abstrakte Klassen eingesetzt werden und
- was man unter Polymorphie versteht.

## 8.1 Klassen erweitern

### extends

Um eine neue Klasse aus einer bestehenden abzuleiten, wird im Kopf der Klassenbeschreibung das Schlüsselwort `extends` zusammen mit dem Namen der Klasse, von der abgeleitet wird, verwendet.

Die neue Klasse wird als *Subklasse*, *Unterklasse*, *abgeleitete Klasse* oder auch als *erweiterte Klasse*, die andere als *Superklasse*, *Oberklasse* oder als *Basisklasse* bezeichnet.

Die Subklasse kann in ihrer Implementierung alle Attribute und Methoden ihrer Superklasse verwenden (mit Ausnahme der mit `private` gekennzeichneten Elemente), so als hätte sie diese selbst implementiert.

Objekte der Subklasse können direkt auf die so geerbten Elemente zugreifen, sofern es die Zugriffsrechte erlauben.

Konstruktoren werden nicht vererbt.

Die Subklasse kann eigene Attribute und Methoden besitzen. Sie kann aber auch geerbte Methoden der Superklasse überschreiben, d. h. neu implementieren.

### Überschreiben / Overriding

Beim *Überschreiben* (Overriding) einer Instanzmethode bleibt ihre Signatur (Name, Parameterliste) unverändert.

Der Rückgabetyp darf vom Rückgabetyp der überschriebenen Methode abgeleitet sein. Dies gilt für Referenztypen, einfache Datentypen und `void` müssen unverändert übernommen werden.

Die Anweisungen im Methodenkörper werden geändert, um so auf die Besonderheiten der Subklasse als Spezialisierung der Superklasse eingehen zu können.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_8](https://doi.org/10.1007/978-3-658-43574-5_8).

Optional kann der überschreibenden Methode die Annotation

```
@Override
```

vorangestellt werden. Dies ermöglicht dem Compiler zu prüfen, ob eine Methode der Superklasse nach den oben angegebenen Regeln überschrieben wird.

Kurze Erläuterung zu Annotationen:

*Annotationen* erlauben es, in einem Quellcode Anmerkungen zur Klasse, zu Attributen und zu Methoden einzubauen, die dann zur Laufzeit von geeigneten Tools ausgewertet werden können. Solche Anmerkungen beginnen im Code mit dem Zeichen @. Java-intern sind Annotationen eine Sonderform von Interfaces.

### Subklasse Girokonto

```
package konto;

public class Girokonto extends Konto {
    private double limit;

    public Girokonto(int kontonummer, double saldo, double limit) {
        super(kontonummer, saldo);
        this.limit = limit;
    }

    public void setLimit(double limit) {
        this.limit = limit;
    }

    @Override
    public void zahleAus(double betrag) {
        double saldo = getSaldo();
        if (betrag <= saldo + limit) {
            saldo -= betrag;
            setSaldo(saldo);
        }
    }

    @Override
    public void info() {
        super.info();
        System.out.println("Limit: " + limit);
    }
}
```

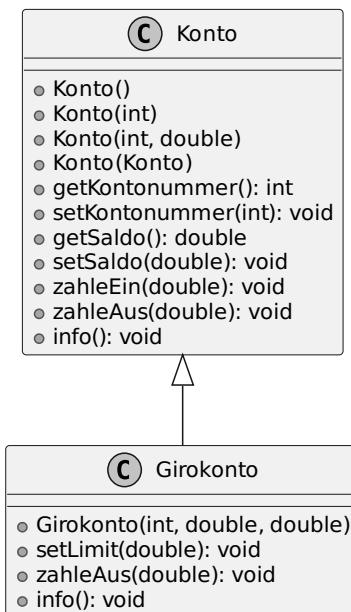
Hier erbt die Klasse Girokonto die Methoden der Klasse Konto (aus Kapitel 6). Da aber beim Girokonto noch der Überziehungskredit geprüft werden soll, wird die Methode zahleAus neu implementiert. Beim Aufruf der Methode zahleAus für ein Objekt vom Typ Girokonto wird nun die neu implementierte Methode der Klasse Girokonto verwendet.

Das Schlüsselwort `super` kann genutzt werden, um in einer überschreibenden Methode `m` der Subklasse (wie in `info` im obigen Beispiel) die ursprüngliche Methode `m` der Superklasse aufzurufen:

```
super.m(...);
```

Wird in einer abgeleiteten Klasse eine Variable definiert, die denselben Namen wie eine Variable in der Superklasse trägt, so wird letztere verdeckt. Der Zugriff innerhalb der abgeleiteten Klasse auf eine verdeckte Variable `x` ist dennoch mittels `super.x` möglich.

Objekte einer Subklasse enthalten zum einen die geerbten Variablen, zum andern die innerhalb der Subklasse definierten Variablen. Bei der Erzeugung eines Objekts der Subklasse müssen beide Arten von Variablen ggf. initialisiert werden.



**Abbildung 8-1:** Girokonto ist abgeleitet von Konto

### Die Klasse Object – Wurzel der Klassenhierarchie

Jede Klasse hat höchstens eine Superklasse (*Einfachvererbung*).

Eine Klasse, die nicht explizit von einer anderen Klasse erbt, ist automatisch von der in Java definierten Klasse `Object` abgeleitet. `Object` selbst besitzt keine Superklasse und ist damit die *Wurzel der Klassenhierarchie*.

## Arrays sind Objekte

Ein Array ist ein Objekt. Die Array-Variable ist eine Referenzvariable, die auf dieses Objekt zeigt. Der Array-Typ ist von der Klasse `Object` abgeleitet und erbt deren Methoden.

Ist die Klasse `B` von der Klasse `A` abgeleitet, so kann ein Array vom Typ `B` einer Array-Variablen vom Typ `A` zugewiesen werden, z. B.:

```
A[] array = new B[]{new B(...)};
```

## Keine Vererbung für Records

*Records* können nicht von einer Klasse abgeleitet werden. Klassen können nicht von *Records* erben.

## 8.2 Konstruktoren und Vererbung

Konstruktoren werden nicht vererbt und können demzufolge nicht überschrieben werden. Innerhalb eines Konstruktors der Subklasse kann alternativ ein anderer Konstruktor der eigenen Klasse mit `this(...)` oder ein Konstruktor der Superklasse mittels `super(...)` als *erste* Anweisung aufgerufen werden.

Beispiel:

```
public Girokonto(int kontonummer, double saldo, double limit) {
    super(kontonummer, saldo);
    this.limit = limit;
}
```

Im Beispiel wird die Initialisierung der Variablen aus `Konto` an den passenden Konstruktor der Superklasse `Konto` delegiert.

## Konstruktorregeln

Fehlt der Aufruf von `super(...)` in einem Konstruktor der Subklasse und wird auch kein anderer Konstruktor der gleichen Subklasse mit `this(...)` aufgerufen, so setzt der Compiler automatisch den Aufruf `super()` ein. Fehlt dann der parameterlose Konstruktor der Superklasse, erzeugt der Compiler einen Fehler. Besitzt die Subklasse überhaupt keinen expliziten Konstruktor, so erzeugt der Compiler automatisch einen parameterlosen Konstruktor, der lediglich den Aufruf des parameterlosen Konstruktors der Superklasse enthält.

Zum folgenden Programm gehören

- `Konto.java` (aus Kapitel 6),
- `Girokonto.java` (siehe oben) und
- `GirokontoTest.java`.

Alle Klassen liegen im Paket `konto`.

```
package konto;

public class GirokontoTest {
    public static void main(String[] args) {
        Girokonto gk = new Girokonto(4711, 500., 2000.);
        gk.info();
        gk.zahleAus(3000.);
        gk.info();
        gk.setLimit(2500.);
        gk.zahleAus(3000.);
        gk.info();
    }
}
```

Ausgabe des Programms:

```
Kontonummer: 4711 Saldo: 500.0
Limit: 2000.0
Kontonummer: 4711 Saldo: 500.0
Limit: 2000.0
Kontonummer: 4711 Saldo: -2500.0
Limit: 2500.0
```

### Konstruktion eines Objekts – ein rekursiver Prozess

Das folgende Programm zeigt, dass die Konstruktion eines Objekts ein rekursiver Prozess ist. Ein Konstruktor wird in drei Phasen ausgeführt:

1. Aufruf des Konstruktors der Superklasse
2. Initialisierung der Variablen
3. Ausführung des Konstruktorrumpfs

Die Objekterzeugung erfolgt also von oben nach unten entlang des Vererbungspfads.

```
package konstruktion1;

public class A {
{
    System.out.println("Initialisierung A");
}

public A() {
    System.out.println("Konstruktorrumpf A");
}
}

package konstruktion1;

public class B extends A {
{
    System.out.println("Initialisierung B");
}
}
```

```

public B() {
    System.out.println("Konstruktorrumpf B");
}
}

package konstruktion1;

public class C extends B {
{
    System.out.println("Initialisierung C");
}

public C() {
    System.out.println("Konstruktorrumpf C");
}

public static void main(String[] args) {
    new C();
}
}

```

Ausgabe des Programms:

```

Initialisierung A
Konstruktorrumpf A
Initialisierung B
Konstruktorrumpf B
Initialisierung C
Konstruktorrumpf C

```

Konstruktoren sollten keine überschreibbaren Methoden aufrufen. Der Konstruktor der Superklasse wird vor dem Konstruktor der Subklasse ausgeführt (siehe oben). Somit wird die überschreibende Methode in der Subklasse aufgerufen, bevor der Konstruktor der Subklasse ausgeführt wurde. Das führt dazu, dass im folgenden Programm die Zahl 0 und nicht 4711 ausgegeben wird.

```

package konstruktion2;

public class Super {
    public Super() {
        doSomething();
    }

    public void doSomething() {
    }
}

package konstruktion2;

public class Sub extends Super {
    private final int number;

```

```

public Sub() {
    number = 4711;
}

@Override
public void doSomething() {
    System.out.println(number);
}

public static void main(String[] args) {
    Sub sub = new Sub();
}
}

```

## 8.3 Methodenauswahl zur Laufzeit

### Zuweisungskompatibilität – Upcast und Downcast

Eine Referenzvariable vom Typ einer Klasse kann jeder Variablen vom Typ einer in der Vererbungshierarchie übergeordneten Klasse zugewiesen werden (*Upcast*).

Ein Objekt einer Subklasse ist insbesondere vom Typ der Superklasse. Überall, wo der Typ der Superklasse zulässig ist, ist auch der Typ einer Subklasse erlaubt (*Liskovsches Substitutionsprinzip*).

```

package cast;

import konto.Girokonto;
import konto.Konto;

public class Cast {
    public static void main(String[] args) {
        // Upcast
        Konto konto = new Girokonto(1020, 800., 2000.);
        konto.zahleAus(3000.);
        System.out.println(konto.getSaldo());

        // Downcast
        ((Girokonto) konto).setLimit(2500.);
        konto.zahleAus(3000.);
        System.out.println(konto.getSaldo());
    }
}

```

Ausgabe des Programms:

```

800.0
-2200.0

```

Das Programm zeigt, dass die Variable `konto` vom Typ `Konto` auf ein Objekt der Subklasse `Girokonto` verweisen kann. Trotzdem wird mit `konto.zahleAus(...)` die Methode `zahleAus` der Klasse `Girokonto` und nicht die Methode `zahleAus` der Klasse `Konto` aufgerufen. Der Betrag 3000 wird nicht vom Saldo abgezogen.

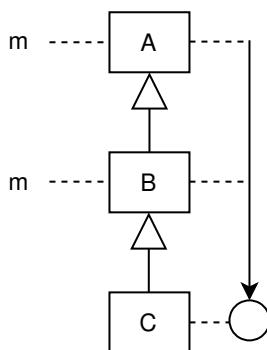
Bei der Ausführung bestimmt nicht der Typ der Referenzvariablen, sondern die *tatsächliche Klasse des Objekts*, welche Implementierung verwendet wird.

Um auf die Methode `setLimit` der Klasse `Girokonto` zugreifen zu können, ist eine explizite Typumwandlung (*Downcast*) erforderlich, da `setLimit` in der Klasse `Konto` nicht existiert.

### Auswahl der richtigen Methode

Allgemein gilt (vgl. Abbildung 8-2):

Um die auszuführende Methode (`m`) zu bestimmen, wird zur Laufzeit ausgehend von der Klasse (`A`), mit deren Typ die Referenzvariable definiert ist, entlang der Vererbungshierarchie in Richtung der Subklasse (`C`) des referenzierten Objekts die letzte überschreibende Methode `m` gesucht.



**Abbildung 8-2:** Ermittlung der auszuführenden Methode

## 8.4 Pattern Matching

Der Operator `instanceof` kann verwendet werden, um festzustellen, zu welcher Klasse ein bestimmtes Objekt gehört:

*Referenzvariable instanceof Klasse*

Dieser Ausdruck hat den Wert `true`, wenn die Referenzvariable auf ein Objekt der Klasse `Klasse` oder auf ein Objekt einer ihrer Subklassen verweist.

Beispiel:

```
Girokonto k = new Girokonto(1020, 800., 2000.);
```

Die Ausdrücke

```
(k instanceof Konto) und (k instanceof Girokonto)
```

haben beide den Wert `true`.

Mit dem ab Java 16 eingeführten *Pattern Matching* für instanceof lassen sich Casts vermeiden und ggf. Zeilen Quellcode einsparen:

*Referenzvariable instanceof Klasse variable*

```
package pattern_matching;

public class PatternMatchingDemo1 {
    private static Object[] objects() {
        return new Object[]{"Hallo", 12};
    }

    public static void main(String[] args) {
        for (Object o : objects()) {
            if (o instanceof String) {
                String s = (String) o;
                if (s.length() > 0) {
                    System.out.println(s.toUpperCase());
                }
            } else if (o instanceof Integer) {
                int i = (Integer) o;
                System.out.println(i * i);
            }
            // ab Java 16
            if (o instanceof String s && s.length() > 0) {
                System.out.println(s.toUpperCase());
            } else if (o instanceof Integer i) {
                System.out.println(i * i);
            }
        }
    }
}
```

Nach dem Typ bei instanceof steht eine Variable dieses Typs, die man im Folgenden sofort verwenden kann. Der Zugriff auf diese Variable ist auch bereits im if-Ausdruck möglich.

Ab Java 21 ist *Pattern Matching* auch bei switch möglich. Hier können Bedingungen statt mit && nun auch mit when formuliert werden.

```
package pattern_matching;

public class PatternMatchingDemo2 {
    private static Object[] objects() {
        return new Object[]{"Hallo", 12};
    }

    public static void main(String[] args) {
        for (Object o : objects()) {
            switch (o) {
                case String s when s.length() > 0 -> System.out.println(
                    s.toUpperCase());
                case Integer i -> System.out.println(i * i);
            }
        }
    }
}
```

```
        default -> System.out.println("Ungültiges Objekt");
    }
}
}
```

Mit *Record Patterns* können ab Java 21 Records in ihre Bestandteile zerlegt werden und damit das *Pattern Matching* bei `instanceof` und `switch` vereinfacht werden. Die beiden folgenden Programme verwenden den Record `Point`.

```
package pattern_matching;

public record Point(int x, int y) {
}

package pattern_matching;

public class PatternMatchingDemo3 {
    private static Object[] objects() {
        return new Object[]{new Point(1, 2), 12};
    }

    public static void main(String[] args) {
        for (Object o : objects()) {
            if (o instanceof Point(int x, int y)) {
                System.out.println("Point: " + x + ", " + y);
            } else if (o instanceof Integer i) {
                System.out.println(i * i);
            }
        }
    }
}

package pattern_matching;

public class PatternMatchingDemo4 {
    private static Object[] objects() {
        return new Object[]{new Point(1, 2), 12};
    }

    public static void main(String[] args) {
        for (Object o : objects()) {
            switch (o) {
                case Point(int x, int y) -> System.out.println(
                    "Point: " + x + ", " + y);
                case Integer i -> System.out.println(i * i);
                default -> System.out.println("Ungültiges Objekt");
            }
        }
    }
}
```

## 8.5 Abstrakte Klassen

Klassen, deren Definition mit dem Schlüsselwort `abstract` beginnt, heißen *abstrakte Klassen*. Von solchen Klassen können keine Objekte erzeugt werden. Sie können nur zur Vererbung genutzt werden.

In abstrakten Klassen dürfen sogenannte *abstrakte Methoden* auftreten. Abstrakte Methoden besitzen keinen Methodenrumpf. Sie müssen in Subklassen überschrieben werden, andernfalls sind die abgeleiteten Klassen ebenfalls abstrakt und müssen wiederum mit dem Schlüsselwort `abstract` gekennzeichnet werden.

Abstrakte Methoden dürfen nicht `private` sein, sonst könnten sie nicht überschrieben werden. Abstrakte Methoden können nicht statisch sein.

Abstrakte Klassen dienen als gemeinsame Superklasse innerhalb einer Klassenhierarchie. Sie sind insbesondere dann hilfreich, wenn sie bereits einen Teil der Implementierung selbst enthalten, es ihren Subklassen aber überlassen, spezielle Implementierungen der abstrakten Methoden bereitzustellen.

Den Nutzen *abstrakter Klassen* veranschaulicht das folgende Beispiel:

```
package abstrakt;

public abstract class Zeit {
    public abstract long getMinuten();

    public long getSekunden() {
        return getMinuten() * 60;
    }
}

package abstrakt;

public class Tage extends Zeit {
    private final long tage;

    public Tage(long tage) {
        this.tage = tage;
    }

    public long getMinuten() {
        return tage * 24 * 60;
    }
}

package abstrakt;

public class StundenMinuten extends Zeit {
    private final long stunden;
    private final long Minuten;

    public StundenMinuten(long stunden, long Minuten) {
        this.stunden = stunden;
    }
}
```

```

        this.minuten = minuten;
    }

    public long getMinuten() {
        return stunden * 60 + minuten;
    }
}

package abstrakt;

public class Test {
    public static void main(String[] args) {
        Zeit z = new Tage(3);
        System.out.println(z.getSekunden());

        z = new StundenMinuten(8, 30);
        System.out.println(z.getSekunden());
    }
}

```

Ausgabe des Programms:

```

259200
30600

```

Die abstrakte Klasse `Zeit` implementiert die Methode `getSekunden` und bedient sich dabei der abstrakten Methode `getMinuten` (sozusagen als Platzhalter). Die von der abstrakten Klasse `Zeit` abgeleiteten Klassen `Tage` und `StundenMinuten` überschreiben jeweils die abstrakte Methode `getMinuten` durch eine konkrete Implementierung. Beim Aufruf der Methode `getSekunden` in der Klasse `Test` wird nun die zum Typ des Objekts passende Implementierung von `getMinuten` ausgeführt.

## Polymorphie

Die Eigenschaft, dass der formal gleiche Aufruf einer Methode (wie in `Test`) unterschiedliche Reaktionen (je nach adressiertem Objekt) auslösen kann, wird als *Polymorphie* (Vielgestaltigkeit) bezeichnet.

Die Variable `z` im Beispiel zeigt im Verlauf des Tests auf Objekte verschiedener Klassen. Zunächst wird die `getSekunden`-Methode für ein `Tage`-Objekt aufgerufen, dann für ein `StundenMinuten`-Objekt. Dasselbe Programmkonstrukt ist also für Objekte mehrerer Typen einsetzbar.

Eine Referenzvariable kann zur Laufzeit auf Objekte verschiedenen Typs zeigen. Das System erkennt, zu welcher Klasse ein Objekt gehört, und ruft entsprechend die "richtige" Methode auf.

Polymorphie setzt voraus, dass vor der Ausführung der Methode eine Verbindung zwischen dem Methodenaufruf und der zum konkreten Objekt passenden Methode hergestellt wird. Der Compiler muss Code generieren, um zur Laufzeit zu entscheiden, welche Implementierung der polymorphen Methoden ausgeführt

werden soll (*dynamisches Binden, spätes Binden*). Der Einsatz dieses Mechanismus vereinfacht die Programmierung wesentlich.

## 8.6 Modifizierer

Mit Hilfe von sogenannten *Modifizierern* können bestimmte Eigenschaften von Klassen, Attributen und Methoden festgelegt werden. Die Tabelle 8-1 führt die Modifizierer auf und zeigt, bei welchen Sprachelementen diese verwendet werden können.

### Zugriffsrechte

Zugriffsrechte für Klassen, Attribute, Methoden und Konstruktoren werden über die Modifizierer `public`, `protected` und `private` gesteuert.

Sind weder `public`, `protected` noch `private` angegeben, so können die Klassen, Attribute, Methoden und Konstruktoren in allen Klassen *dieselben* Pakets benutzt werden.

**Tabelle 8-1:** Modifizierer im Überblick<sup>1</sup>

Modifizierer	Klasse	Attribut	Methode	Konstruktor
<code>public</code>	x	x	x	x
<code>protected</code>		x	x	x
<code>private</code>		x	x	x
<code>static</code>		x	x	
<code>final</code>	x	x	x	
<code>abstract</code>	x		x	
<code>native</code>			x	
<code>synchronized</code>			x	
<code>transient</code>		x		
<code>volatile</code>		x		

### `public, protected, private`

Klassen, die mit `public` gekennzeichnet sind, können überall (auch in anderen Paketen) genutzt werden.<sup>2</sup> In einer Quelldatei darf höchstens eine Klasse als `public`

1 `static` und `abstract` wurden bereits erläutert. `synchronized` und `volatile` werden im Zusammenhang mit Threads behandelt. `transient` wird im Zusammenhang mit der Serialisierung von Objekten erläutert.

2 Das ändert sich bei Verwendung von Modulen des Java-Modulsystems.

deklariert werden. Auf Attribute, Methoden und Konstruktoren, die als `public` vereinbart sind, kann überall dort zugegriffen werden, wo auf die Klasse zugegriffen werden kann.

Auf Attribute, Methoden und Konstruktoren, die mit `protected` gekennzeichnet sind, können die eigene Klasse und alle anderen Klassen innerhalb desselben Pakets zugreifen. Derartige Attribute und Methoden werden an alle Subklassen, die auch anderen Paketen angehören können, weitervererbt und sind dann dort zugänglich.

Beim Überschreiben von Methoden in Subklassen dürfen die Zugriffsrechte nicht reduziert werden. Insbesondere kann also eine `public`-Methode nur `public` überschrieben werden. Eine `protected`-Methode kann mit `public` oder `protected` überschrieben werden.

Attribute, Methoden und Konstruktoren, die mit `private` gekennzeichnet sind, können nur in der eigenen Klasse genutzt werden. Attribute sollten in der Regel immer als `private` deklariert und über Methoden zugänglich gemacht werden (Prinzip der *Datenkapselung*).

Will man die Objekterzeugung für eine Klasse außerhalb dieser Klasse verhindern, genügt es, nur einen einzigen parameterlosen Konstruktor mit dem Modifizierer `private` zu definieren.

### **final**

Aus einer mit `final` gekennzeichneten Klasse sind keine Subklassen ableitbar.

Attribute mit dem Modifizierer `final` sind nicht veränderbar. Solche *Konstanten* müssen bei ihrer Definition oder in einem Initialisierungsblock initialisiert werden oder es muss ihnen in jedem Konstruktor ein Wert zugewiesen werden. Eine `final`-Variable vom Referenztyp referenziert also immer dasselbe Objekt. Der Zustand dieses Objekts kann aber ggf. über Methodenaufrufe verändert werden.

Eine `final`-Methode kann in Subklassen nicht überschrieben werden.

### **native**

Methoden, die mit `native` gekennzeichnet sind, werden wie abstrakte Methoden ohne Anweisungsblock definiert. Die Implementierung erfolgt extern in einer anderen Programmiersprache (*Java Native Interface*).

Mehrere Modifizierer können auch in Kombination auftreten. Es bestehen aber die folgenden Einschränkungen:

- Ein Attribut kann nicht gleichzeitig `final` und `volatile` sein.
- Eine abstrakte Methode kann nicht `static`, `final`, `synchronized` oder `native` sein.

## 8.7 Versiegelte Klassen

Mit `final` markierte Klassen bilden die Endpunkte einer Vererbungshierarchie. Ab Java 17 kann mit `sealed` ebenfalls eine Ableitung verboten werden, Ausnahmen sind aber erlaubt:

```
sealed class A permits B, C, D { ... }
```

Nur die hinter `permits` aufgeführten Klassen können von `A` ableiten.

Ein Klasse (wie `D`), die eine *versiegelte Klasse* `A` erweitert, kann mit `non-sealed` markiert werden und damit durch beliebige Klassen erweitert werden.

Die Möglichkeit, Ableitungen mit `sealed` einzuschränken, ist insbesondere für Entwickler von Bibliotheken sinnvoll, die die Kontrolle über mögliche Implementierungen behalten wollen.

Versiegelung kann auch für Interfaces verwendet werden.

## 8.8 Aufgaben

### Aufgabe 1

Was versteht man unter der *Signatur* einer Methode und worin besteht der Unterschied zwischen dem *Überladen* und dem *Überschreiben* einer Methode?

Lösung: *Signatur.txt*

### Aufgabe 2

Sind folgende Anweisungen korrekt? `K2` sei hier Subklasse von `K1`.

```
K1 p1 = new K1();
K2 p2 = new K2();
p1 = p2;
p2 = (K2) p1;
```

Lösung: *Upcast-Downcast.txt*

### Aufgabe 3

Erstellen Sie die abstrakte Klasse `Mitarbeiter` und leiten Sie davon die Klassen `Angestellter` und `Azubi` ab.

Vorgaben für die Definition der drei Klassen:

Abstrakte Klasse `Mitarbeiter`:

```
protected String nachname;
protected String vorname;
protected double gehalt;
```

```
public Mitarbeiter(String nachname, String vorname, double gehalt)
```

```
// Erhöhung des Gehalts um betrag
public void erhoeheGehalt(double betrag)

// Ausgabe aller Variableninhalte
public void zeigeDaten()

// Gehalt durch Zulage erhöhen
public abstract void addZulage(double betrag)
```

Klasse Azubi:

```
private int abgelegtePruefungen;

public Azubi(String nachname, String vorname, double gehalt)

// Zahl der abgelegten Prüfungen setzen
public void setPruefungen(int anzahl)

// Ausgabe aller Variableninhalte
public void zeigeDaten()
```

Implementierung der Methode addZulage:

```
if (abgelegtePruefungen > 3)
    Gehaltserhöhung
```

Klasse Angestellter:

```
private static final int MAX_STUFE = 5;
private int stufe;

public Angestellter(String nachname, String vorname, double gehalt)

// Stufe um 1 erhöhen
public void befoerdere()

// Ausgabe aller Variableninhalte
public void zeigeDaten()
```

Implementierung der Methode addZulage:

```
if (stufe > 1)
    Gehaltserhöhung
```

Testen Sie alle Methoden.

*Lösung: Paket mitarbeiter*

#### Aufgabe 4

Erstellen Sie eine Klasse Figur mit den abstrakten Methoden:

```
public abstract void zeichne();
public abstract double getFlaeche();
```

Die Klassen Kreis und Rechteck sind von Figur abgeleitet und implementieren die beiden Methoden zeichne und getFlaeche.

Schreiben Sie ein Testprogramm, das die Methoden für verschiedene Figuren testet. Nehmen Sie hierzu die Figuren in einem Array vom Typ Figur auf.

*Lösung: Paket figur*

### Aufgabe 5

Ein Obstlager kann verschiedene Obstsorten (Apfel, Birne, Orange) aufnehmen. Die abstrakte Klasse Obst soll die folgenden abstrakten Methoden enthalten:

```
abstract String getName()  
abstract String getFarbe()
```

Die Klassen Apfel, Birne und Orange sind von Obst abgeleitet. Die Klasse Obstlager enthält ein Array vom Typ Obst. Die Methode void print() soll für alle Obstsorten im Lager die Methoden getName und getFarbe aufrufen.

*Lösung: Paket obstlager*

### Aufgabe 6

Implementieren Sie die Klasse Rechteck mit den beiden Attributen breite und hoehe, einem Konstruktor, den entsprechenden get- und set-Methoden und der Methode getFlaeche, die den Flächeninhalt ermittelt.

Leiten Sie von dieser Klasse die Klasse Quadrat ab, die das Attribut laenge enthält, einen Konstruktor und die beiden Methoden getLaenge und setLaenge. Ein Quadrat ist ein spezielles Rechteck. Zeigen Sie, dass jedoch einige Methoden der Superklasse, die Eigenschaft der Subklasse, Quadrat zu sein (Breite = Höhe), zerstören können.

Wie kann das Problem gelöst werden? Nicht immer ist Vererbung die richtige Strategie zur Wiederverwendung von Code. Später sehen Sie, dass Delegation die bessere Vorgehensweise ist.

*Lösung: Paket quadrat*

### Aufgabe 7

Warum terminiert der Aufruf von main in der unten aufgeführten Klasse B mit dem Fehler NullPointerException?

```
public class A {  
    public A() {  
        m();  
    }  
  
    public void m() {  
        System.out.println("m aus A");  
    }  
}
```

```
public class B extends A {  
    private A a;  
  
    public B() {  
        a = new A();  
        m();  
    }  
  
    public void m() {  
        a.m();  
        System.out.println("m aus B");  
    }  
  
    public static void main(String[] args) {  
        new B();  
    }  
}
```

*Lösung: Paket konstruktion*



# 9 Interfaces

*Interfaces* sind Schnittstellenbeschreibungen, die festlegen, was man mit der Schnittstelle machen kann. Dabei handelt es sich hauptsächlich um abstrakte Methoden, die in Klassen implementiert werden können. Objekte von Klassen, die dieselbe Schnittstelle implementieren, können einheitlich verarbeitet werden.

## Lernziele

In diesem Kapitel lernen Sie

- wie Interfaces zur Trennung von Schnittstelle und Implementierung verwendet werden,
- welche Besonderheiten bei Interfaces zu beachten sind und
- wie sich Interfaces von abstrakten Klassen abgrenzen.

## 9.1 Interfaces definieren und einsetzen

Neben Konstanten und abstrakten Methoden können Interfaces auch sogenannte Default-Methoden sowie statische und private Methoden enthalten.

Sie haben den folgenden Aufbau:

```
[public] interface Interface-Name [extends Interface-Liste] {  
    Konstanten  
    abstrakte Methoden  
  
    Default-Methoden  
    statische Methoden  
    private Methoden  
}
```

Die in eckigen Klammern angegebenen Einheiten sind optional. Namen für Interfaces unterliegen den gleichen Namenskonventionen wie Klassennamen.

Die Konstanten haben implizit die Modifizierer `public`, `static` und `final`.

Methoden, die ohne Modifizierer und ohne Methodenrumpf codiert sind, sind implizit `public` und `abstract`. Diese automatisch vergebenen Modifizierer sollten nicht explizit gesetzt werden.

Default-Methoden, statische und private Methoden sind erst ab der Java-Version 8 bzw. 9 hinzugekommen. Sie werden im Laufe dieses Kapitels behandelt.

### extends

Interfaces können im Gegensatz zu Klassen von *mehreren* anderen Interfaces abgeleitet werden. Damit können mehrere Schnittstellenbeschreibungen in einem Inter-

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_9](https://doi.org/10.1007/978-3-658-43574-5_9).

face zusammengefasst und neue Konstanten und Methoden hinzugefügt werden. Die Interface-Namen werden nach `extends`, durch Kommas voneinander getrennt, aufgeführt. Ein Interface "erbt" alle Konstanten und alle Methoden seiner Super-Interfaces.

## Interfaces implementieren

Klassen können ein oder mehrere Interfaces mit Hilfe des Schlüsselwortes `implements` implementieren. Mehrere Interface-Namen werden durch Kommas voneinander getrennt. Jede nicht abstrakte Klasse, die ein Interface implementiert, muss *alle* abstrakten Methoden des Interface implementieren. Dabei kann die Implementierung einer Interface-Methode auch von einer Superklasse geerbt werden.

Bei der Implementierung einer Interface-Methode müssen Signatur und Rückgabetypr in der Regel beibehalten werden. Es besteht aber folgende Ausnahme:

Im Falle einer Klasse als Rückgabetypr einer Methode des Interface darf der Rückgabetypr der implementierten Methode eine Subklasse dieser Klasse sein. Dies gilt analog auch für Interfaces als Rückgabetypr. Hier darf dann der Rückgabetypr ein Subinterface sein.

### *Behandlung von Namenskonflikten:*

Haben verschiedene Konstanten der implementierten Interfaces den gleichen Namen, so muss der Zugriff auf diese durch Qualifizierung mit dem betreffenden Interface-Namen erfolgen:

`Interface-Name.Konstantenname`

Die Implementierung zweier Interfaces, die Methoden mit gleicher Signatur, aber unterschiedlichen Rückgabetypen haben, ist nicht möglich.

Optional kann der implementierenden Methode die Annotation `@Override` (analog zum Überschreiben bei Vererbung) vorangestellt werden.

Auch *Records* können Interfaces implementieren.

## Wesentlicher Nutzen

Interfaces erzwingen, dass verschiedene Klassen, die keine Gemeinsamkeiten haben müssen, die gleichen Methoden bereitstellen, ohne dass eine abstrakte Superklasse vereinbart werden muss.

## Das Interface als Typ

Ein Interface ist ein *Referenztyp*. Variablen können vom Typ eines Interface sein. Ein Objekt einer Klasse, die ein Interface implementiert, ist auch vom Typ dieses Interface.

Eine Variable vom Typ des Interface x kann auf ein Objekt verweisen, dessen Klasse das Interface x implementiert. Mit Hilfe dieser Referenzvariablen kann allerdings nur auf die in x angegebenen Methoden der Klasse zugegriffen werden. Alle anderen Methoden können durch explizite Abwärtskonvertierung (*Downcast*) zugänglich gemacht werden.

Der Operator `instanceof` kann auch für Interfaces anstelle von Klassen verwendet werden.

Das folgende Programmbeispiel zeigt diesen Sachverhalt:

```
package interface_als_typ;

public interface X {
    void method();
}

package interface_als_typ;

public class A implements X {
    @Override
    public void method() {
        System.out.println("method");
    }

    public void doSomething() {
        System.out.println("doSomething");
    }
}

package interface_als_typ;

public class Test {
    public static void main(String[] args) {
        X x = new A();

        x.method();
        ((A) x).doSomething();

        System.out.println(x instanceof A);
        System.out.println(x instanceof X);
    }
}
```

Ausgabe des Programms:

```
method
doSomething
true
true
```

Der Zugriff auf Objekte mit Referenzvariablen vom Interface-Typ erleichtert den Austausch einer implementierenden Klasse gegen eine andere, die dasselbe Interface implementiert.

Das folgende Programm zeigt die Mächtigkeit des Interface-Konzepts. Die Klassen `Rechteck` und `Kreis` implementieren die im Interface `Geo` vorgegebene Methode `getFlaeche`. Die Methode `vergleiche` der Klasse `GeoVergleich` vergleicht zwei beliebige Objekte vom Typ `Geo` anhand der Größe ihres Flächeninhalts miteinander. Die Methode muss nicht "wissen", ob es sich um ein Rechteck oder einen Kreis handelt. Sie ist also universell einsetzbar für alle Klassen, die das Interface `Geo` implementieren.

```
package geo;

public interface Geo {
    double getFlaeche();
}

package geo;

public class GeoVergleich {
    public static int vergleiche(Geo a, Geo b) {
        final double EPSILON = 0.001;
        double x = a.getFlaeche();
        double y = b.getFlaeche();

        if (x <= y - EPSILON)
            return -1;
        else if (x >= y + EPSILON)
            return 1;
        else
            return 0;
    }
}

package geo;

public class Rechteck implements Geo {
    private final double breite;
    private final double hoehe;

    public Rechteck(double breite, double hoehe) {
        this.breite = breite;
        this.hoehe = hoehe;
    }

    @Override
    public double getFlaeche() {
        return breite * hoehe;
    }
}
```

```
package geo;

public class Kreis implements Geo {
    private final double radius;
    private static final double PI = 3.14159;

    public Kreis(double radius) {
        this.radius = radius;
    }

    @Override
    public double getFlaeche() {
        return PI * radius * radius;
    }
}

package geo;

public class Test {
    public static void main(String[] args) {
        Rechteck r = new Rechteck(10.5, 4.799);
        Kreis k = new Kreis(4.0049);

        System.out.println(r.getFlaeche());
        System.out.println(k.getFlaeche());
        System.out.println(GeoVergleich.vergleiche(r, k));
    }
}
```

Ausgabe des Programms:

```
50.389500000000005
50.38866575757591
0
```

Es können nach Bedarf weitere Klassen, die das Interface `Geo` implementieren, entwickelt werden. Die Methode `vergleiche` "funktioniert" auch für Objekte dieser neuen Klassen.

## Vererbung vs. Interfaces

Polymorphie lässt sich auch mittels Interfaces realisieren.

Eine Klasse, die ein Interface implementiert, "erbt" nur die Methodensignatur (inkl. Rückgabetyp) und ist typkompatibel zum Interface. In einem Programm kann die Klasse leicht gegen eine andere, die dasselbe Interface implementiert, ausgetauscht werden. Die Klasse ist also nur lose an das Interface gekoppelt.

Im Gegensatz dazu ist bei Vererbung eine Subklasse eng an ihre Superklasse gekoppelt. Die Subklasse ist von der Implementierung der Superklasse abhängig. Wird der Code der Superklasse geändert, kann das gravierende Auswirkungen auf alle Subklassen haben. Große Vererbungshierarchien sind schlecht zu warten.

Hier ist abzuwägen:

- Trennung von Schnittstelle und Implementierung (bei Verwendung von Interfaces) oder
- weniger Redundanz, aber größere Abhängigkeit (bei Vererbung).

### Geschachtelte Interfaces

Interfaces können innerhalb einer Klasse oder eines anderen Interface definiert werden. Von außen kann dann voll qualifiziert auf sie zugegriffen werden:

*Typname.Interface-Name*

```
package geschachtelt;

public interface X {
    void x();

    interface Y {
        void y();
    }
}

package geschachtelt;

public class A implements X, X.Y {
    public interface Z {
        void z();
    }

    public void x() {
        System.out.println("Methode x");
    }

    public void y() {
        System.out.println("Methode y");
    }
}

package geschachtelt;

public class B implements A.Z {
    public void z() {
        System.out.println("Methode z");
    }

    public static void main(String[] args) {
        X a1 = new A();
        a1.x();

        X.Y a2 = new A();
        a2.y();
    }
}
```

```
A.Z b = new B();  
b.z();  
}  
}
```

Interfaces können auch lokal innerhalb einer Methode definiert werden.

## 9.2 Default-Methoden

Interfaces können neben abstrakten Methoden auch sogenannte *Default-Methoden* enthalten. Diese haben einen Methodenkörper mit Code und sind mit dem Schlüsselwort `default` gekennzeichnet. Die Implementierung wird an alle abgeleiteten Interfaces und Klassen "vererbt", sofern diese sie nicht mit einer eigenen Implementierung überschreiben.

Die folgenden Programmbeispiele demonstrieren verschiedene Aspekte hierzu.

### Beispiel 1

Default-Methoden können abstrakte Methoden des Interface verwenden. Klassen, die ein Interface implementieren, müssen wie bisher alle abstrakten Methoden implementieren, können die Default-Methoden aufrufen oder diese selbst neu implementieren.

```
package default1;  
  
public interface X {  
    void m1();  
  
    default void m2() {  
        System.out.println("Methode m2");  
    }  
  
    default void m3() {  
        System.out.println("Methode m3");  
        m1();  
    }  
}  
  
package default1;  
  
public class XImpl1 implements X {  
    @Override  
    public void m1() {  
        System.out.println("Methode m1");  
    }  
  
    public static void main(String[] args) {  
        X x = new XImpl1();  
    }  
}
```

```
        x.m2();
        x.m3();
    }
}

package default1;

public class XImpl2 implements X {
    @Override
    public void m1() {
        System.out.println("Methode m1");
    }

    @Override
    public void m2() {
        System.out.println("Methode m2 neu implementiert");
    }

    public static void main(String[] args) {
        X x = new XImpl2();
        x.m2();
        x.m3();
    }
}
```

## Beispiel 2

Default-Methoden konkurrieren mit geerbten Methoden.

```
package default2;

public interface X {
    default void m() {
        System.out.println("Methode m aus X");
    }
}

package default2;

public class A {
    public void m() {
        System.out.println("Methode m aus A");
    }
}

package default2;

public class B extends A implements X {
    public static void main(String[] args) {
        B b = new B();
        b.m();
    }
}
```

---

Ausgabe des Programms:

Methode m aus A

Die von A geerbte Methode hat Vorrang gegenüber der Default-Methode des implementierten Interface X.

*Fazit*

Sobald eine Methode von einer Klasse (Superklasse in der Vererbungshierarchie) implementiert ist, werden entsprechende Default-Methoden ignoriert.

### Beispiel 3

Im folgenden Beispiel implementieren die Interfaces X und Y eine Default-Methode mit demselben Methodenkopf. Y ist von X abgeleitet. Die Klasse A implementiert das Interface Y.

```
package default3;

public interface X {
    default void m() {
        System.out.println("Methode m aus X");
    }
}

package default3;

public interface Y extends X {
    default void m() {
        System.out.println("Methode m aus Y");
    }
}

package default3;

public class A implements Y {
    public static void main(String[] args) {
        A a = new A();
        a.m();
    }
}
```

Ausgabe des Programms:

Methode m aus Y

*Fazit*

Spezifischere Interfaces "gewinnen" über weniger spezifische.

### Beispiel 4

Implementiert eine Klasse zwei Interfaces, die jeweils eine Default-Methode mit demselben Methodenkopf anbieten, so bricht der Compiler beim Versuch, den Methodenaufruf zu übersetzen, mit einem Fehler ab.

Der Konflikt kann dadurch gelöst werden, dass die Klasse diese Methode selbst implementiert, oder dass man sich für ein Interface entscheidet und statt einer eigenen Implementierung dessen Default-Methode aufruft. Hierzu wird der Interface-Name in Kombination mit `super` benutzt (siehe Kommentar im folgenden Quellcode).

```
package default4;

public interface X {
    default void m() {
        System.out.println("Methode m aus X");
    }
}

package default4;

public interface Y {
    default void m() {
        System.out.println("Methode m aus Y");
    }
}

package default4;

public class A implements X, Y {
    public void m() {
        System.out.println("Methode m aus A");
        // X.super.m();
    }

    public static void main(String[] args) {
        A a = new A();
        a.m();
    }
}
```

### Fazit

Im Konfliktfall muss die implementierende Klasse den Konflikt selbst lösen.

### Abstrakte Klassen vs. Interfaces

Sowohl Interfaces als auch abstrakte Klassen können abstrakte Methoden und implementierte Methoden enthalten.

Im Gegensatz zu Interfaces können abstrakte Klassen aber Konstruktoren und Instanzvariablen definieren.

Für die Implementierung von Default-Methoden in Interfaces können keine Instanzvariablen definiert werden. Natürlich kann die Funktionalität der anderen (abstrakten) Methoden des Interface genutzt werden.

Default-Methoden können bei der Weiterentwicklung von Software sehr hilfreich sein (API-Evolution), ein entscheidender Grund für ihre Einführung.

Hierzu das folgende Beispiel:

Die Klasse `LifecycleImpl` implementiert das Interface `Lifecycle`.

```
package lifecycle1;

public interface Lifecycle {
    void start();
    void stop();
}

package lifecycle1;

public class LifecycleImpl implements Lifecycle {
    public void start() {
        System.out.println("Start");
    }

    public void stop() {
        System.out.println("Stop");
    }
}
```

Bei der Weiterentwicklung möchte man das Interface `Lifecycle` um die neuen abstrakten Methoden `init` und `destroy` ergänzen. Das bedeutet aber, dass die Klasse `LifecycleImpl` ebenfalls angepasst werden muss und dass diese Klasse die neuen Methoden implementieren muss (z. B. durch einen leeren Methodenrumpf).

Werden die neuen Methoden aber als Default-Methoden im Interface `Lifecycle` definiert, so ist eine Anpassung der Klasse `LifecycleImpl` nicht nötig. Das zeigt das folgende Beispiel.

```
package lifecycle2;

public interface Lifecycle {
    void start();
    void stop();

    default void init() {
    }
}
```

```
default void destroy() {  
}  
}  
  
package lifecycle2;  
  
public class LifecycleImpl implements Lifecycle {  
    public void start() {  
        System.out.println("Start");  
    }  
  
    public void stop() {  
        System.out.println("Stop");  
    }  
}
```

### Fazit

Interfaces können problemlos weiterentwickelt werden. Betroffene Klassen müssen nicht angepasst werden. Die Default-Methoden gewähren somit die Abwärtskompatibilität.

## 9.3 Statische und private Methoden in Interfaces

Interfaces können auch `static`-Methoden implementieren. Solche Methoden sind wie bei allen anderen Interface-Methoden automatisch `public`.

`private`-Methoden sind in Interfaces ebenfalls erlaubt. Diese können nur in Default-Methoden verwendet werden. Nutzen mehrere Default-Methoden den gleichen Code-Teil, so kann dieser in einer `private`-Methode ausgelagert werden, womit man Redundanz vermeidet.

```
package static_private;  
  
public interface X {  
    default void m1() {  
        System.out.println("Methode m1 aus X");  
        m2();  
    }  
  
    private void m2() {  
        System.out.println("Methode m2 aus X");  
    }  
  
    static void m3() {  
        System.out.println("Methode m3 aus X");  
    }  
}  
  
package static_private;  
  
public class A {
```

```
public static void main(String[] args) {
    X.m3();
}

package static_private;

public class B implements X {
    public static void main(String[] args) {
        B b = new B();
        b.m1();

        // Folgende Aufrufe sind nicht möglich:
        // b.m2();
        // b.m3();
        // B.m3();
    }
}
```

Auch wenn die Klasse `B` das Interface `X` implementiert, kann (im Unterschied zum Verhalten von statischen Methoden bei der Vererbung von Klassen) die Methode `m` aus `X` nicht mit einer Referenz auf ein Objekt dieser Klasse oder mit dem Klassennamen selbst aufgerufen werden.

## 9.4 Aufgaben

### Aufgabe 1

Das Interface `Displayable` soll die abstrakte Methode `void display()` enthalten. Implementieren Sie für die Klasse `Sparbuch` aus Aufgabe 5 in Kapitel 6 dieses Interface. Die Methode `display` soll alle Attributwerte des jeweiligen Objekts ausgeben.

Definieren Sie dann die Klasse `Utilities`, die die Klassenmethode

```
public static void display(Displayable a)
```

enthält. Diese Methode soll nach Ausgabe einer laufenden Nummer, die bei jedem Aufruf um 1 erhöht wird, die `Displayable`-Methode `display` aufrufen.

*Lösung: Paket displayable*

### Aufgabe 2

Eine Liste ganzer Zahlen größer oder gleich 0 soll als Interface `IntegerList` mit den folgenden abstrakten Methoden definiert werden:

```
int getLength()
```

liefert die Länge der Liste.

```
void insertLast(int value)
```

fügt `value` am Ende der Liste ein.

```

int getFirst()
    liefert das erste Element der Liste.

void deleteFirst()
    löscht das erste Element der Liste.

boolean search(int value)
    prüft, ob value in der Liste vorhanden ist.

```

Implementieren Sie dieses Interface in der Klasse `ArrayList`. Verwenden Sie hierzu ein Array. Testen Sie alle Methoden.

*Lösung: Paket list*

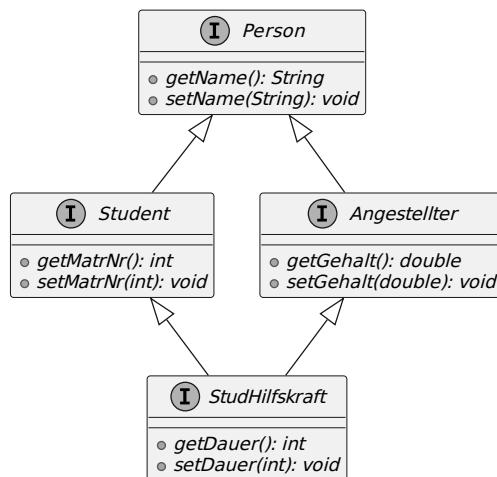
### Aufgabe 3

Lösen Sie Aufgabe 5 in Kapitel 8 mit Hilfe eines Interface anstelle der abstrakten Klasse `Obst`.

*Lösung: Paket obst*

### Aufgabe 4

Erstellen Sie die in der folgenden Abbildung dargestellten Interfaces sowie die Klasse `HiWi`, die das Interface `StudHilfskraft` implementiert.



*Lösung: Paket hiwi*

### Aufgabe 5

Das Interface `Rechteck` soll die abstrakten Methoden

```
int getBreite()
```

und

```
int getHoehe()
```

haben.

Es soll die Default-Methode

```
boolean isQuadrat()
```

und die statische Methode

```
static int compare(Rechteck a, Rechteck b)
```

implementieren.

Letztere soll die Flächeninhalte zweier Rechtecke vergleichen und -1, 0 oder 1 zurückgeben, je nachdem die Flächen von a und b in einer Kleiner-, Gleich- bzw. Größer-Beziehung zueinander stehen. Erstellen Sie dieses Interface, eine Klasse, die `Rechteck` implementiert, sowie ein Testprogramm.

*Lösung: Paket rechteck*



# 10 Vererbung vs. Delegation

Es ist nicht immer sinnvoll, die Wiederverwendung von Code in Klassen durch Vererbung zu lösen. Insbesondere dann, wenn die Klasse, von der abgeleitet werden soll, nicht unter der Kontrolle desselben Programmierers ist, kann eine unbedarfte Nutzung sogar zu Fehlern führen. Die Subklasse hängt von den Implementierungsdetails der Superklasse ab. Diese muss man kennen, damit die Subklasse einwandfrei funktioniert.

## Lernziele

In diesem Kapitel lernen Sie

- warum Vererbung nicht immer die erste Wahl ist und
- wie die Wiederverwendung von Code sicher durch Delegation erfolgen kann.

## 10.1 Warum Vererbung problematisch sein kann

Wir beginnen mit einem Beispiel, das zeigt, wie die Erweiterung einer Klasse, deren Implementierung man nicht kennt, zu Fehlern führen kann.

Die Klasse, von der geerbt wird, soll eine Menge von Zahlen verwalten, die keine doppelten Einträge enthalten darf. Es kann eine Zahl hinzugefügt werden oder auch gleich ein Array von Zahlen. Hier folgt der Quellcode:

```
package set;

public class IntegerSet {
    private int[] set = new int[1000];
    private int size;

    public int[] getIntegers() {
        int[] result = new int[size];
        for (int i = 0; i < size; i++) {
            result[i] = set[i];
        }
        return result;
    }

    public void add(int n) {
        boolean found = false;
        for (int value : set) {
            if (value == n) {
                found = true;
                break;
            }
        }
    }
}
```

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_10](https://doi.org/10.1007/978-3-658-43574-5_10).

```

        if (!found) {
            set[size++] = n;
        }
    }

    public void addAll(int[] integers) {
        for (int i : integers) {
            add(i);
        }
    }

    public static void main(String[] args) {
        IntegerSet integerSet = new IntegerSet();
        integerSet.add(1);
        integerSet.add(2);
        integerSet.add(1);
        integerSet.addAll(new int[] {1, 2, 3, 4});

        int[] integers = integerSet.getIntegers();
        for (int i : integers) {
            System.out.println(i);
        }
    }
}

```

Ausgabe des Programms:

```

1
2
3
4

```

`IntegerSet` ist die Klasse, von der abgeleitet werden soll. Per Dokumentation sind nur die Aufrufsschnittstellen der Methoden `getIntegers`, `add` und `addAll` bekannt:

`public int[] getIntegers()`  
liefert die verwaltete Zahlenmenge als Array.

`public void add(int n)`  
fügt die Zahl `n` in die Menge ein.

`public void addAll(int[] integers)`  
fügt die Zahlen des Arrays `integers` in die Menge ein.

Die verwaltete Menge enthält nur Unikate, also keine doppelten Einträge.

Für das Weitere gehen wir nun davon aus, dass wir die Implementierungsdetails von `IntegerSet` nicht kennen.

Wir wollen `IntegerSet` um einen Zähler erweitern, der zählt, wie oft wir Zahlen in die Menge einzufügen versuchen, unabhängig davon, ob die Zahl aufgenommen wird oder abgelehnt wird, da sie sich bereits in der Menge befindet.

Das soll mittels Vererbung geschehen.

```
package v1;

import set.IntegerSet;

public class CountingSet extends IntegerSet {
    private long count;

    @Override
    public void add(int n) {
        count++;
        super.add(n);
    }

    @Override
    public void addAll(int[] integers) {
        count += integers.length;
        super.addAll(integers);
    }

    public long getCount() {
        return count;
    }

    public static void main(String[] args) {
        CountingSet set = new CountingSet();
        set.addAll(new int[] {1, 2, 3});
        System.out.println("count: " + set.getCount());
        int[] integers = set.getIntegers();
        for (int i : integers) {
            System.out.println(i);
        }
    }
}
```

Ausgabe des Programms:

```
count: 6
1
2
3
```

Wie ist das möglich, da wir doch genau drei verschiedene Zahlen eingefügt haben?  
Schauen wir uns die Implementierung von `IntegerSet` an, erkennen wir, dass die Methode `addAll` intern die Methode `add` aufruft. Es wird zur Laufzeit also die überschriebene Methode `add` verwendet (*dynamisches Binden*).  
Es wird also doppelt gezählt!

## 10.2 Delegation als Alternative

Unter *Delegation* versteht man die Weitergabe einer Aufgabe an ein Objekt einer Klasse, die auf die Ausführung dieser Aufgabe spezialisiert ist.

Bezogen auf unser Beispiel heißt das:

Anstatt, dass `IntegerSet` Superklasse für unsere eigene Klasse ist, soll sie nun die Klasse sein, die *eine Aufgabe übernimmt*.

```
package v2;

import set.IntegerSet;

public class CountingSet {
    private final IntegerSet delegate = new IntegerSet();
    private long count;

    public void add(int n) {
        count++;
        delegate.add(n);
    }

    public void addAll(int[] integers) {
        count += integers.length;
        delegate.addAll(integers);
    }

    public long getCount() {
        return count;
    }

    public int[] getIntegers() {
        return delegate.getIntegers();
    }

    public static void main(String[] args) {
        CountingSet set = new CountingSet();
        set.addAll(new int[] {1, 2, 3});
        System.out.println("count: " + set.getCount());
        int[] integers = set.getIntegers();
        for (int i : integers) {
            System.out.println(i);
        }
    }
}
```

Ausgabe des Programms:

```
count: 3
1
2
3
```

Auf diese Weise kann auch die Funktionalität von *mehreren* unterschiedlichen Klassen genutzt werden. Aufgrund der Einfachvererbung in Java ist Vererbung dann keine Lösungsalternative mehr.

## 10.3 Aufgaben

### Aufgabe 1

Lösen Sie das in Aufgabe 6 aus Kapitel 8 dargestellte Problem mit Hilfe von *Delegation*.

Anmerkung:

In diesem Beispiel delegieren wir an das Rechteck nur, um das Ergebnis der Flächenberechnung zu erhalten. Das kann man für ein Quadrat hier natürlich direkt in der Klasse Quadrat implementieren, ohne Delegation zu nutzen. Aber es geht hier um die Veranschaulichung des Prinzips.

*Lösung: Paket quadrat*

### Aufgabe 2

Erstellen Sie die Klasse Printer, die zur Laufzeit das Drucken an einen Schwarz-Weiß-Drucker (MonochromePrinter) oder einen Farb-Drucker (ColorPrinter) delegieren kann. Diese beiden Klassen sollen das Interface Printable mit der abstrakten Methode void print() implementieren.

Die Printer-Methode

```
public void switchTo(Printable printer)
```

wechselt den Drucker.

Testen Sie das Programm.

*Lösung: Paket drucker*



# 11 Geschachtelte Klassen

*Geschachtelte Klassen* sind Klassen, die *innerhalb* einer bestehenden Klasse definiert sind. Solche Klassen werden beispielsweise bei der Ereignisbehandlung im Rahmen der Entwicklung von grafischen Oberflächen verwendet. Zur Abgrenzung nennt man die "normalen" Klassen auch *Top-Level-Klassen*.

## Lernziele

In diesem Kapitel lernen Sie

- welche Arten von geschachtelten Klassen existieren und
- wie solche sinnvoll genutzt werden können.

Wir behandeln im Folgenden vier Arten von geschachtelten Klassen:

- statische Klassen,
- Instanzklassen,
- lokale Klassen und
- anonyme Klassen.

## 11.1 Statische Klasse

Eine *statische Klasse* B wird innerhalb einer Klasse A mit dem Schlüsselwort `static` definiert. Objekte von B können mit

```
A.B obj = new A.B(...);
```

erzeugt werden. A.B ist der vollständige Name der statischen Klasse B. Aus B heraus kann direkt auf Klassenvariablen und Klassenmethoden von A zugegriffen werden. B ist unabhängig von Instanzen der Klasse A.

Statische Klassen bieten die Möglichkeit, inhaltlich aufeinander bezogene Klassen im Zusammenhang in einer einzigen Datei zu definieren.

Statische Klassen werden wie Attribute vererbt. Sie können mit `public`, `protected` oder `private` in der üblichen Bedeutung gekennzeichnet werden.

Das folgende Programm veranschaulicht die Verwendung einer statischen Klasse. Die statische Klasse `Permissions` verwaltet die Zugriffsrechte eines Benutzers.

```
package statiche_klasse;

public class Account {
    private final int userId;
    private final Permissions perm;
```

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_11](https://doi.org/10.1007/978-3-658-43574-5_11).

```

public static class Permissions {
    public boolean canRead;
    public boolean canWrite;
    public boolean canDelete;

    public String info() {
        return "read: " + canRead + ", " + "write: " + canWrite + ", " +
               "delete: " + canDelete;
    }
}

public Account(int userId) {
    this.userId = userId;
    perm = new Permissions();
}

public int getUserId() {
    return userId;
}

public Permissions getPermissions() {
    return perm;
}
}

package statische_klasse;

public class Test {
    public static void main(String[] args) {
        Account account = new Account(4711);
        Account.Permissions perm = account.getPermissions();
        perm.canRead = true;
        perm.canWrite = true;
        System.out.println(account.getUserId());
        System.out.println(perm.info());
    }
}

```

Ausgabe des Programms:

```

4711
read: true, write: true, delete: false

```

## 11.2 Instanzklasse

Objekte von *Instanzklassen* können nur im Verbund mit einer Instanz der sie umgebenden Klasse entstehen. Sie werden in der Regel in Instanzmethoden der umgebenden Klasse erzeugt und haben Zugriff auf alle Attribute und Methoden dieser äußeren Klasse. Instanzklassen können ab Java 16 auch statische Klassen, statische Attribute oder statische Methoden enthalten.

Die Instanz der äußeren Klasse A kann mit A.this explizit referenziert werden. Instanzklassen werden wie Attribute vererbt. Sie können mit public, protected oder private in der üblichen Bedeutung gekennzeichnet werden.

Das nächste Programm veranschaulicht die Verwendung einer Instanzklasse. Die Klasse Konto verwaltet die letzte für das Konto durchgeführte Transaktion (Einzahlung oder Auszahlung) in einem Objekt einer Instanzklasse. Mit Hilfe der Punktnotation kann der Typname Transaktion außerhalb von Konto benutzt werden.

```
package instanz_klasse;

public class Konto {
    private final int kontonummer;
    private double saldo;
    private Transaktion last;

    public class Transaktion {
        private final String name;
        private final double betrag;

        public Transaktion(String name, double betrag) {
            this.name = name;
            this.betrag = betrag;
        }

        public String toString() {
            return kontonummer + ": " + name + " " + betrag + ", Saldo " + saldo;
        }
    }

    public Konto(int kontonummer, double saldo) {
        this.kontonummer = kontonummer;
        this.saldo = saldo;
    }

    public Transaktion getLast() {
        return last;
    }

    public void zahleEin(double betrag) {
        saldo += betrag;
        last = new Transaktion("Einzahlung", betrag);
    }

    public void zahleAus(double betrag) {
        saldo -= betrag;
        last = new Transaktion("Auszahlung", betrag);
    }
}

package instanz_klasse;

public class Test {
    public static void main(String[] args) {
        Konto k = new Konto(4711, 1000.);
        k.zahleEin(500.);
        k.zahleAus(700.);
        Konto.Transaktion t = k.getLast();
        System.out.println(t.toString());
    }
}
```

Ausgabe des Programms:

4711: Auszahlung 700.0, Saldo 800.0

Für ein Konto k kann auch eine neue Transaktion erzeugt werden:

```
k.new Transaktion(...);
```

Das macht allerdings in dieser Anwendung keinen Sinn.

### 11.3 Lokale Klasse

*Lokale Klassen* können in einem Methodenrumpf, einem Konstruktor oder einem Initialisierungsblock analog zu lokalen Variablen definiert werden.

Der Code in einer lokalen Klasse kann auf alle Attribute und Methoden der umgebenden Klasse zugreifen, sofern die lokale Klasse nicht innerhalb einer Klassenmethode oder eines statischen Initialisierungsblocks definiert ist.

Die Instanz der äußeren Klasse A kann mit `A.this` explizit referenziert werden.

Lokale Klassen können ab Java 16 auch statische Klassen, statische Attribute oder statische Methoden enthalten.

Der Code kann außerdem auf alle lokalen `final`-Variablen eines umgebenden Blocks und alle `final`-Parameter eines umgebenden Methodenrumpfs zugreifen.

Ab Java-Version 8 muss eine lokale Variable, die zur Laufzeit ihren Wert nicht ändert, nicht mehr explizit mit `final` gekennzeichnet werden (*effectively final*).

Das folgende Beispiel verwendet innerhalb der Methode `iterator` der Klasse `Liste` eine lokale Klasse. Die Klasse `Liste` verwaltet beliebige Objekte vom Typ `Object` in einer sogenannten *verketteten Liste*. Eine solche Liste besteht aus Elementen, die jeweils eine Referenz auf ein Objekt sowie eine Referenz auf das nächste Element der Liste enthalten.

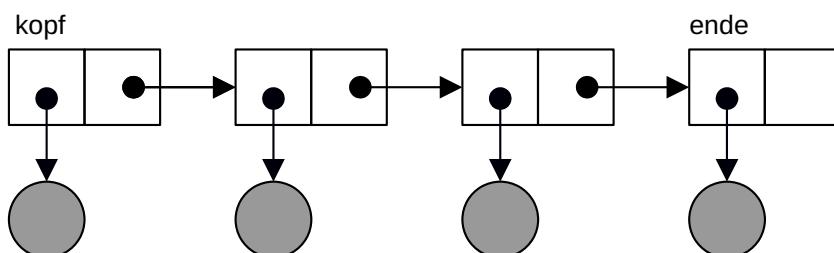


Abbildung 11-1: Verkettete Liste

```
package lokale_klasse;

public interface Iterator {
    boolean hasNext();
    Object next();
}

package lokale_klasse;

public class Liste {
    private Element kopf, ende;

    private static class Element {
        private Object obj;
        private Element next;
    }

    public void add(Object obj) {
        if (obj == null)
            return;
        Element neu = new Element();
        neu.obj = obj;
        if (kopf == null)
            kopf = ende = neu;
        else {
            ende.next = neu;
            ende = neu;
        }
    }

    public Iterator iterator() {
        class IteratorImpl implements Iterator {
            private Element e = kopf;

            public boolean hasNext() {
                return e != null;
            }

            public Object next() {
                if (e == null)
                    return null;
                Object obj = e.obj;
                e = e.next;
                return obj;
            }
        }
        return new IteratorImpl();
    }
}

package lokale_klasse;

public class Test {
    public static void main(String[] args) {
        Liste liste = new Liste();
        liste.add("Element 1");
        liste.add("Element 2");
    }
}
```

```

        liste.add("Element 3");

        Iterator it = liste.iterator();
        while (it.hasNext()) {
            String s = (String) it.next();
            System.out.println(s);
        }
    }
}

```

Ausgabe des Programms:

```

Element 1
Element 2
Element 3

```

Die Methode `iterator` liefert ein Objekt vom Typ des Interface `Iterator`, mit Hilfe dessen Methoden die verkettete Liste bequem durchlaufen werden kann. In Kapitel 20.1 wird das generische Interface `java.util.Iterator` vorgestellt.

## 11.4 Anonyme Klasse

Eine *anonyme Klasse* ist eine lokale Klasse, die ohne Klassennamen in einer `new`-Anweisung definiert wird. Klassendefinition und Objekterzeugung sind also in einer Anweisung zusammengefasst.

Die namenlose Klasse erweitert eine andere Klasse oder implementiert ein Interface, hat aber keine eigenen Konstruktoren.

```

new Konstruktor(...) { Klassenrumpf }
new Interface() { Klassenrumpf }

```

*Konstruktor* steht für einen Konstruktor der Basisklasse, *Interface* für das im Klassenrumpf implementierte Interface.

Der Rückgabewert ist vom Typ der Klasse des Konstruktors bzw. vom Typ des Interface.

Anonyme Klassen können ab Java 16 auch statische Klassen, statische Attribute oder statische Methoden enthalten.

Aus dem Klassenrumpf kann auf Attribute und Methoden der umgebenden Klasse sowie auf alle lokalen `final`-Variablen eines umgebenden Blocks bzw. Methodenrumpfs zugegriffen werden.

Ab Java-Version 8 muss eine lokale Variable, die zur Laufzeit ihren Wert nicht ändert, nicht mehr explizit mit `final` gekennzeichnet werden (*effectively final*).

Die Instanz der äußeren Klasse A kann mit `A.this` explizit referenziert werden. Alle abstrakten Methoden der Basisklasse bzw. des Interface müssen implementiert werden.

Anstelle der lokalen Klasse im letzten Programm kann eine anonyme Klasse verwendet werden.

```
package anonyme_klasse;

public class Liste {
    private Element kopf, ende;

    private static class Element {
        private Object obj;
        private Element next;
    }

    public void add(Object obj) {
        if (obj == null)
            return;
        Element neu = new Element();
        neu.obj = obj;
        if (kopf == null)
            kopf = ende = neu;
        else {
            ende.next = neu;
            ende = neu;
        }
    }

    public Iterator iterator() {
        return new Iterator() {
            private Element e = kopf;

            public boolean hasNext() {
                return e != null;
            }

            public Object next() {
                if (e == null)
                    return null;
                Object obj = e.obj;
                e = e.next;
                return obj;
            }
        };
    }
}
```

Die mit der Java-Version 8 eingeführten Lambda-Ausdrücke können in vielen Fällen anonyme Klassen ersetzen und die Programmierung vereinfachen. Lambda-Ausdrücke werden in einem späteren Kapitel ausführlich behandelt.

Da *Records* implizit `static` sind, können sie auch innerhalb von Klassen definiert werden.

## 11.5 Aufgaben

### Aufgabe 1

Implementieren Sie einen *Stack* (Stapel) mit den Methoden `push` und `pop` (siehe Aufgabe 8 in Kapitel 6). Die Elemente des Stapels sind Objekte der geschachtelten Klasse `Node`:

```
static class Node {
    int data;
    Node next;
}
```

Diese Objekte sind über `next` miteinander verkettet. Die Instanzvariable `top` der Klasse `Stack` ist vom Typ `Node` und zeigt auf das oberste Element im Stapel.

*Lösung: Paket stack*

### Aufgabe 2

Eine *Queue* (Warteschlange) kann eine beliebige Menge von Objekten aufnehmen und gibt diese in der Reihenfolge ihres Einfügens wieder zurück. Die Elemente der Queue sind Objekte der geschachtelten Klasse `Node`:

```
static class Node {
    int data;
    Node next;
}
```

Es stehen die folgenden Methoden zur Verfügung:

```
void enter(int x)
    fügt ein Objekt hinzu.

int leave()
    gibt den Inhalt des Objekts zurück und entfernt es aus der Schlange.
```

Dabei wird nach dem *FIFO-Prinzip* (First In – First Out) gearbeitet. Es wird von `leave` immer das Objekt aus der Warteschlange zurückgegeben, das von den in der Warteschlange noch vorhandenen Objekten als erstes mit `enter` hinzugefügt wurde. Die Objekte vom Typ `Node` sind über `next` miteinander verkettet. Die Instanzvariablen `head` und `tail` der Klasse `Queue` sind vom Typ `Node` und zeigen auf das erste bzw. letzte Element der Schlange.

*Lösung: Paket queue*



## 12 Konstanten und enum-Aufzählungen

*Konstanten* sind Variablen, die nach der erstmaligen Initialisierung nicht mehr verändert werden können. Der Definition werden die Schlüsselwörter `static` und `final` vorangestellt.

Beispiel:

```
static final PI = 3.14159;
```

Wird eine Referenzvariable mit `final` versehen, so kann die Referenz nicht mehr geändert werden. Wird ein Objekt referenziert, ist dieses aber im Allgemeinen veränderbar.

Eine *Aufzählung* ist ein mit dem Schlüsselwort `enum` definierter Datentyp, dessen Wertebereich aus einer Gruppe von Konstanten besteht.

### Lernziele

In diesem Kapitel lernen Sie

- welche Vorteile `enum`-Aufzählungen haben und
- wie weitere Eigenschaften solcher Aufzählungen genutzt werden können.

### 12.1 Verwendung von int-Konstanten

Beispiel:

Der Wertebereich des Aufzählungstyps `Ampelfarbe` wird durch die Aufzählung der Konstanten `ROT`, `GELB` und `GRUEN` beschrieben.

Im folgenden Programm werden die drei Ampelfarben als vordefinierte Konstanten vom Typ `int` in einer eigenen Klasse vereinbart.

```
package int_konstanten;

public class Ampelfarbe {
    public static final int ROT = 0;
    public static final int GELB = 1;
    public static final int GRUEN = 2;
}

package int_konstanten;

public class Test {
    public static String info(int farbe) {
        return switch (farbe) {
            case Ampelfarbe.ROT -> "Anhalten";
            case Ampelfarbe.GELB -> "Achtung";
            case Ampelfarbe.GRUEN -> "Weiterfahren";
        }
    }
}
```

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_12](https://doi.org/10.1007/978-3-658-43574-5_12).

```

        default -> "???";
    };
}

public static void main(String[] args) {
    System.out.println(info(Ampelfarbe.ROT));
    System.out.println(info(Ampelfarbe.GELB));
    System.out.println(info(Ampelfarbe.GRUEN));
    System.out.println(info(4711));
}
}

```

Ausgabe des Programms:

```

Anhalten
Achtung
Weiterfahren
???

```

Diese Implementierung einer Aufzählung ist *nicht typsicher*, da die Methode `info` mit beliebigen `int`-Werten aufgerufen werden kann, also nicht nur mit den vorgegebenen `int`-Konstanten `ROT`, `GELB` oder `GRUEN`.

Im nächsten Programm kann die Methode `info` nur mit den vorgegebenen Konstanten (Objekte der Klasse `Ampelfarbe`) aufgerufen werden. Es können keine neuen Objekte der Klasse `Ampelfarbe` erzeugt werden, da der Konstruktor als `private` gekennzeichnet ist. Der Wertebereich ist also auf die drei Konstanten `ROT`, `GELB` und `GRUEN` beschränkt.

Diese Implementierung ist *typsicher*, aber mit einem hohen Aufwand verbunden.

```

package typ_sicher;

public class Ampelfarbe {
    public static final Ampelfarbe ROT = new Ampelfarbe(0);
    public static final Ampelfarbe GELB = new Ampelfarbe(1);
    public static final Ampelfarbe GRUEN = new Ampelfarbe(2);

    private final int value;

    private Ampelfarbe(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

package typ_sicher;

public class Test {
    public static String info(Ampelfarbe farbe) {

```

```

        if (farbe == Ampelfarbe.ROT)
            return "Anhalten";
        else if (farbe == Ampelfarbe.GELB)
            return "Achtung";
        else if (farbe == Ampelfarbe.GRUEN)
            return "Weiterfahren";
        else return "???";
    }

    public static void main(String[] args) {
        System.out.println(info(Ampelfarbe.ROT));
        System.out.println(info(Ampelfarbe.GELB));
        System.out.println(info(Ampelfarbe.GRUEN));
    }
}

```

## 12.2 Einfache enum-Aufzählung

Typsichere Aufzählungen können ab Java-Version 5 als spezielle Art von Klassen mit dem Schlüsselwort `enum` realisiert werden:

```
enum Bezeichner { Werteliste }
```

Die Werte des Aufzählungstyps `Bezeichner` bestehen aus einer festgelegten Menge von benannten konstanten Objekten dieses Typs. Alle `enum`-Klassen sind implizit von der abstrakten Klasse `java.lang.Enum` abgeleitet.

`enum`-Konstanten können als `case`-Konstanten in `switch`-Anweisungen genutzt werden. Es ist kein `default` nötig, wenn alle Werte der Aufzählung abgedeckt sind.

Die Methode `ordinal` eines Aufzählungstyps liefert die Positionsnummer der Konstanten in der Aufzählung.

Die Methoden `name` und `toString` liefern den Namen der Konstanten, wie er in der Werteliste angegeben ist. Allerdings kann `toString` überschrieben werden.

Die Klassenmethode `values` liefert ein Array mit allen Aufzählungskonstanten.

```

package enum_aufzaehlung1;

public enum Ampelfarbe {
    ROT, GELB, GRUEN
}

package enum_aufzaehlung1;

public class Test {
    public static String info(Ampelfarbe farbe) {
        return switch (farbe) {
            case ROT -> "Anhalten";
            case GELB -> "Achtung";
            case GRUEN -> "Weiterfahren";
        };
    }
}

```

```

public static void main(String[] args) {
    System.out.println(info(Ampelfarbe.ROT));
    System.out.println(info(Ampelfarbe.GELB));
    System.out.println(info(Ampelfarbe.GRUEN));

    for (Ampelfarbe farbe : Ampelfarbe.values()) {
        System.out.println(farbe.ordinal() + " " + farbe.name() + " " + farbe);
    }
}

```

Ausgabe des Programms:

```

Anhalten
Achtung
Weiterfahren
0 ROT ROT
1 GELB GELB
2 GRUEN GRUEN

```

Eine `enum`-Aufzählung kann auch innerhalb einer Klasse stehen und sogar lokal innerhalb einer Methode.

Einige weitere Eigenschaften von `enum`-Aufzählungen:<sup>1</sup>

- Jede `enum`-Aufzählung ist automatisch `static`.
- Jede `enum`-Aufzählung ist automatisch `final`, es kann also nicht von ihr abgeleitet werden.
- Jeder Wert einer `enum`-Aufzählung ist vom Typ der Aufzählung selbst.
- Jeder `enum`-Wert kommt zur Laufzeit nur ein einziges Mal vor. Die Gleichheit von `enum`-Werten kann mit Hilfe von `==` überprüft werden.

## 12.3 `enum`-Aufzählung mit Attributen und Methoden

Ein Aufzählungstyp kann wie eine Klasse weitere Attribute und Methoden haben. Bei der Definition der Konstanten werden in runden Klammern Argumente für den Konstruktoraufruf angegeben. Der Konstruktor ist standardmäßig `private`.

Das folgende Programm definiert den Aufzählungstyp `Note` mit den Werten `SEHR_GUT`, `GUT`, `BEFRIEDIGEND`, `AUSREICHEND` und `MANGELHAFT`. Die jeweiligen konstanten Objekte dieses Aufzählungstyps enthalten die Attribute `von` und `bis`, die die Umrechnung von Punktzahlen in Noten ermöglichen. Hat z. B. der Prüfling in der Klausur 75 Punkte erhalten, so erhält er die Note "gut".

---

<sup>1</sup> Heitzmann, C.: Der Einsatz von enums abseits reiner Aufzählungen, JavaSPEKTRUM 1/2019, S. 46 – 50

```
package noten;

public enum Note {
    SEHR_GUT(82, 90), GUT(70, 81), BEFRIEDIGEND(58, 69),
    AUSREICHEND(46, 57), MANGELHAFT(0, 45);

    private final int von;
    private final int bis;

    Note(int von, int bis) {
        this.von = von;
        this.bis = bis;
    }

    public String getPunkte() {
        return von + " - " + bis;
    }

    public static Note getNote(int punkte) {
        Note result = null;
        for (Note n : Note.values()) {
            if (n.von <= punkte && punkte <= n.bis) {
                result = n;
                break;
            }
        }
        return result;
    }
}
```

```
package noten;

public class NotenTest {
    public static void main(String[] args) {
        for (Note n : Note.values()) {
            System.out.println(n + ": " + n.getPunkte());
        }

        System.out.println("50 Punkte: " + Note.getNote(50));
        System.out.println("82 Punkte: " + Note.getNote(82));
    }
}
```

Ausgabe des Programms:

```
SEHR_GUT: 82 - 90
GUT: 70 - 81
BEFRIEDIGEND: 58 - 69
AUSREICHEND: 46 - 57
MANGELHAFT: 0 - 45
50 Punkte: AUSREICHEND
82 Punkte: SEHR_GUT
```

Das Semikolon am Ende der Aufzählungswerte ist immer dann nötig, wenn noch andere Attribute und Methoden folgen.

## 12.4 Konstantenspezifische Implementierung von Methoden

enum-Aufzählungen können beliebige Interfaces bzw. abstrakte Methoden implementieren.

Das demonstriert das folgende Beispiel:

```
package enum_aufzaehlung2;

public enum Ampelfarbe {
    ROT {
        @Override
        public String info() {
            return "Anhalten";
        }
    },
    GELB {
        @Override
        public String info() {
            return "Achtung";
        }
    },
    GRUEN {
        @Override
        public String info() {
            return "Weiterfahren";
        }
    };
    public abstract String info();
}

package enum_aufzaehlung2;

public class Test {
    public static void main(String[] args) {
        System.out.println(Ampelfarbe.ROT.info());
        System.out.println(Ampelfarbe.GELB.info());
        System.out.println(Ampelfarbe.GRUEN.info());
    }
}
```

Für jeden Aufzählungswert wird direkt in einer anonymen Klasse die abstrakte Methode `info` überschrieben.

## 12.5 Singleton

Ein *Singleton* ist eine Klasse, von der es während der Laufzeit der Anwendung *nur eine einzige* Instanz gibt. Singletons werden benutzt, um beispielsweise teure Ressourcen garantiert nur einmal zu erzeugen.

```
package singleton1;

public final class ClassicSingleton {
    public static final ClassicSingleton INSTANCE = new ClassicSingleton();
    private final double value;

    private ClassicSingleton() {
        value = Math.random();
    }

    public double getValue() {
        return value;
    }
}

package singleton1;

public class Test {
    public static void main(String[] args) {
        ClassicSingleton singleton1 = ClassicSingleton.INSTANCE;
        System.out.println(singleton1.getValue());

        ClassicSingleton singleton2 = ClassicSingleton.INSTANCE;
        System.out.println(singleton2.getValue());
    }
}
```

Die statische Methode `Math.random` erzeugt eine Zufallszahl zwischen 0 und 1.

Ausgabe des Programms (Beispiel):

```
0.021986960676211087
0.021986960676211087
```

Das Wesentliche:

- Der Konstruktor eines Singletons muss `private` sein, damit von außen keine Instanzen erzeugt werden können.
- Die einzige Instanz des Singletons ist in einem öffentlichen und statischen Attribut gespeichert, um den Zugriff von außen zu gewähren.

Eine enum-Aufzählung *mit genau einem Wert* ist eine elegante Möglichkeit, ein Singleton zu implementieren:

```
package singleton2;

public enum EnumSingleton {
    INSTANCE;

    private final double value = Math.random();

    public double getValue() {
        return value;
    }
}
```

```
package singleton2;

public class Test {
    public static void main(String[] args) {
        EnumSingleton singleton1 = EnumSingleton.INSTANCE;
        System.out.println(singleton1.getValue());

        EnumSingleton singleton2 = EnumSingleton.INSTANCE;
        System.out.println(singleton2.getValue());
    }
}
```

## 12.6 Aufgaben

### Aufgabe 1

Entwickeln Sie die enum-Aufzählung `Wochentag` mit den Werten `Mo`, `Di`, ..., `So`, einem Konstruktor zur Festlegung der Namen `Montag`, `Dienstag`, ..., `Sonntag` sowie einer statischen Methode, die die Tage bis zum Wochenende ermittelt:

```
static int bisWochenende(Wochentag wochentag)
```

*Lösung: Paket wochenende*

### Aufgabe 2

Entwickeln Sie eine enum-Aufzählung, um die vier Grundrechenarten zu repräsentieren. Verwenden Sie eine konstantenspezifische Methodenimplementierung. Überschreiben Sie jeweils die abstrakte Methode

```
public abstract double compute(double x, double y);
```

*Lösung: Paket rechnen*



## 13 Ausnahmebehandlung

Während der Ausführung eines Programms können diverse Fehler bzw. *Ausnahmen* (*Exceptions*) auftreten, z. B. eine ganzzahlige Division durch 0, der Zugriff auf ein Array-Element mit einem Index außerhalb der Grenzen des Arrays, der Zugriff auf eine nicht vorhandene Datei oder der Aufruf einer Methode mit einer Referenzvariablen, deren Wert null ist.

Java bietet einen Mechanismus, solche Fehler in einer strukturierten Form zu behandeln. Dabei kann die *Ausnahmebehandlung* (Exception Handling) von dem Code der Methode, in der die Ausnahmebedingung auftritt, isoliert werden. Fehlerursache und Fehlerbehandlung sind getrennt. Diese Unabhängigkeit ist gerade beim Entwickeln von Klassenbibliotheken und der Entwicklung von Paketen in unabhängigen Teams von großer Bedeutung.

Das *Grundprinzip des Mechanismus* sieht wie folgt aus:

- Das Laufzeitsystem erkennt eine Ausnahmesituation bei der Ausführung einer Methode, erzeugt ein Objekt einer bestimmten Klasse (*Ausnahmetyp*) und löst damit eine Ausnahme aus.
- Die Ausnahme kann entweder in derselben Methode abgefangen und sofort behandelt werden oder sie kann an die aufrufende Methode weitergereicht werden.
- Die Methode, an die die Ausnahme weitergereicht wurde, hat nun ihrerseits die Möglichkeit, diese entweder abzufangen und zu behandeln oder ebenfalls weiterzureichen.
- Wird die Ausnahme nur immer weitergereicht und in keiner Methode behandelt, bricht das Programm letztlich mit einer Fehlermeldung ab.

### Lernziele

In diesem Kapitel lernen Sie

- wie Ausnahmen ausgelöst, weitergereicht oder abgefangen werden können,
- wie zwischen kontrollierten und nicht kontrollierten Ausnahmen unterschieden wird und
- wie Sie eigene Ausnahmeklassen definieren können.

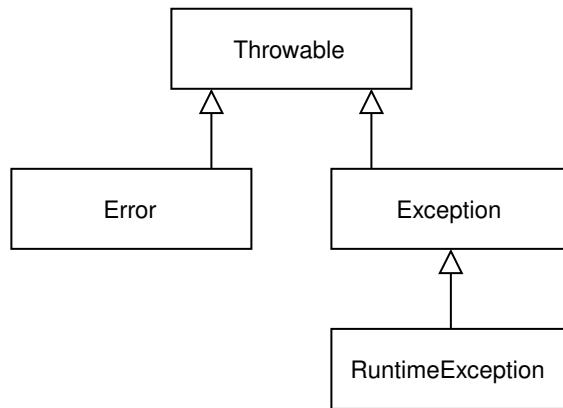
### 13.1 Ausnahmetypen

*Ausnahmen* sind Objekte der Klasse `Throwable` oder ihrer Subklassen. Die in Abbildung 13-1 aufgeführten Klassen liegen im Paket `java.lang`. Viele Pakete der Klassenbibliothek definieren ihre eigenen Ausnahmeklassen, die von den aufgeführten Klassen abgeleitet sind. Die Klasse `Error` und ihre Subklassen reprä-

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_13](https://doi.org/10.1007/978-3-658-43574-5_13).

sentieren alle schwerwiegenden Fehler, die innerhalb des Laufzeitsystems auftreten und von einem Anwendungsprogramm nicht behandelt werden können.



**Abbildung 13-1:** Typhierarchie der Ausnahmen

Die Klasse `Exception` und ihre Subklassen repräsentieren sogenannte normale Fehler eines Programms.

### Kontrollierte und nicht kontrollierte Ausnahmen

Dabei gibt es zwei Varianten:

- solche Ausnahmen, die vom Programm behandelt werden müssen (*kontrollierte Ausnahmen*, auch *geprüfte Ausnahmen* genannt) und
- solche, die vom Programm behandelt werden können, aber nicht müssen (*nicht kontrollierte Ausnahmen*, auch *nicht geprüfte Ausnahmen* genannt).

Die Klasse `RuntimeException` und ihre Subklassen repräsentieren *nicht kontrollierten Ausnahmen*. Alle anderen von `Exception` abgeleiteten Klassen, die nicht Subklasse von `RuntimeException` sind, repräsentieren *kontrollierte Ausnahmen*.

```

package runtime;

public class Division {
    public static void main(String[] args) {
        for (int i = 5; i >= 0; i--) {
            System.out.println(10 / i);
        }
    }
}
  
```

Das Laufzeitsystem erzeugt eine *nicht kontrollierte Ausnahme*, die zum Programmabbruch führt:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
at runtime.Division.main(Division.java:6)
```

Die Klasse `java.lang.ArithmetricException` ist Subklasse von `RuntimeException`.

## Exception

Konstruktoren der Klasse `Exception` sind:

```
Exception()
Exception(String message)
Exception(String message, Throwable cause)
Exception(Throwable cause)
```

`message` beinhaltet eine Beschreibung der Ausnahme, `cause` ist die Ursache für diese Ausnahme.

## Throwable

Wichtige Methoden der Klasse `Throwable`, die durch Vererbung allen Subklassen zur Verfügung stehen, sind:

```
String getMessage()
liefert die beim Aufruf des Konstruktors angegebene Beschreibung der Aus-
nahme.

void printStackTrace()
gibt Informationen zum Laufzeit-Stack aus.
```

Die beiden folgenden Methoden werden später in diesem Kapitel verwendet:

```
Throwable initCause(Throwable e)
legt die Ursache für die Ausnahme fest und liefert eine Referenz auf die
Ausnahme, für die diese Methode aufgerufen wurde.

Throwable getCause()
liefert die Ursache dieser Ausnahme oder null.
```

## Selbst Ausnahmeklassen definieren

Die Ausnahmeklasse `KontoAusnahme` wird in den nachfolgenden Beispielen genutzt, wenn ein negativer Betrag ein- bzw. ausgezahlt oder zu viel Geld abgehoben werden soll. Hier handelt es sich um eine *kontrollierte Ausnahme*, die im Programm behandelt werden muss.

```
package konto;

public class KontoAusnahme extends Exception {
    public KontoAusnahme() {
    }
```

```
public KontoAusnahme(String message) {
    super(message);
}
```

### Standard-Ausnahmen

Statt eigene Ausnahmeklassen für eine Anwendung zu definieren, sollte vorher geprüft werden, ob nicht Standardausnahmen der Java-Bibliothek wiederverwendet werden können.

Die folgende Tabelle enthält einige häufig verwendete *nicht kontrollierte* Ausnahmen:

**Tabelle 13-1:** Häufig verwendete nicht kontrollierte Ausnahmen

Ausnahme	Beschreibung
IllegalArgumentException	Parameterwerte sind ungeeignet.
IllegalStateException	Der Objektzustand lässt den Methodenaufruf nicht zu.
NullPointerException	Referenz ist null.
IndexOutOfBoundsException	Der Index liegt nicht im gültigen Wertebereich.
UnsupportedOperationException	Die Methode wird nicht unterstützt.
ArithmaticException	Eine arithmetische Bedingung ist nicht erfüllt.
NumberFormatException	Eine Zeichenkette kann nicht in eine Zahl konvertiert werden.

## 13.2 Auslösung und Weitergabe von Ausnahmen

*Kontrollierte Ausnahmen müssen entweder abgefangen und behandelt oder an den Aufrufer weitergegeben werden (catch or throw). Der Compilerachtet auf die Einhaltung dieser Regel. Nicht kontrollierte Ausnahmen können so behandelt werden, müssen aber nicht.*

### throw

Mit Hilfe der Anweisung

```
throw Ausnahmeobjekt;
```

wird eine Ausnahme ausgelöst. `throw` unterbricht das Programm an der aktuellen Stelle, um die Ausnahme zu behandeln oder die Weitergabe auszuführen.

## throws

Wird die kontrollierte Ausnahme nicht in der sie auslösenden Methode behandelt, muss sie weitergereicht werden. Die Weitergabe geschieht mit Hilfe der `throws`-Klausel im Methodenkopf:

`throws Exception-Liste`

`Exception-Liste` führt einen oder mehrere durch Kommas getrennte Ausnahmetypen auf.

### Regeln für throws-Klauseln

Nur die Ausnahmen müssen aufgelistet werden, die *nicht* in der Methode abgefangen werden. Eine hier aufgeführte Ausnahmeklasse kann auch Superklasse der Klasse des Ausnahmeobjekts in der `throw`-Klausel sein.

### Ausnahmen und Vererbung

Wird eine geerbte Methode überschrieben, so darf die neue Methode *nicht mehr* kontrollierte Ausnahmen in der `throws`-Klausel aufführen als die geerbte Methode selbst. Außerdem müssen die Ausnahmen in der Subklasse zu denen der `throws`-Klausel in der Superklasse zuweisungskompatibel sein.

Eine analoge Aussage gilt für die Implementierung von Interface-Methoden mit `throws`-Klausel.

Ein Konstruktor und die Methoden `setSaldo`, `zahleEin` und `zahleAus` der Klasse `Konto` können im folgenden Programm Ausnahmen vom Typ `KontoAusnahme` auslösen.

Die Ausnahmen werden dann an den Aufrufer der jeweiligen Methode zur Behandlung weitergereicht.

```
package konto;

public class Konto {
    private int kontonummer;
    private double saldo;

    public Konto() {
    }

    public Konto(int kontonummer) {
        this.kontonummer = kontonummer;
    }

    public Konto(int kontonummer, double saldo) throws KontoAusnahme {
        if (saldo < 0)
            throw new KontoAusnahme("Negativer Saldo: " + saldo);
    }
}
```

```

        this.kontonummer = kontonummer;
        this.saldo = saldo;
    }

    public int getKontonummer() {
        return kontonummer;
    }

    public void setKontonummer(int nr) {
        kontonummer = nr;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double betrag) throws KontoAusnahme {
        if (betrag < 0)
            throw new KontoAusnahme("Negativer Saldo: " + betrag);
        saldo = betrag;
    }

    public void zahleEin(double betrag) throws KontoAusnahme {
        if (betrag < 0)
            throw new KontoAusnahme("Negativer Betrag: " + betrag);
        saldo += betrag;
    }

    public void zahleAus(double betrag) throws KontoAusnahme {
        if (betrag < 0)
            throw new KontoAusnahme("Negativer Betrag: " + betrag);
        if (saldo < betrag)
            throw new KontoAusnahme("Betrag > Saldo");
        saldo -= betrag;
    }

    public void info() {
        System.out.println("Kontonummer: " + kontonummer + " Saldo: " + saldo);
    }
}

package konto;

public class Test1 {
    public static void main(String[] args) throws KontoAusnahme {
        // Ausnahmen vom Typ KontoAusnahme werden weitergereicht
        // und führen zum Abbruch des Programms.

        Konto kto = new Konto(4711, 500);
        kto.zahleAus(1000);
        kto.info();
    }
}

```

Das Programm bricht ab mit:

Exception in thread "main" konto.KontoAusnahme: Betrag > Saldo

Das Fehlen der `throws`-Klausel in diesem Beispiel würde zu einer Fehlermeldung des Compilers führen.

## 13.3 Auffangen von Ausnahmen

### try ... catch

Das Auffangen von Ausnahmen innerhalb einer Methode erfolgt mit Hilfe der Anweisung `try`:

```
try {  
    Anweisungen  
} catch (Ausnahmetyp Bezeichner) {  
    Anweisungen  
}
```

Der `try`-Block legt den Bereich fest, in dem die abzufangenden Ausnahmen auftreten können.

Tritt in diesem Block eine Ausnahme auf, die zum Ausnahmetyp der `catch`-Klausel passt, so fährt die Programmausführung mit der ersten Anweisung dieses `catch`-Blocks fort. Anweisungen im `try`-Block, die hinter der Anweisung liegen, die die Ausnahme verursacht hat, werden nicht mehr ausgeführt.

Im `catch`-Block kann eine Fehlerbehebung oder eine andere Reaktion auf die Ausnahme codiert werden.

Es können mehrere `catch`-Blöcke für unterschiedliche Ausnahmetypen codiert werden. Es wird immer die erste `catch`-Klausel gesucht, deren Parameter das Ausnahmeobjekt zugewiesen werden kann. Hierauf folgende `catch`-Blöcke werden nicht durchlaufen.

Kommen `catch`-Klauseln vor, deren Ausnahmeklassen voneinander abgeleitet sind, so muss die Klausel mit dem spezielleren Typ (Subklasse) vor der Klausel mit dem allgemeineren Typ (Superklasse) erscheinen.

Ist kein passender `catch`-Block vorhanden, wird die aktuelle Methode beendet und die Ausnahme an die aufrufende Methode weitergereicht.

Tritt im `try`-Block keine Ausnahme auf, wird mit der Anweisung hinter dem letzten `catch`-Block fortgesetzt.

### Multicatch

Mehrere Ausnahmetypen können zusammenfassend in einem einzigen `catch`-Block behandelt werden. Sie dürfen dann aber in keinem Subklassen-Verhältnis zueinander stehen.

Die verschiedenen Typen werden durch das "Oder"-Symbol `|` voneinander getrennt:

```
catch (Ausnahmetyp1 | Ausnahmetyp2 | ... Bezeichner) { ... }
```

Hierdurch kann redundanter Code vermieden werden.

```
package multicatch;

public class Division {
    public static void main(String[] args) {
        try {
            int a = Integer.parseInt(args[0]);
            System.out.println(100 / a);
        } catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
            System.out.println(e.getMessage());
        } catch (ArithmException e) {
            System.out.println("Autsch! " + e.getMessage());
        }
    }
}
```

Hier werden die nicht kontrollierten Ausnahmen

`ArrayIndexOutOfBoundsException` und `NumberFormatException`

in einem einzigen `catch`-Block abgefangen. `ArrayIndexOutOfBoundsException` wird geworfen, wenn beim Aufruf des Programms der Aufrufparameter vergessen wurde. `NumberFormatException` wird geworfen, wenn dieser Parameter nicht in eine Zahl konvertiert werden konnte.<sup>1</sup> `ArithmException` (bei Division durch 0) wird getrennt behandelt.

In der `catch`-Klausel könnte hier natürlich auch eine gemeinsame Superklasse wie `RuntimeException` oder `Exception` mit einer einheitlichen Fehlerbehandlung stehen.

Im folgenden Programm werden mögliche Ausnahmen abgefangen.

```
package konto;

public class Test2 {
    public static void main(String[] args) {
        Konto kto = null;
        try {
            kto = new Konto(4711, 500);
            kto.zahleAus(1000);
            kto.info();
        } catch (KontoAusnahme e) {
            System.out.println(e);
        }
        if (kto != null)
            kto.info();
    }
}
```

---

1 Zur Umwandlung von Strings in Zahlen siehe Kapitel 15.

**Ausgabe des Programms:**

```
konto.KontoAusnahme: Betrag > Saldo  
Kontonummer: 4711 Saldo: 500.0
```

**finally**

Hinter dem letzten catch-Block kann die finally-Klausel auftreten:

```
finally {  
    Anweisungen  
}
```

Der finally-Block enthält Anweisungen, die *in jedem Fall* ausgeführt werden sollen.

Er wird durchlaufen,

- wenn der try-Block ohne Auftreten einer Ausnahme normal beendet wurde,
- wenn eine Ausnahme in einem catch-Block behandelt wurde,
- wenn eine aufgetretene Ausnahme in keinem catch-Block behandelt wurde,
- wenn der try-Block durch break, continue oder return verlassen wurde.

Die try-Anweisung kann auch ohne catch-Blöcke, aber dann mit einem finally-Block auftreten.

Das nächste Programm zeigt drei Testfälle:

- Im Schritt 1 wird die Methode fehlerfrei ausgeführt.
- Im Schritt 2 wird eine Ausnahme ausgelöst und abgefangen.
- Im Schritt 3 wird ein Laufzeitfehler (Division durch 0) ausgelöst, der nicht abgefangen wird und somit zum Programmabbruch führt.

In allen drei Fällen wird der finally-Block durchlaufen.

```
package konto;  
  
public class FinallyTest {  
    public static void main(String[] args) {  
        try {  
            Konto kto = new Konto(4711, 500);  
  
            for (int i = 1; i <= 3; i++) {  
                System.out.println("BEGINN SCHRITT " + i);  
  
                try {  
                    switch (i) {  
                        case 1 -> kto.zahleAus(100);  
                        case 2 -> kto.zahleAus(700);  
                    }  
                } catch (Exception e) {  
                    System.out.println("Fangblock " + i);  
                }  
            }  
        } catch (Exception e) {  
            System.out.println("Fangblock");  
        }  
    }  
}
```

```
        case 3 -> kto.zahleAus(200 / 0);
    }
} catch (KontoAusnahme e) {
    System.out.println(e);
} finally {
    System.out.println("Ausgabe im finally-Block: " + kto.getSaldo());
}

System.out.println("ENDE SCHRITT " + i);
System.out.println();
}
} catch (KontoAusnahme e) {
    System.out.println(e);
}
}
```

## Ausgabe des Programms:

```
BEGINN SCHRITT 1
Ausgabe im finally-Block: 400.0
ENDE SCHRITT 1
```

```
BEGINN SCHRITT 2
KontoAusnahme: Betrag > Saldo
Ausgabe im finally-Block: 400.0
ENDE SCHRITT 2
```

```
BEGINN SCHRITT 3
Ausgabe im finally-Block: 400.0
Exception in thread "main" java.lang.ArithmetricException: / by zero
    at konto.FinallyTest.main(FinallyTest.java:15)
```

## 13.4 Verkettung von Ausnahmen

Eine Methode kann in einem `catch`-Zweig eine Ausnahme abfangen und eine neue Ausnahme eines anderen Typs an den Aufrufer der Methode weitergeben. Dabei kann die ursprüngliche Ausnahme als Ursache (*cause*) in der neuen Ausnahme gespeichert werden.

Dieser Mechanismus wird als *Ausnahmen-Verkettung* (*exception chaining*) bezeichnet.

Verschiedene Ausnahmen einer niedrigen (implementierungsnahen) Ebene können so in eine Ausnahme einer höheren (anwendungsnahen) Ebene übersetzt werden, wobei die ursprüngliche Ausnahme mit der `Throwable`-Methode `getCause` abgerufen werden kann.

Wir demonstrieren die *Ausnahmen-Verkettung* anhand eines einfachen Beispiels in zwei Varianten:

Die Methode `getNachbarn` soll für ein vorgegebenes `int`-Array und eine vorgegebene Indexposition `i` die Werte an den Positionen `i-1`, `i` und `i+1` ausgeben.

Hierbei werden Laufzeitfehler wie `NullPointerException` und `ArrayIndexOutOfBoundsException` abgefangen und in einer Ausnahme eines anderen Typs weitergereicht.

Im *ersten Fall* werden die `Throwable`-Methoden `initCause` und `getCause` zum Speichern bzw. Abfragen der ursprünglichen Ausnahme genutzt.

```
package chaining1;

public class MyException extends Exception {
}

package chaining1;

public class ChainingTest {
    public static String getNachbarn(int[] x, int i) throws MyException {
        try {
            return x[i - 1] + " " + x[i] + " " + x[i + 1];
        } catch (RuntimeException e) {
            throw (MyException) new MyException().initCause(e);
        }
    }

    public static void main(String[] args) {
        int[] a = null;
        int[] b = { 1, 2, 3, 4, 5 };

        try {
            System.out.println(getNachbarn(a, 4));
        } catch (MyException e) {
            System.out.println("Ursache: " + e.getCause());
        }

        try {
            System.out.println(getNachbarn(b, 4));
        } catch (MyException e) {
            System.out.println("Ursache: " + e.getCause());
        }
    }
}
```

Im *zweiten Fall* enthält die Ausnahmeklasse `MyException` einen Konstruktor, an den die Referenz auf die ursprüngliche Ausnahme direkt als Argument übergeben wird.

```
package chaining2;

public class MyException extends Exception {
    public MyException() {}

    public MyException(Throwable t) {
        super(t);
    }
}
```

```

package chaining2;

public class ChainingTest {
    public static String getNachbarn(int[] x, int i) throws MyException {
        try {
            return x[i - 1] + " " + x[i] + " " + x[i + 1];
        } catch (RuntimeException e) {
            throw new MyException(e);
        }
    }

    public static void main(String[] args) {
        int[] a = null;
        int[] b = { 1, 2, 3, 4, 5 };

        try {
            System.out.println(getNachbarn(a, 4));

        } catch (MyException e) {
            System.out.println("Ursache: " + e.getCause());
        }

        try {
            System.out.println(getNachbarn(b, 4));
        } catch (MyException e) {
            System.out.println("Ursache: " + e.getCause());
        }
    }
}

```

Ausgabe beider Programme:

```

Ursache: java.lang.NullPointerException: Cannot load from int array because "x" is
null
Ursache: java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length
5

```

## 13.5 Aufgaben

### Aufgabe 1

Erläutern Sie den Mechanismus der Ausnahmebehandlung. Welche Java-Schlüsselwörter sind mit den einzelnen Schritten verbunden?

*Lösung: Ausnahmebehandlung.txt*

### Aufgabe 2

Was besagt die *catch-or-throw*-Regel?

*Lösung: Regel.txt*

### Aufgabe 3

Implementieren Sie die statische Methode

```
public static long sum(int n),
```

die die Summe  $1 + 2 + \dots + n$  ermittelt. Ist  $n \leq 0$ , soll die Methode die nicht kontrollierte Ausnahme `IllegalArgumentException` auslösen.

*Lösung: Paket sum*

### Aufgabe 4

Erstellen Sie die Klasse `Monat` mit dem Attribut

```
private int monat,
```

dem Konstruktor

```
public Monat(int monat) throws MonatAusnahme
```

und der Methode

```
public String getMonatsname(),
```

die den Namen zur Monatszahl zurückgibt.

Wird beim Aufruf des Konstruktors eine Zahl außerhalb des Bereichs von 1 bis 12 als Argument eingesetzt, soll eine Ausnahme vom Typ `MonatAusnahme` ausgelöst und weitergereicht werden. Beim Aufruf von `getMessage` für ein Ausnahmeobjekt dieser Klasse soll die falsche Monatsangabe im Fehlertext mit ausgegeben werden.

Testen Sie den Konstruktor und die Methode und fangen Sie Ausnahmen ab.

*Lösung: Paket monat*

### Aufgabe 5

Die `int`-Variable `zufallszahl` kann in einer Schleife zufällige Werte zwischen 0 und 99 annehmen. Es soll eine Ausnahme vom Typ `Exception` ausgelöst und abgefangen werden, wenn der Wert 0 ist. Die Zufallszahl kann mittels

```
(int) (Math.random() * 100.)
```

erzeugt werden.

*Lösung: Paket random*

### Aufgabe 6

Erzeugen Sie eine Klasse `Person`, die nur einen Wert zwischen 0 und 120 für das Lebensalter einer Person erlaubt. Im Konstruktor der Klasse soll im Fall des Verstoßes gegen diese Regel eine nicht kontrollierte Ausnahme vom Typ `OutOfRangeException` ausgelöst werden. Diese selbst definierte Ausnahme soll Informationen über den ungültigen Wert sowie die Grenzen des gültigen Intervalls zur Verfügung stellen.

Schreiben Sie auch ein Programm, das verschiedene Fälle testet.

*Lösung: Paket out\_of\_range*

**Aufgabe 7**

Schreiben Sie ein Programm, das ein sehr großes Array erzeugt und damit eine Ausnahme vom Typ `OutOfMemoryError` auslöst. `OutOfMemoryError` ist Subklasse von `Error`.

*Lösung: Paket out\_of\_memory*



# 14 Zeichenketten

Zeichenketten (Strings) werden durch die Klasse `String` repräsentiert. Daneben gibt es eine Reihe weiterer Klassen, die die Verarbeitung von Zeichenketten unterstützen, z. B. `StringBuilder` und `StringBuffer`.

## Lernziele

In diesem Kapitel lernen Sie

- wie Zeichenketten verwaltet werden,
- wie mit Zeichenketten gearbeitet werden kann und
- welche Klassen zur effizienten Verarbeitung von Texten zur Verfügung stehen.

## 14.1 Die Klasse String

Die Klasse `java.lang.String` repräsentiert Zeichenketten, bestehend aus Unicode-Zeichen.

*Strings sind nicht änderbar*

Objekte vom Typ `String` (im Folgenden auch kurz *Strings* genannt) sind nach der Initialisierung *nicht mehr veränderbar*. Bei jeder String-Manipulation durch eine der unten aufgeführten Methoden wird ein neues `String`-Objekt erzeugt. Alle `String`-Literale werden als Objekte dieser Klasse implementiert.

So zeigt z. B. die Referenzvariable `str` nach der Initialisierung durch

```
String str = "Hallo Welt";
```

auf das Objekt, das die Zeichenkette `Hallo Welt` repräsentiert.

Das Laufzeitsystem verwaltet alle `String`-Literale eines Programms in einem Pool und liefert für alle Objekte, die das gleiche `String`-Literal repräsentieren, dieselbe Referenz zurück.

### Konstruktoren

`String()`

erzeugt eine leere Zeichenkette mit dem Wert `" "`.

`String(String s)`

erzeugt einen String, der eine Kopie des Strings `s` ist.

`String(char[] c)`

`String(byte[] b)`

`String(char[] c, int offset, int length)`

`String(byte[] b, int offset, int length)`

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_14](https://doi.org/10.1007/978-3-658-43574-5_14).

Diese Konstruktoren erzeugen einen String aus einem char- bzw. byte-Array. Eine anschließende Inhaltsänderung des benutzten Arrays beeinflusst nicht die Zeichenkette.

Bei der Umwandlung von Bytes in Unicode-Zeichen wird die Standardcodierung *UTF-8* verwendet (siehe Kapitel 24.2).

In den beiden letzten Fällen werden `length` Zeichen bzw. Bytes aus einem Array beginnend mit dem Index `offset` in eine Zeichenkette umgewandelt.

`int length()`

gibt die Länge dieser Zeichenkette zurück. Diese entspricht der Anzahl der 16-Bit-Unicode-Zeichen in der Zeichenkette.

`boolean isEmpty()`

liefert true genau dann, wenn die Länge 0 ist.

`String concat(String s)`

liefert einen neuen String, der aus der Aneinanderreihung des Strings, für den die Methode aufgerufen wurde, und des Strings `s` besteht.

Strings können auch mit dem Operator `+` verkettet werden. Ein `String`-Objekt entsteht auch dann, wenn dieser Operator auf einen String und eine Variable oder ein Literal vom einfachen Datentyp angewandt wird.

`String repeat(int count)`

liefert die `count`-malige Aneinanderreihung des Strings, für den die Methode aufgerufen wurde.

```
public class Erzeugen {
    public static void main(String[] args) {
        String a = "Hallo";
        String b = "Hallo";
        String c = new String(a);
        String d = new String(new char[]{ 'H', 'a', 'l', 'l', 'o' });
        String e = a.concat(" ").concat("Welt");
        String f = a + " " + "Welt";
        int preis = 5;
        String g = "Dieser Artikel kostet " + preis + " Euro.";
        String h = "*".repeat(10);

        System.out.println(a == b);
        System.out.println(a == c);
        System.out.println(d);
        System.out.println(e);
        System.out.println(f);
        System.out.println(f.length());
        System.out.println(g);
        System.out.println(h);
    }
}
```

Ausgabe des Programms:

```
true  
false  
Hallo  
Hallo Welt  
Hallo Welt  
10  
Dieser Artikel kostet 5 Euro.  
*****
```

## Vergleichen

`boolean equals(Object obj)`

liefert true, wenn obj ein String ist und beide Strings die gleiche Zeichenkette darstellen.

`boolean equalsIgnoreCase(String s)`

wirkt wie equals, ignoriert aber eventuelle Unterschiede in der Groß- und Kleinschreibung.

`boolean startsWith(String s, int start)`

liefert true, wenn dieser String mit der Zeichenkette s an der Position start beginnt.

`boolean startsWith(String s)`

ist gleichbedeutend mit `startsWith(s, 0)`.

`boolean endsWith(String s)`

liefert true, wenn dieser String mit der Zeichenkette s endet.

`boolean regionMatches(int i, String s, int j, int length)`

liefert true, wenn der Teilstring dieses Strings, der ab der Position i beginnt und die Länge length hat, mit dem Teilstring von s übereinstimmt, der an der Position j beginnt und length Zeichen lang ist.

`int compareTo(String s)`

vergleicht diese Zeichenkette mit der Zeichenkette s lexikographisch und liefert 0, wenn beide Zeichenketten gleich sind, einen negativen Wert, wenn diese Zeichenkette kleiner ist als s, und einen positiven Wert, wenn diese Zeichenkette größer ist als s.

`int compareToIgnoreCase(String s)`

wirkt wie compareTo, ignoriert aber eventuelle Unterschiede in der Groß- und Kleinschreibung.

`boolean isBlank()`

prüft, ob der String leer ist oder nur *white spaces* enthält.

*White spaces* sind Leerzeichen (\u0020), Tabulator- und Zeilentrennzeichen.

```
public class Vergleichen {
    public static void main(String[] args) {
        String a = "Hallo";
        String b = "Hallo";
        String c = "Das ist ein Test";
        String d = "\n\t ";

        System.out.println(a.equals(b));
        System.out.println(c.startsWith("Das") && c.endsWith("Test"));
        System.out.println("a".compareTo("b"));
        System.out.println(d.isBlank());
    }
}
```

Ausgabe des Programms:

```
true
true
-1
true
```

## Suchen

`int indexOf(int c)`  
 liefert die Position des ersten Auftretens des Zeichens `c` in der Zeichenkette, andernfalls wird `-1` zurückgegeben.

`int indexOf(int c, int start)`  
 wirkt wie die vorige Methode mit dem Unterschied, dass die Suche ab der Position `start` in der Zeichenkette beginnt.

`int indexOf(String s)`  
 liefert die Position des ersten Zeichens des ersten Auftretens der Zeichenkette `s` in dieser Zeichenkette, andernfalls wird `-1` zurückgegeben.

`int indexOf(String s, int start)`  
 wirkt wie die vorige Methode mit dem Unterschied, dass die Suche ab der Position `start` in der Zeichenkette beginnt.

Die folgenden vier Methoden suchen analog nach dem letzten Auftreten des Zeichens `c` bzw. der Zeichenkette `s`:

```
int lastIndexOf(int c)
int lastIndexOf(int c, int start)
int lastIndexOf(String s)
int lastIndexOf(String s, int start)
```

`boolean contains(CharSequence s)`  
 liefert `true` genau dann, wenn der String die Zeichen aus `s` enthält.  
 Das Interface `java.lang.CharSequence` repräsentiert eine Folge von Zeichen und wird u. a. von `String`, `StringBuffer` und `StringBuilder` implementiert.

```
public class Suchen {
    public static void main(String[] args) {
        String str = "Das ist ein Test.";
        int idx = str.indexOf(" ");
        System.out.println(idx);
        System.out.println(str.indexOf(" ", idx + 1));
        System.out.println(str.lastIndexOf(" "));
        System.out.println(str.contains("ist"));
    }
}
```

Ausgabe des Programms:

```
3
7
11
true
```

## Extrahieren

`char charAt(int i)`

liefert das Zeichen an der Position `i` der Zeichenkette. Das erste Zeichen hat die Position `0`.

`String substring(int start)`

liefert einen String, der alle Zeichen des Strings, für den die Methode aufgerufen wurde, ab der Position `start` enthält.

`String substring(int start, int end)`

liefert einen String, der alle Zeichen ab der Position `start` bis `end - 1` enthält. Die Länge ist also `end - start`.

`String trim()`

schneidet alle zusammenhängenden Leer- und Steuerzeichen (das sind die Zeichen kleiner oder gleich '`\u0020`') am Anfang und Ende der Zeichenkette ab.

`String strip()`

entfernt führende und nachfolgende *white spaces*.

`stripLeading()` und `stripTrailing()` entfernen *white spaces* nur am Anfang bzw. nur am Ende des Strings.

```
public class Extrahieren {
    public static void main(String[] args) {
        String a = "Schmitz, Hugo";
        String b = "\n\tHallo\b ";
        System.out.println(a.substring(0, a.indexOf(",")));
        System.out.println(b.trim().length());
        System.out.println(b.strip().length());
        System.out.println("#" + b.trim() + "#");
        System.out.println("#" + b.strip() + "#");
    }
}
```

Ausgabe des Programms:

```
Schmitz
5
6
#Hallo#
#Hall#
```

Das Steuerzeichen \b (Backspace) bewegt den Cursor eine Position zurück.

## **Ersetzen**

`String replace(char c, char r)`

erzeugt einen neuen String, in dem jedes Auftreten des Zeichens c in diesem String durch das Zeichen r ersetzt ist.

`String replace(CharSequence t, CharSequence r)`

erzeugt einen neuen String, in dem jeder Teilstring, der mit t übereinstimmt, durch die Zeichen aus r ersetzt wird.

`String toLowerCase()`

wandelt die Großbuchstaben der Zeichenkette in Kleinbuchstaben um und liefert diesen String zurück.

`String toUpperCase()`

wandelt die Kleinbuchstaben der Zeichenkette in Großbuchstaben um und liefert diesen String zurück.

```
public class Ersetzen {
    public static void main(String[] args) {
        String a = "4711,Hammer,20";
        String b = a.replace(',', ';');
        System.out.println(b);
        System.out.println(b.toUpperCase());
    }
}
```

Ausgabe des Programms:

```
4711;Hammer;20
4711;HAMMER;20
```

## **Konvertieren**

```
static String valueOf(boolean x)
static String valueOf(char x)
static String valueOf(int x)
static String valueOf(long x)
static String valueOf(float x)
static String valueOf(double x)
static String valueOf(char[] x)
static String valueOf(Object x)
```

Diese Methoden wandeln `x` in einen String um. Im letzten Fall wird "null" geliefert, falls `x` den Wert null hat, sonst wird `x.toString()` zurückgegeben.

`byte[] getBytes()`

liefert ein Array von Bytes, das die Zeichen des Strings in der Standardcodierung enthält.

`byte[] getBytes(Charset charset)`

codiert diesen String unter Verwendung des angegebenen Zeichensatzes in eine Folge von Bytes (siehe Kapitel 24.2).

`char[] toCharArray()`

liefert ein char-Array, das die Zeichen des Strings enthält.

```
import java.util.Arrays;

public class Konvertieren {
    public static void main(String[] args) {
        String a = String.valueOf(3.57);
        byte[] bytes = "Hallo".getBytes();
        char[] chars = "Hallo".toCharArray();
        System.out.println(a.replace('.', ','));
        System.out.println(Arrays.toString(bytes));
        System.out.println(Arrays.toString(chars));
    }
}
```

Die statische Methode `java.util.Arrays.toString(...)` liefert eine String-Darstellung des als Argument übergebenen Arrays.

Ausgabe des Programms:

```
3,57
[72, 97, 108, 108, 111]
[H, a, l, l, o]
```

## Split und Join

Die `String`-Methode `split` zerlegt eine Zeichenkette in ein Array anhand eines Suchmusters:

`String[] split(String regex)`

Das Suchmuster kann ein sogenannter *regulärer Ausdruck* sein.<sup>1</sup>

Zur Beschreibung des Suchmusters gibt es eine eigene Syntax.<sup>2</sup> Das folgende Programm macht hiervon nicht Gebrauch und verwendet nur eine einfache Zeichenkette als Suchmuster.

<sup>1</sup> [https://de.wikipedia.org/wiki/Regulärer\\_Ausdruck](https://de.wikipedia.org/wiki/Regulärer_Ausdruck)

<sup>2</sup> [https://www.w3schools.com/java/java\\_regex.asp](https://www.w3schools.com/java/java_regex.asp)

Die statische `String`-Methode `join` hängt mehrere Zeichenketten mit einem gemeinsamen Verbindungszeichen zusammen:

```
static String join(CharSequence delimiter, CharSequence... elements)
```

```
import java.util.Arrays;

public class Split {
    public static void main(String[] args) {
        String data = "Hugo Meier;12345;Musterdorf";
        String[] words = data.split(";");
        System.out.println(Arrays.toString(words));
        String str = String.join(";", words);
        System.out.println(str);
    }
}
```

Ausgabe des Programms:

```
[Hugo Meier, 12345, Musterdorf]
Hugo Meier;12345;Musterdorf
```

## Formatieren

Die `String`-Methode

```
static String format(String format, Object... args)
```

liefert einen formatierten String, wobei für `format` eine eigene Syntax genutzt wird.

Hier wird das aktuell eingestellte *Default-Locale* verwendet. Optional kann an die Methode `format` auch ein `Locale`-Objekt als erstes Argument übergeben werden.

Format-Syntax zur Ausgabe eines Arguments:

```
%[flags][width][.precision]conversion
```

`flags` steuert die Ausgabe:

- 0 führende Nullen bei Zahlen
- + Vorzeichen bei positiven Zahlen
- linksbündige Ausgabe

`width` gibt die minimale Anzahl Zeichen an, die ausgegeben werden sollen, `precision` gibt für Zahlen die Anzahl Nachkommastellen an.

`conversion` gibt an, wie das Argument aufbereitet werden soll:

- |   |                 |   |                           |
|---|-----------------|---|---------------------------|
| d | ganze Zahl      | s | String                    |
| f | Fließkommazahl  | n | Zeilenvorschub            |
| o | Oktalzahl       | % | das Prozentzeichen selbst |
| x | Hexadezimalzahl |   |                           |

```
String formatted(Object... args)
```

entspricht der obigen Methode. Hier wird `formatted` auf den String, der das Format enthält, angewandt.

```
public class Formatieren {
    public static void main(String[] args) {
        String f = "Der Artikel %s kosten %.2f €.";
        String aus1 = String.format(f, "A4711", 7.5);
        System.out.println(aus1);
        String aus2 = f.formatted("A4711", 7.5);
        System.out.println(aus2);
    }
}
```

Ausgabe des Programms:

```
Der Artikel A4711 kosten 7,50 €.  
Der Artikel A4711 kosten 7,50 €.
```

## Die Object-Methode `toString`

Die Klasse `Object` (Wurzel der Klassenhierarchie, von der jede Klasse direkt oder indirekt erbt) enthält die Methode `toString`. Für jedes Objekt wird durch Aufruf dieser Methode eine Zeichenkettendarstellung geliefert.

Um eine sinnvolle Darstellung für Objekte einer bestimmten Klasse zu erhalten, muss `toString` in dieser Klasse überschrieben werden. `toString` wird beim Aufruf von `System.out.println(obj)` und bei der String-Verkettung mit `+` automatisch verwendet.

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name + " ist " + age + " Jahre alt.";
    }

    public static void main(String[] args) {
        Person person = new Person("Hugo", 18);
        System.out.println(person);
    }
}
```

Ausgabe des Programms:

Hugo ist 18 Jahre alt.

## 14.2 Mehrzeilige Texte

Ab Java 15 können Zeichenketten, die sich über mehrere Zeilen erstrecken, einfach als *Textblock* erstellt werden. Die mühselige Verkettung einzelner Strings mit Zeilentrennzeichen und dem Operator + ist nicht mehr nötig.

Der Textblock startet und endet mit jeweils drei Anführungszeichen.

Nach den einleitenden Zeichen """" muss ein Zeilenumbruch erfolgen. Wenn vor den abschließenden Zeichen """" ein Zeilenumbruch erfolgt, ist dieser Zeilenumbruch Teil des Strings.

Die unteren drei Anführungszeichen bestimmen den Start der Einrückung. Der Textblock beginnt bei demjenigen Zeichen, das am weitesten links steht.

```
public class TextBlocks {
    public static void main(String[] args) {
        System.out.println("""
            1) Guten Tag.
            Auf Wiedersehen.
            """
        );
        System.out.println("""
            2) Guten Tag.
            Auf Wiedersehen."""
        );
        System.out.println("""
            3) "Guten Tag."
            Auf Wiedersehen.""""
        );
        System.out.println("""
            4) Guten Tag. \
            Auf Wiedersehen.""""
        );
        System.out.println("""
            5) Guten Tag.      \s
            Auf Wiedersehen.""""
        );
    }
}
```

Doppelte Anführungszeichen können direkt im String verwendet werden. \ am Ende einer Zeile verhindert den Zeilenumbruch. \s wird in ein Leerzeichen umgewandelt und verhindert, dass sofort nach dem Text ein Umbruch erfolgt.

Ausgabe des Programms:

- 1) Guten Tag.  
Auf Wiedersehen.
- 2) Guten Tag.  
Auf Wiedersehen.
- 3) "Guten Tag."  
Auf Wiedersehen.
- 4) Guten Tag. Auf Wiedersehen.
- 5) Guten Tag.  
Auf Wiedersehen.

## 14.3 Die Klassen `StringBuilder` und `StringBuffer`

Da `String`-Objekte unveränderbar sind, entstehen bei `String`-Manipulationen fortlaufend neue `String`-Objekte als Zwischenergebnisse, was relativ aufwändig ist. Hier helfen die Klassen `StringBuilder` und `StringBuffer`.

`StringBuffer` bietet die gleichen Methoden wie die Klasse `StringBuilder` an.

Da die Methoden von `StringBuffer` dort, wo es nötig ist, synchronisiert sind, können mehrere Threads parallel auf demselben `StringBuffer`-Objekt arbeiten. Im Gegensatz hierzu ist die Klasse `StringBuilder` nicht *Thread-sicher*.

Immer wenn `String`-Puffer nur von einem einzigen Thread genutzt werden, sollten `StringBuilder`-Objekte eingesetzt werden. Die Verarbeitungsgeschwindigkeit ist hier deutlich höher.

Objekte der Klasse `java.lang.StringBuilder` enthalten *veränderbare* Zeichenketten. Bei Änderungen wird der benötigte Speicherplatz automatisch in der Größe angepasst.

### Konstruktoren

`StringBuilder()`

erzeugt einen leeren Puffer mit einer Anfangsgröße von 16 Zeichen.

`StringBuilder(int capacity)`

erzeugt einen leeren Puffer mit einer Anfangsgröße von `capacity` Zeichen.

`StringBuilder(String s)`

erzeugt einen Puffer mit dem Inhalt von `s` und einer Anfangsgröße von 16 + (Länge von `s`) Zeichen.

### Einige Methoden

`int length()`

liefert die Länge der enthaltenen Zeichenkette.

`void setLength(int newLength)`

setzt die Länge des Puffers. Ist newLength kleiner als die aktuelle Länge des Puffers, wird die enthaltene Zeichenkette abgeschnitten. Der Puffer enthält dann genau newLength Zeichen. Ist newLength größer als die aktuelle Länge, werden an die enthaltene Zeichenkette so viele Zeichen '\u0000' angehängt, bis die neue Zeichenkette newLength Zeichen enthält.

```
String toString()
    wandelt die Zeichenkette im StringBuilder-Objekt in einen String um.

String substring(int start)
String substring(int start, int end)
    liefert einen String, der bei start beginnt und die Zeichen bis zum Ende des Puffers bzw. bis end - 1 enthält.

void getChars(int srcStart, int srcEnd, char[] dst, int dstStart)
    kopiert Zeichen aus dem Puffer von srcStart bis srcEnd - 1 in das Array dst ab der Position dstStart.

StringBuilder append(Typ x)
    hängt die String-Darstellung von x an das Ende der Zeichenkette im Puffer. Typ steht hier für boolean, char, int, long, float, double, char[], String oder Object.

StringBuilder append(StringBuilder sb)
    hängt die Zeichenkette in sb an das Ende der Zeichenkette im Puffer.

StringBuilder insert(int i, Typ x)
    fügt die String-Darstellung von x an der Position i der Zeichenkette im Puffer ein. Typ wie bei append.

StringBuilder delete(int start, int end)
    löscht den Teilstring ab der Position start bis zur Position end - 1 im Puffer.

StringBuilder deleteCharAt(int i)
    löscht das Zeichen an der Position i.

StringBuilder replace(int start, int end, String s)
    ersetzt die Zeichen von start bis end - 1 durch die Zeichen in s.

Die Methoden append, insert, delete und replace verändern jeweils das Original und liefern es zusätzlich als Rückgabewert.

char charAt(int i)
    liefert das Zeichen an der Position i.

void setCharAt(int i, char c)
    ersetzt das an der Position i stehende Zeichen durch c.
```

```
StringBuilder reverse()
```

ersetzt diese Zeichenkette durch die Umkehrung der Zeichenkette.

Obige Methoden mit Rückgabetyp `StringBuilder` liefern eine Referenz auf das `StringBuilder`-Objekt, für das sie aufgerufen wurden. Das ermöglicht eine Verkettung der Aufrufe, wie das folgende Beispiel zeigt.

```
public class StringBuilderTest {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder();  
        sb.append("Ich")  
            .append("was soll es bedeuten ...")  
            .insert(3, " weiß nicht, ");  
        System.out.println(sb);  
    }  
}
```

Ausgabe des Programms:

```
Ich weiß nicht, was soll es bedeuten ...
```

## 14.4 Die Klasse StringTokenizer

Die Klasse `java.util.StringTokenizer` ist ein Werkzeug zur Zerlegung von Texten.

Ein Text in Form einer Zeichenkette wird als Mischung von Textteilen (*Token*), die aus zusammenhängenden Zeichen bestehen, und besonderen Zeichen (z. B. Leerzeichen, Interpunktionszeichen), die die Textteile voneinander trennen (*Trennzeichen*), angesehen.

Die Leistung der Klasse ist nun, den Text mittels spezieller Methoden in einzelne Tokens zu zerlegen. Dabei können die Zeichen, die als Trennzeichen dienen sollen, vorgegeben werden.

Beispiel:

Trennzeichen seien Leerzeichen, Komma und Punkt.

Die Tokens der Zeichenkette "Ich weiß nicht, was soll es bedeuten." sind dann:

```
"Ich", "weiß", "nicht", "was", "soll", "es" und "bedeuten".
```

### Konstruktoren

```
StringTokenizer(String s)  
StringTokenizer(String s, String delim)  
StringTokenizer(String s, String delim, boolean returnDelims)
```

`s` ist die Zeichenkette, die zerlegt werden soll. `delim` enthält die Trennzeichen. Ist `delim` nicht angegeben, so wird " \t\n\r\f" benutzt. Hat `returnDelims` den Wert `true`, so werden auch die Trennzeichen als Tokens geliefert.

## Methoden

`boolean hasMoreTokens()`

liefert den Wert `true`, wenn noch mindestens ein weiteres Token vorhanden ist.

`String nextToken()`

liefert das nächste Token. Falls es kein Token mehr gibt, wird die nicht kontrollierte Ausnahme `java.util.NoSuchElementException` ausgelöst.

`String nextToken(String delim)`

liefert das nächste Token, wobei jetzt und für die nächsten Zugriffe die Zeichen in `delim` als Trennzeichen gelten. Falls es kein Token mehr gibt, wird die nicht kontrollierte Ausnahme `java.util.NoSuchElementException` ausgelöst.

`int countTokens()`

liefert die Anzahl der noch verbleibenden Tokens in Bezug auf den aktuellen Satz von Trennzeichen.

```
import java.util.StringTokenizer;

public class StringTokenizerTest {
    public static void main(String[] args) {
        String text = "Ich weiß nicht, was soll es bedeuten.";
        StringTokenizer st = new StringTokenizer(text, ",.");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

## 14.5 Aufgaben

### Aufgabe 1

Implementieren Sie die Methode

```
public static String delimitedString(String s, char start, char end),
```

die den Teilstring von `s` zurückgibt, der mit dem Zeichen `start` beginnt und mit dem Zeichen `end` endet.

*Lösung: DelimitedString*

**Aufgabe 2**

Die Methoden `padStart` und `padEnd` fügen zu einem String am Anfang bzw. Ende Auffüllzeichen hinzu, bis der String die vorgegebene Länge erreicht hat. Implementieren Sie die beiden Methoden:

```
public static String padStart(String str, int length, char pad) und  
public static String padEnd(String str, int length, char pad)
```

*Lösung: Padding*

**Aufgabe 3**

Implementieren Sie die Methode

```
public static String encode(String text),
```

die einen beliebigen Text verschlüsselt, indem zunächst alle Großbuchstaben in Kleinbuchstaben gewandelt werden und dann jeder Buchstabe durch seine Positionsnummer im Alphabet ersetzt wird. Umlaute wie ä sowie ö sollen wie æ bzw. ss behandelt werden. Alle Zeichen, die keine Buchstaben sind, sollen ignoriert werden.

*Lösung: Encode*

**Aufgabe 4**

Schreiben Sie eine Methode, die testet, ob eine Zeichenkette ein *Palindrom* ist. Ein Palindrom ist eine Zeichenkette, die vorwärts und rückwärts gelesen dasselbe ergibt. Groß- und Kleinschreibung, Satzzeichen und Leerzeichen sollen nicht berücksichtigt werden.

*Lösung: Palindrom*

**Aufgabe 5**

Testen Sie den Performance-Unterschied zwischen `String`-, `StringBuilder`- und `StringBuffer`-Operationen.

In einer Schleife, die genügend oft durchlaufen wird, ist bei jedem Schleifen-durchgang eine Zeichenkette mit einem einzelnen Zeichen "x" zu verketteten; zum einen durch Verkettung von Strings mit `+`, zum anderen durch Anwendung der `StringBuilder`- bzw. `StringBuffer`-Methode `append`.

Messen Sie die aktuelle Uhrzeit unmittelbar vor und nach der Ausführung der entsprechenden Anweisungen mit `System.currentTimeMillis()`.

*Lösung: PerformanceTest*



# 15 Ausgewählte Klassen

In diesem Kapitel werden wichtige Klassen der Java-Standard-Edition vorgestellt, die nutzbringend in verschiedenen Szenarien eingesetzt werden können.

## Lernziele

In diesem Kapitel lernen Sie unter anderem

- wie Zeichenketten in Zahlen umgewandelt werden können,
- wie Objekte kopiert werden können,
- wie Objekte in sogenannten Containern verwaltet werden können,
- wie einfache Performance-Messungen durchgeführt werden können,
- wie Objekte von Klassen erzeugt werden können, deren Existenz zum Zeitpunkt der Entwicklung des nutzenden Programms noch nicht bekannt sein müssen,
- wie Arrays komfortabel sortiert und durchsucht werden können,
- wie mathematische Funktionen genutzt werden können und
- wie mit Datums- und Zeitangaben gearbeitet werden kann.

## 15.1 Wrapper-Klassen für einfache Datentypen

Zu jedem einfachen Datentyp gibt es eine sogenannte *Wrapper-Klasse* (*Hüllklasse*), deren Objekte Werte dieses Datentyps speichern. Über diesen Weg können dann einfache Werte als Objekte angesprochen werden.

Wrapper-Klassen bieten eine Reihe von nützlichen Methoden. So können z. B. Zeichenketten in Werte des entsprechenden einfachen Datentyps umgewandelt werden.

**Tabelle 15-1:** Wrapper-Klassen

Einfacher Datentyp	Wrapper-Klasse
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_15](https://doi.org/10.1007/978-3-658-43574-5_15).

## Erzeugung von Wrapper-Klassen-Instanzen

Zu jeder Wrapper-Klasse gibt es die Klassenmethode `valueOf`, mit der eine Instanz auf Basis eines Wertes vom einfachen Datentyp erzeugt werden kann. Instanzen von Wrapper-Klassen sind nach ihrer Erzeugung nicht mehr veränderbar.

Beispiel:

```
Integer intObj = Integer.valueOf(4711);
```

Zu jeder Wrapper-Klasse mit Ausnahme von `Character` gibt es die Klassenmethode:

```
static Typ valueOf(String s)
```

Sie erzeugt aus dem von `s` repräsentierten Wert ein Objekt der entsprechenden Wrapper-Klasse und liefert es als Rückgabewert. Wenn `s` nicht umgewandelt werden kann, wird die nicht kontrollierte Ausnahme `java.lang.NumberFormatException` ausgelöst. `Typ` steht hier für die Klasse `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` bzw. `Double`.

Für die Klassen `Byte`, `Short`, `Integer` und `Long` gibt es noch die Variante:

```
static Typ valueOf(String s, int base)
```

`base` bezeichnet die Basis des Zahlensystems, das bei der Umwandlung zu Grunde gelegt wird.

Beispiel:

```
Integer.valueOf("101", 2) erzeugt ein Objekt, das die ganze Zahl 5 umhüllt.
```

Für `Character` gibt es die Methode

```
static Character valueOf(char c)
```

## Weitere Methoden

Für alle Wrapper-Klassen gibt es die Methoden `equals` und `toString`:

```
boolean equals(Object obj)
```

vergleicht mit `obj` und gibt `true` zurück, falls `obj` vom Typ der Wrapper-Klasse ist und denselben Wert umhüllt.

```
String toString()
```

liefert den Wert als Zeichenkette.

Die `Integer`-Methode

```
static int compare(int x, int y)
```

vergleicht zwei `int`-Werte numerisch und liefert den Wert 0 wenn `x == y`, einen Wert kleiner als 0, wenn `x < y` und einen Wert größer als 0, wenn `x > y`.

Analoge Methoden gibt es für die übrigen Wrapper-Klassen.

Für die Wrapper-Klassen Integer und Long gibt es die Methoden:

```
static String toBinaryString(typ i)
static String toOctalString(typ i)
static String toHexString(typ i)
```

Sie erzeugen eine Zeichenkettendarstellung von *i* als Dual-, Oktal- bzw. Hexadezimalzahl. Für *typ* ist *int* oder *long* einzusetzen.

Einige Character-Methoden:

```
static char toLowerCase(char c)
static char toUpperCase(char c)
```

wandeln Großbuchstaben in Kleinbuchstaben um bzw. umgekehrt.

```
static boolean isXxx(char c)
```

testet, ob *c* zu einer bestimmten Zeichenkategorie gehört. Für *xxx* kann hier

- Digit (Ziffer),
- ISOControl (Steuerzeichen),
- Letter (Buchstabe),
- LetterOrDigit (Buchstabe oder Ziffer),
- LowerCase (Kleinbuchstabe),
- SpaceChar (Leerzeichen),
- UpperCase (Großbuchstabe) oder
- WhiteSpace (Leerzeichen, Tabulator, Seitenvorschub, Zeilenende usw.)

eingesetzt werden.

## Rückgabe umhüllter Werte

Die folgenden Methoden liefern den durch das Objekt der jeweiligen Wrapper-Klasse umhüllten Wert:

```
boolean booleanValue()
char charValue()
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

Beispiel:

```
String s = "123.75";
double x = Double.valueOf(s).doubleValue();
```

Hier wird aus dem String *s* zunächst ein Objekt vom Typ *Double* erzeugt und zurückgeliefert, dann der entsprechende *double*-Wert des Objekts ermittelt und der Variablen *x* zugewiesen.

## Umwandlung von Zeichenketten in Zahlen

Die Umwandlung von Zeichenketten in Zahlen kann bequemer auch mit den Methoden

```
static byte parseByte(String s)
static short parseShort(String s)
static int parseInt(String s)
static long parseLong(String s)
static float parseFloat(String s)
static double parseDouble(String s)
```

der entsprechenden Klassen `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` durchgeführt werden. Wenn `s` nicht umgewandelt werden kann, wird die nicht kontrollierte Ausnahme `java.lang.NumberFormatException` ausgelöst.

## Numerische Konstanten

In jeder numerischen Wrapper-Klasse gibt es die Konstanten `MIN_VALUE` und `MAX_VALUE`, die den kleinsten bzw. größten Wert des Wertebereichs des entsprechenden Datentyps darstellen:

```
static final typ MIN_VALUE
static final typ MAX_VALUE
```

Für `typ` kann `byte`, `short`, `int`, `long`, `float` oder `double` eingesetzt werden.

Für Fließkommazahlen stellen die Konstanten den kleinsten bzw. größten *positiven* Wert dar.

Die Klassen `Float` und `Double` haben zusätzlich die Konstanten `NaN` (Not a Number), `NEGATIVE_INFINITY` und `POSITIVE_INFINITY`.

`NaN` stellt einen undefinierten Wert dar, wie er bei der Division `0.0 / 0.0` entsteht. `NEGATIVE_INFINITY` bzw. `POSITIVE_INFINITY` entsteht bei der Division negativer bzw. positiver Zahlen durch `0`.

Ganzzahlige Operationen, die aus diesem Wertebereich hinaus führen, brechen nicht ab. Sie werden auf dem darstellbaren Bereich ausgeführt und erzeugen dann durch Überlauf falsche Ergebnisse.

```
package wrapper;

public class WrapperTest {
    public static void main(String[] args) {
        // Zeichenkette in Zahl umwandeln
        String a = "10500";
        int x = Integer.parseInt(a);
        System.out.println("Zahl: " + x);

        // Zahl in Zeichenkette umwandeln
        double y = 123.76;
        String b = String.valueOf(y);
        System.out.println("String: " + b);
```

```
// Zahl als Dual-, Hexadezimal- bzw. Oktalzahl darstellen
System.out.println("Dualzahl: " + Integer.toBinaryString(61695));
System.out.println("Hexadezimalzahl: " + Integer.toHexString(61695));
System.out.println("Oktalzahl: " + Integer.toOctalString(61695));

// Zeichen testen
char c = '8';
System.out.println("Ziffer? " + Character.isDigit(c));

// Numerische Konstanten
System.out.println("byte Min: " + Byte.MIN_VALUE);
System.out.println("byte Max: " + Byte.MAX_VALUE);
System.out.println("short Min: " + Short.MIN_VALUE);
System.out.println("short Max: " + Short.MAX_VALUE);
System.out.println("int Min: " + Integer.MIN_VALUE);
System.out.println("int Max: " + Integer.MAX_VALUE);
System.out.println("long Min: " + Long.MIN_VALUE);
System.out.println("long Max: " + Long.MAX_VALUE);
System.out.println("float Min: " + Float.MIN_VALUE);
System.out.println("float Max: " + Float.MAX_VALUE);
System.out.println("double Min: " + Double.MIN_VALUE);
System.out.println("double Max: " + Double.MAX_VALUE);

System.out.println(0. / 0.);
System.out.println(1. / 0.);
System.out.println(-1. / 0.);

}

}
```

### Ausgabe des Programms:

```
Zahl: 10500
String: 123.76
Dualzahl: 1111000011111111
Hexadezimalzahl: f0ff
Oktalzahl: 170377
Ziffer? true
byte Min: -128
byte Max: 127
short Min: -32768
short Max: 32767
int Min: -2147483648
int Max: 2147483647
long Min: -9223372036854775808
long Max: 9223372036854775807
float Min: 1.4E-45
float Max: 3.4028235E38
double Min: 4.9E-324
double Max: 1.7976931348623157E308
NaN
Infinity
-Infinity
```

## Autoboxing

Der explizite Umgang mit Instanzen von Wrapper-Klassen ist umständlich.

Beispiel:

Umwandlung eines int-Werts in ein Integer-Objekt:

```
int i = 4711;
Integer iObj = Integer.valueOf(i);
```

Umwandlung eines Integer-Objekts in einen int-Wert:

```
i = iObj.intValue();
```

Ab Java-Version 5 steht das sogenannte *Autoboxing* zur Verfügung.

Beim Autoboxing wandelt der Compiler bei Bedarf einfache Datentypen in Objekte der entsprechenden Wrapper-Klasse um.

*Auto-Unboxing* bezeichnet den umgekehrten Vorgang: die automatische Umwandlung eines umhüllten Werts in den entsprechenden einfachen Wert.

Beispiel:

```
int i = 4711;
Integer iObj = i; // boxing
i = iObj;         // unboxing
```

Die automatische Umwandlung erfolgt bei der Zuweisung mittels =, aber auch bei der Übergabe von Argumenten an eine Methode.

```
package wrapper;

public class IntegerBox {
    private Integer value;

    public void setValue(Integer value) {
        this.value = value;
    }

    public Integer getValue() {
        return value;
    }
}

package wrapper;

public class BoxingTest {
    public static void main(String[] args) {
        Integer integer = 1234;    // boxing
        int i = integer;          // unboxing

        IntegerBox box = new IntegerBox();
        box.setValue(4711);
```

```
i = box.getValue();
System.out.println(i);

integer++;
System.out.println(integer);
integer += 10;
System.out.println(integer);
}

}
```

Ausgabe des Programms:

```
4711
1235
1245
```

## 15.2 Die Klasse Object

Die Klasse `java.lang.Object` ist die Wurzel der Klassenhierarchie. Jede Klasse, die nicht explizit mit `extends` von einer anderen Klasse abgeleitet ist, hat als Superklasse die Klasse `Object`. Damit erweitert jede Klasse direkt oder indirekt `Object`. Eine Referenzvariable vom Typ `object` kann demnach auf ein beliebiges Objekt verweisen.

`Object` enthält neben Methoden zur Unterstützung von *Multithreading* weitere allgemeine Methoden, von denen wir hier einige vorstellen.

### equals

```
boolean equals(Object obj)
```

liefert `true`, wenn das Objekt, für das die Methode aufgerufen wurde, und `obj` "gleich" sind. Die Klasse `Object` implementiert `equals` standardmäßig so, dass Gleichheit genau dann vorliegt, wenn `this == obj` gilt, es sich also um ein und dasselbe Objekt handelt.

Viele Klassen überschreiben diese Methode, um die Gleichheit von Objekten anwendungsspezifisch zu implementieren und nutzen dazu die Werte ihrer Instanzvariablen.

Die überschreibende Methode sollte die folgenden Eigenschaften haben:

- Für ein Objekt `x` gilt: `x.equals(x)` hat den Wert `true` (*Reflexivität*).
- Für Objekte `x` und `y` gilt: `x.equals(y)` hat den Wert `true` genau dann, wenn `y.equals(x)` den Wert `true` hat (*Symmetrie*).
- Für Objekte `x`, `y` und `z` gilt: Haben `x.equals(y)` und `y.equals(z)` beide den Wert `true`, so hat auch `x.equals(z)` den Wert `true` (*Transitivität*).
- Für ein Objekt `x` gilt: `x.equals(null)` hat den Wert `false`.
- Die zu vergleichenden Objekte müssen vom gleichen Typ sein.

Diese letzte Regel kann unterschiedlich streng ausgelegt werden:

1. Beide Objekte sind zur Laufzeit exakt vom gleichen Typ. Das kann mit der Methode `getClass` getestet werden (siehe später: Klasse `Class`).
2. Das Objekt einer Subklasse kann mit einem Objekt der Superklasse verglichen werden (mittels `instanceof`). Das ist sinnvoll, wenn die Subklasse nur Methoden der Superklasse überschreibt und keine eigenen Instanzvariablen hinzufügt.

Ein Muster zur Implementierung der `equals`-Methode findet man im folgenden Programm.

### **hashCode**

Ein *Hashcode* dient dazu, ein Objekt *möglichst eindeutig* zu identifizieren. Ändern sich Eigenschaften eines Objekts, so muss sich auch sein Hashcode ändern.

`int hashCode()`

liefert einen ganzzahligen Wert, den sogenannten *Hashcode*, der beispielsweise für die effiziente Speicherung von Objekten in *Hash-Tabellen* (wie `Hashtable` und `HashMap`) gebraucht wird.

Wurde in einer Klasse `equals` überschrieben, so sollte `hashCode` ebenfalls überschrieben werden, und zwar so, dass zwei "gleiche" Objekte auch den gleichen Hashcode haben. Zwei gemäß `equals` verschiedene Objekte dürfen aber den gleichen Hashcode haben.

Viele Entwicklungsumgebungen ermöglichen die Generierung von `equals` und `hashCode`.

In der Klasse `Konto` (siehe unten) sind die beiden geerbten Methoden `equals` und `hashCode` neu implementiert. Zwei `Konto`-Objekte sind genau dann gleich, wenn ihre Kontonummern (`id`) gleich sind. Der Hashcode ist der `int`-Wert der Kontonummer.

```
package object;

public class Konto {
    private int id;
    private double saldo;
    private Kunde kunde;

    public Konto() {
    }

    public Konto(int id, double saldo) {
        this.id = id;
        this.saldo = saldo;
    }
}
```

```
public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public void setSaldo(double saldo) {
    this.saldo = saldo;
}

public double getSaldo() {
    return saldo;
}

public void setKunde(Kunde kunde) {
    this.kunde = kunde;
}

public Kunde getKunde() {
    return kunde;
}

public void add(double betrag) {
    saldo += betrag;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Konto)) return false;
    return id == ((Konto) o).id;
}

@Override
public int hashCode() {
    return id;
}
}

package object;

public class Kunde {
    private String name;
    private String adresse;

    public Kunde() {
    }

    public Kunde(String name, String adresse) {
        this.name = name;
        this.adresse = adresse;
    }
}
```

```

public Kunde(Kunde other) {
    this.name = other.name;
    this.adresse = other.adresse;
}

public void setName(String name) {
    this.name = name;
}

public String getName() {
    return name;
}

public void setAdresse(String adresse) {
    this.adresse = adresse;
}

public String getAdresse() {
    return adresse;
}
}

package object;

public class Test {
    public static void main(String[] args) {
        Kunde kunde = new Kunde("Hugo Meier", "Hauptstr. 12, 40880 Ratingen");

        Konto konto1 = new Konto(4711, 10000.);
        konto1.setKunde(kunde);

        Konto konto2 = new Konto(4811, 0.);

        System.out.println("Objekt konto1 gleicht Objekt konto2: "
            + konto1.equals(konto2));
        System.out.println("Hashcode von Objekt kto1: " + konto1.hashCode());
        System.out.println("Hashcode von Objekt kto2: " + konto2.hashCode());
    }
}

```

Ausgabe des Programms:

```

Objekt konto1 gleicht Objekt konto2: false
Hashcode von Objekt kto1: 4711
Hashcode von Objekt kto2: 4811

```

Mitunter sind die folgenden statischen Methoden der Klasse `java.util.Objects` hilfreich:

```

static boolean equals(Object a, Object b)
static int hashCode(Object o)
static int hash(Object... values)

```

`equals` gibt `true` zurück, wenn die Argumente gleich sind, andernfalls `false`.

`hashCode` gibt den Hashcode eines Nicht-Null-Arguments und 0 für ein Null-Argument zurück.

`hash` generiert einen Hashcode für eine Folge von Eingabewerten. Diese Methode ist nützlich, um einen Hashcode für Objekte zu erzeugen, die mehrere Felder enthalten.

## Objekte kopieren

Die `Object`-Methode

```
protected Object clone() throws CloneNotSupportedException
```

gibt eine "Kopie" (*Klon*) des Objekts zurück, für das die Methode aufgerufen wurde, indem sie alle Instanzvariablen des neuen Objekts mit den Werten der entsprechenden Variablen des ursprünglichen Objekts initialisiert.

Klassen, die das Klonen anbieten wollen, müssen die Methode `clone` der Klasse `Object` als `public`-Methode überschreiben und das "leere" Interface `java.lang.Cloneable`, das weder Methoden noch Konstanten deklariert (ein sogenanntes Markierungs-Interface), implementieren (siehe Klasse `Konto` unten).

Wird `clone` für ein Objekt aufgerufen, dessen Klasse `Cloneable` nicht implementiert, wird die kontrollierte Ausnahme `java.lang.CloneNotSupportedException` ausgelöst.

In einer Subklasse, die `clone` überschreibt, kann diese Ausnahme auch bewusst ausgelöst werden, um anzudeuten, dass das Klonen nicht unterstützt wird.

Im folgenden Beispiel ist die Klasse `Konto` um die Methode `clone` ergänzt worden. `Konto` implementiert das Interface `Cloneable`.

```
package clone1;

public class Konto implements Cloneable {
    ...

    @Override
    public Konto clone() throws CloneNotSupportedException {
        return (Konto) super.clone();
    }
}

package clone1;

public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Kunde kunde = new Kunde("Hugo Meier", "Hauptstr. 12, 40880 Ratingen");

        Konto konto1 = new Konto(4711, 10000.);
```

```

konton.setKunde(kunde);

Konto konton2 = konton.clone();

System.out.println("VORHER");
System.out.println(konton2.getId());
System.out.println(konton2.getSaldo());
System.out.println(konton2.getKunde().getName());
System.out.println(konton2.getKunde().getAdresse());

kunde.setAdresse("Hauptstr. 42, 40880 Ratingen");

System.out.println();
System.out.println("NACHHER");
System.out.println(konton2.getId());
System.out.println(konton2.getSaldo());
System.out.println(konton2.getKunde().getName());
System.out.println(konton2.getKunde().getAdresse());
}
}

```

### Ausgabe des Programms:

```

VORHER
4711
10000.0
Hugo Meier
Hauptstr. 12, 40880 Ratingen

NACHHER
4711
10000.0
Hugo Meier
Hauptstr. 42, 40880 Ratingen

```

### Flache Kopie

Der Test zeigt, dass zwar ein Klon des Objekts konton erzeugt wird, dass aber die Instanzvariablen kunde des Originals und des Klons beide dasselbe Kunde-Objekt referenzieren. Es wurde demnach eine *flache Kopie* erzeugt.

### Tiefe Kopie

Sogenannte *tiefe Kopien* müssen auch die referenzierten Objekte berücksichtigen.

Die `clone`-Methode der folgenden Klasse Konto erzeugt mit Hilfe des Konstruktors `Kunde(Kunde other)` der Klasse Kunde ein neues Kunde-Objekt und weist es der Instanzvariablen kunde des Klons zu.

```

package clone2;

public class Konto implements Cloneable {

```

```
...
@Override
public Konto clone() throws CloneNotSupportedException {
    Konto k = (Konto) super.clone();
    k.kunde = new Kunde(kunde);
    return k;
}

package clone2;

public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Kunde kunde = new Kunde("Hugo Meier", "Hauptstr. 12, 40880 Ratingen");

        Konto konto1 = new Konto(4711, 10000.);
        konto1.setKunde(kunde);

        Konto konto2 = konto1.clone();

        System.out.println("VORHER");
        System.out.println(konto2.getId());
        System.out.println(konto2.getSaldo());
        System.out.println(konto2.getKunde().getName());
        System.out.println(konto2.getKunde().getAdresse());

        kunde.setAdresse("Hauptstr. 42, 40880 Ratingen");

        System.out.println();
        System.out.println("NACHHER");
        System.out.println(konto2.getId());
        System.out.println(konto2.getSaldo());
        System.out.println(konto2.getKunde().getName());
        System.out.println(konto2.getKunde().getAdresse());
    }
}
```

Ausgabe des Programms:

```
VORHER
4711
10000.0
Hugo Meier
Hauptstr. 12, 40880 Ratingen

NACHHER
4711
10000.0
Hugo Meier
Hauptstr. 12, 40880 Ratingen
```

Hier wird in der Methode `clone` der sogenannte *Kopierkonstruktor* der Klasse `Kunde` verwendet.

Genauso wie für die Klasse `Konto` kann man auch für die Klasse `Kunde` das Interface `Cloneable` implementieren und die `clone`-Methode der Klasse `Konto` dann wie folgt realisieren (siehe Paket `clone3` im Begleitmaterial):

```
@Override
public Konto clone() throws CloneNotSupportedException {
    Konto k = (Konto) super.clone();
    k.kunde = kunde.clone();
    return k;
}
```

### 15.3 Die Klassen `Vector`, `Hashtable` und `Properties`

*Container* sind Datenstrukturen, in denen man beliebige Objekte aufbewahren kann. Sie sind so organisiert, dass ein effizienter Zugriff auf die abgelegten Objekte möglich ist.

Die Zugriffsmethoden der hier besprochenen Klassen `Vector`, `Hashtable` und `Properties` sind *synchronisiert*, d. h. sie gewährleisten einen ungestörten parallelen Zugriff mehrerer Threads auf die Container-Elemente.

Das Paket `java.util` enthält daneben eine Reihe von Interfaces und Klassen, die ein leistungsfähiges Framework für Container bereitstellen: das *Collections Framework*.

Diese Container können mit ihrem Elementtyp parametrisiert werden, sodass nur Objekte des vorgegebenen Typs abgelegt werden können. In dem hier vorliegenden Kapitel gehen wir auf diese Möglichkeit nicht ein. Diesbezügliche Warnungen des Compilers können also ignoriert werden.

#### Die Klasse `Vector`

Die Klasse `java.util.Vector` implementiert eine dynamisch wachsende Reihung von Elementen des Typs `Object` (im Folgenden kurz "Vektor" genannt), auf die über einen Index zugegriffen werden kann.

Jeder der folgenden *Konstruktoren* erzeugt einen leeren Container. Der interne Speicherplatz wird zunächst für eine bestimmte Anzahl von Elementen bereitgestellt. Die Anfangskapazität (Anzahl Elemente) und der Zuwachswert, um den im Bedarfsfall die Kapazität erhöht werden soll, können vorgegeben werden, sonst werden Standardwerte angenommen.

```
Vector()
Vector(int initialCapacity)
Vector(int initialCapacity, int capacityIncrement)
```

Die Klasse `Vector` implementiert das Interface `java.util.List` des *Collections Framework*.

Bei einigen der im Folgenden aufgeführten Methoden ist die `List`-Methode mit der identischen Funktionalität beigefügt (siehe "Entspricht").

`Object clone()`

erzeugt einen Klon dieses Vektors als *flache Kopie*, d. h. die Elemente des Vektors werden nicht geklont.

`int size()`

liefert die aktuelle Anzahl der Elemente des Vektors.

`boolean isEmpty()`

liefert `true`, falls der Vektor kein Element enthält.

`void addElement(Object obj)`

hängt `obj` an das Ende des Vektors.

*Entspricht:* `boolean add(Object obj)`

`void insertElementAt(Object obj, int i)`

fügt `obj` an der Position `i` in den Vektor ein. Das erste Element des Vektors hat die Position `0`. Die Elemente, die sich bisher an dieser bzw. einer dahinter liegenden Position befanden, werden um eine Position zum Ende des Vektors hin verschoben.

*Entspricht:* `void add(int i, Object obj)`

`void setElementAt(Object obj, int i)`

ersetzt das Element an der Position `i` durch `obj`.

*Entspricht:* `Object set(int i, Object obj)`

`void removeElementAt(int i)`

entfernt das Element an der Position `i`. Die folgenden Elemente werden um eine Position nach vorne verschoben.

*Entspricht:* `Object remove(int i)`

`boolean removeElement(Object obj)`

entfernt das erste Element `obj` und liefert `true`. Die folgenden Elemente werden um eine Position nach vorne verschoben. Enthält der Vektor kein Element `obj`, wird `false` zurückgegeben. Es wird die Methode `boolean equals(Object o)` benutzt, um `obj` mit den Elementen des Vektors zu vergleichen.

*Entspricht:* `boolean remove(Object obj)`

`void removeAllElements()`

entfernt alle Elemente aus dem Vektor.

*Entspricht:* `void clear()`

`Object firstElement()`

liefert das erste Element des Vektors.

`Object lastElement()`

liefert das letzte Element des Vektors.

```

Object elementAt(int i)
    liefert das Element an der Position i.
    Entspricht: Object get(int i)

void copyInto(Object[] array)
    kopiert die Elemente des Vektors in das angegebene Array. Dabei wird das i-te
    Element des Vektors in das i-te Element des Arrays kopiert.

package container;

import object.Kunde;

import java.util.Vector;

public class VectorTest1 {
    public static void main(String[] args) {
        Vector kunden = new Vector();

        kunden.add(new Kunde("Hugo Meier", "Hauptstr. 12, 40880 Ratingen"));
        kunden.add(new Kunde("Otto Schmitz", "Dorfstr. 5, 40880 Ratingen"));
        kunden.add(0, new Kunde("Willi Peters", "Hauptstr. 22, 40880 Ratingen"));

        int size = kunden.size();
        for (int i = 0; i < size; i++) {
            Kunde k = (Kunde) kunden.get(i);
            System.out.println(k.getName() + ", " + k.getAdresse());
        }

        System.out.println();
        for (Object obj : kunden) {
            Kunde k = (Kunde) obj;
            System.out.println(k.getName() + ", " + k.getAdresse());
        }
    }
}

```

Wie bei Arrays kann auch hier die `foreach`-Schleife genutzt werden.

Ausgabe des Programms:

Willi Peters, Hauptstr. 22, 40880 Ratingen

Hugo Meier, Hauptstr. 12, 40880 Ratingen

Otto Schmitz, Dorfstr. 5, 40880 Ratingen

Willi Peters, Hauptstr. 22, 40880 Ratingen

Hugo Meier, Hauptstr. 12, 40880 Ratingen

Otto Schmitz, Dorfstr. 5, 40880 Ratingen

## Enumeration

Das Interface `java.util Enumeration` kann verwendet werden, um alle Elemente einer Aufzählung nacheinander zu durchlaufen. Die Schnittstelle deklariert die beiden folgenden abstrakten Methoden:

```
boolean hasMoreElements()
```

liefert true, wenn noch weitere Elemente vorliegen.

```
Object nextElement()
```

liefert das nächste Element. Falls kein Element mehr existiert, wird die nicht kontrollierte Ausnahme `java.util.NoSuchElementException` ausgelöst.

Die `Vector`-Methode

```
Enumeration elements()
```

erzeugt ein Objekt vom Typ `Enumeration` für alle Elemente des Vektors.

```
package container;

import object.Kunde;

import java.util.Enumeration;
import java.util.Vector;

public class VectorTest2 {
    public static void main(String[] args) {
        Vector kunden = new Vector();

        kunden.add(new Kunde("Hugo Meier", "Hauptstr. 12, 40880 Ratingen"));
        kunden.add(new Kunde("Otto Schmitz", "Dorfstr. 5, 40880 Ratingen"));
        kunden.add(0, new Kunde("Willi Peters", "Hauptstr. 22, 40880 Ratingen"));

        Enumeration e = kunden.elements();
        while (e.hasMoreElements()) {
            Kunde k = (Kunde) e.nextElement();
            System.out.println(k.getName() + ", " + k.getAdresse());
        }
    }
}
```

## Die Klasse `Hashtable`

Objekte der Klasse `java.util.Hashtable` ermöglichen die Speicherung von Datenpaaren aus einem *Schlüssel* und einem zugeordneten *Wert* sowie den effizienten Zugriff auf den Wert über den Schlüssel. Eine solche Struktur wird üblicherweise als zweispaltige Tabelle dargestellt.

Die Klassen der Objekte, die in die Tabelle als Schlüssel eingetragen werden sollen, müssen die Methoden `equals` und `hashCode` in geeigneter Form implementieren.

Der Eintrag und der Zugriff auf Schlüssel erfolgt intern mit Hilfe der Methode `equals`. Zwei Schlüssel werden als identisch angesehen, wenn sie gemäß `equals` gleich sind. Die Methode `hashCode` der Schlüssel wird intern verwendet, um den Speicherplatz in der Tabelle zu bestimmen.

Ein Objekt der Klasse `Hashtable` kann mit Hilfe des Konstruktors

`Hashtable()`

angelegt werden.

`Object clone()`

erzeugt einen Klon dieser Tabelle als *flache Kopie*, d. h. Schlüssel und Werte werden nicht geklont.

`int size()`

liefert die Anzahl der Schlüssel in der Tabelle.

`boolean isEmpty()`

liefert `true`, wenn die Tabelle keine Einträge enthält.

`Object put(Object key, Object value)`

fügt das Paar (`key, value`) in die Tabelle ein. `key` und `value` dürfen nicht `null` sein. Falls `key` bisher noch nicht in der Tabelle eingetragen ist, liefert die Methode `null` zurück, ansonsten den alten zugeordneten Wert, der nun durch den neuen Wert ersetzt ist.

`Object remove(Object key)`

entfernt den Schlüssel `key` und seinen Wert. Der zugeordnete Wert wird zurückgegeben. Ist `key` nicht in der Tabelle vorhanden, wird `null` zurückgegeben.

`void clear()`

entfernt alle Einträge aus der Tabelle.

`Object get(Object key)`

liefert den dem Schlüssel `key` zugeordneten Wert. Ist der Schlüssel nicht in der Tabelle eingetragen, wird `null` zurückgegeben.

`boolean containsValue(Object value)`

liefert `true`, falls `value` in der Tabelle als Wert vorkommt.

`boolean containsKey(Object key)`

liefert `true`, falls `key` in der Tabelle als Schlüssel vorkommt.

`Enumeration elements()`

liefert eine Aufzählung aller Werte der Tabelle.

`Enumeration keys()`

liefert eine Aufzählung aller Schlüssel der Tabelle.

```
package container;
```

```
import java.util.Enumeration;  
import java.util.Hashtable;
```

```
public class HashtableTest {  
    public static void main(String[] args) {  
        Hashtable h = new Hashtable();
```

```

h.put("Willi Franken", "willi.franken@fh-xxx.de");
h.put("Hugo Meier", "hugo.meier@abc.de");
h.put("Otto Schmitz", "otto.schmitz@xyz.de");
h.put("Sabine Moll", "sabine.moll@fh-xxx.de");

System.out.println("NAME -> E-MAIL-ADRESSE");
Enumeration keys = h.keys();
while (keys.hasMoreElements()) {
    String key = (String) keys.nextElement();
    String value = (String) h.get(key);
    System.out.println(key + " -> " + value);
}

System.out.println();
System.out.println("E-MAIL-ADRESSEN");
Enumeration elements = h.elements();
while (elements.hasMoreElements()) {
    String value = (String) elements.nextElement();
    System.out.println(value);
}
}
}
}

```

### Ausgabe des Programms:

```

NAME -> E-MAIL-ADRESSE
Hugo Meier -> hugo.meier@abc.de
Otto Schmitz -> otto.schmitz@xyz.de
Sabine Moll -> sabine.moll@fh-xxx.de
Willi Franken -> willi.franken@fh-xxx.de

E-MAIL-ADRESSEN
hugo.meier@abc.de
otto.schmitz@xyz.de
sabine.moll@fh-xxx.de
willi.franken@fh-xxx.de

```

## Die Klasse Properties

Die Klasse `java.util.Properties` ist eine Subklasse der Klasse `Hashtable`. Als Schlüssel und Werte sind nur Strings erlaubt. Es existieren Methoden zum Laden und Speichern aus bzw. in Dateien. Ein Objekt dieser Klasse wird auch als *Property-Liste* bezeichnet.

```

Properties()
Properties(Properties defaults)

```

erzeugen eine leere Property-Liste. Ist `defaults` angegeben, so wird in dem Fall, dass der Schlüssel in der Liste nicht gefunden wird, in der Liste `defaults` gesucht.

```
String getProperty(String key)
```

liefert den dem Schlüssel `key` zugeordneten Wert oder `null`, wenn der Schlüssel nicht gefunden wurde.

---

```
String getProperty(String key, String defaultValue)
    liefert den dem Schlüssel key zugeordneten Wert oder den Wert defaultValue,
    wenn der Schlüssel nicht gefunden wurde.

Object setProperty(String key, String value)
    entspricht der Hashtable-Methode put.

Enumeration propertyNames()
    liefert ein Enumeration-Objekt, mit dem alle Schlüssel der Liste aufgelistet
    werden können.
```

Property-Listen können in Dateien gespeichert und aus diesen geladen werden.

```
void store(OutputStream out, String comments) throws java.io.IOException
    schreibt die Liste in den java.io.OutputStream out. Als Kommentar wird
    comments in die Ausgabedatei geschrieben. store speichert nicht die Einträge
    aus der Default-Liste.

void load(InputStream in) throws java.io.IOException
    lädt die Einträge aus dem java.io.InputStream in.
```

store und load erzeugt bzw. erwartet ein spezielles Format für den Dateinhalt  
(siehe folgendes Beispiel).<sup>1</sup>

Datei *properties.txt*:

```
#Beispiel Version 1
Durchmesser=150.0
Gewicht=5.0
Farbe=rot
Hoehe=100.0
```

Zeilen werden durch # auf Kommentar gesetzt.

Das folgende Programm lädt und speichert eine Property-Liste im Dateisystem.<sup>2</sup>

```
package container;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Properties;

public class PropertiesTest1 {
    public static void main(String[] args) throws IOException {
        Properties p = new Properties();
```

---

1 InputStream und OutputStream werden in Kapitel 24 behandelt.

2 Die Konstrukte, die die Dateiverarbeitung betreffen, werden im Kapitel 24 ausführlich behandelt.

```

FileInputStream in = new FileInputStream("properties.txt");
p.load(in);
in.close();

Enumeration keys = p.propertyNames();
while (keys.hasMoreElements()) {
    String key = (String) keys.nextElement();
    String value = p.getProperty(key);
    System.out.println(key + " = " + value);
}

p.put("Gewicht", "6.5");
p.put("Farbe", "gelb");

FileOutputStream out = new FileOutputStream("properties2.txt");
p.store(out, "Beispiel Version 2");
out.close();
}
}

```

Es wird auch eine XML-basierte Version von load und store unterstützt:

```

void loadFromXML(InputStream in) throws java.io.IOException,
    java.util.InvalidPropertiesFormatException

void storeToXML(OutputStream out, String comments)
    throws java.io.IOException

```

Datei *properties.xml*:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>Beispiel Version 1</comment>
    <entry key="Durchmesser">150.0</entry>
    <entry key="Gewicht">5.0</entry>
    <entry key="Farbe">rot</entry>
    <entry key="Hoehe">100.0</entry>
</properties>

```

```

package container;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Properties;

public class PropertiesTest2 {
    public static void main(String[] args) throws IOException {
        Properties p = new Properties();

        FileInputStream in = new FileInputStream("properties.xml");
        p.loadFromXML(in);
    }
}

```

```

        in.close();

        Enumeration keys = p.propertyNames();
        while (keys.hasMoreElements()) {
            String key = (String) keys.nextElement();
            String value = p.getProperty(key);
            System.out.println(key + " = " + value);
        }

        p.put("Gewicht", "6.5");
        p.put("Farbe", "gelb");

        FileOutputStream out = new FileOutputStream("properties2.xml");
        p.storeToXML(out, "Beispiel Version 2");
        out.close();
    }
}

```

## 15.4 Die Klasse System

Die Klasse `java.lang.System` enthält wichtige Methoden zur Kommunikation mit dem Betriebssystem. Es können keine Objekte vom Typ `System` erzeugt werden. Alle Methoden von `System` sind Klassenmethoden.

### Standarddatenströme

`System` enthält folgende Klassenvariablen:

```

public static final InputStream in
public static final PrintStream out
public static final PrintStream err

```

Hierbei handelt es sich um die *Standarddatenströme* zur Eingabe, Ausgabe und Fehlerausgabe.

Die Klassen `java.io.InputStream` und `java.io.PrintStream` werden im Kapitel 24 behandelt.

Mit `System.out.println(...)` wird die Methode `println` der Klasse `PrintStream` für das Objekt `out` aufgerufen, um Zeichenketten auszugeben.

Mit Hilfe der Methode `nextLine` der Klasse `java.util.Scanner` können Zeichenketten von der Tastatur (`System.in`) eingelesen werden. Die Eingabe muss mit der Eingabetaste (Return) abgeschlossen werden:

```

Scanner sc = new Scanner(System.in);
String line = sc.nextLine();
sc.close();

```

### Aktuelle Systemzeit

```
static long currentTimeMillis()
```

liefert die Anzahl Millisekunden, die seit dem 1.1.1970 00:00:00 Uhr UTC (Universal Time Coordinated) vergangen sind. 1 Sekunde entspricht 1000 Millisekunden.

```
static long nanoTime()
```

gibt den aktuellen Wert in Nanosekunden zurück. 1 Sekunde entspricht 1.000.000.000 Nanosekunden.

Um die Performance einer in `testMethod` implementierten Berechnung zu messen, umklammert man den Aufruf von `testMethod` mit Aufrufen von `System.nanoTime()` und ermittelt die Differenz zwischen den beiden Werten. Um evtl. Störeinflüssen zu begegnen, wird eine bestimmte Anzahl von Iterationen genutzt und dann der Durchschnittswert errechnet.

Hier ein Muster zur einfachen Performance-Messung:

```
package system;

public class PerformanceTest {
    private static final int MAX_ITERATIONS = 100;

    public static void testMethod() {
        // ...
    }

    public static void main(String[] args) {
        long start = System.nanoTime();
        for (int i = 0; i < MAX_ITERATIONS; i++) {
            testMethod();
        }
        long end = System.nanoTime();
        System.out.println((end - start) / MAX_ITERATIONS + " ns");
    }
}
```

### Arrays kopieren

```
static void arraycopy(Object src, int srcPos, Object dst, int dstPos,
                      int length)
```

kopiert `length` Elemente aus dem Array `src` in das Array `dst`, jeweils ab der Position `srcPos` bzw. `dstPos`.

### Programm beenden

```
static void exit(int status)
```

beendet das laufende Programm. `status` wird an den Aufrufer des Programms (z. B. ein Shell-Skript) übergeben. Üblicherweise signalisiert 0 ein fehlerfreies Programmende.

## Umgebungsvariablen

`static String getenv(String name)`

liefert den Wert der Umgebungsvariablen `name` des Betriebssystems.

## System Properties

**Tabelle 15-2:** Einige Java-Systemeigenschaften

Property	Bedeutung
<code>file.separator</code>	Dateipfadtrennzeichen
<code>java.class.path</code>	aktueller Klassenpfad
<code>java.class.version</code>	Version der Klassenbibliothek
<code>java.home</code>	Installationsverzeichnis
<code>java.vendor</code>	Herstellername
<code>java.vendor.url</code>	URL des Herstellers
<code>java.version</code>	Java-Versionsnummer
<code>line.separator</code>	Zeilentrennzeichen
<code>os.arch</code>	Betriebssystemarchitektur
<code>os.name</code>	Betriebssystemname
<code>os.version</code>	Betriebssystemversion
<code>path.separator</code>	Trennzeichen in PATH-Angaben
<code>user.dir</code>	aktuelles Arbeitsverzeichnis
<code>user.home</code>	Home-Verzeichnis
<code>user.name</code>	Anmeldename

`static Properties getProperties()`

liefert die Java-Systemeigenschaften (*System Properties*) der Plattform als Property-Liste.

`static String getProperty(String key)`

liefert den Wert der Java-Systemeigenschaft mit dem Namen `key` oder `null`, wenn keine Eigenschaft mit diesem Namen gefunden wurde.

`static String getProperty(String key, String defaultValue)`

liefert den Wert der Java-Systemeigenschaft mit dem Namen `key` oder den Wert `defaultValue`, wenn keine Eigenschaft mit diesem Namen gefunden wurde.

```
static String setProperty(String key, String value)
```

setzt die Java-Systemeigenschaft key auf den Wert value und liefert den alten Wert dieser Eigenschaft oder null, falls dieser nicht existiert.

Systemeigenschaften können auch beim Aufruf eines Programms als Option gesetzt werden:

```
java -Dproperty=value ...
```

Folgendes Programm fragt Java-Systemeigenschaften ab bzw. listet sie alle auf.

```
package system;

import java.util.Enumeration;
import java.util.Properties;
import java.util.Scanner;

public class SystemTest {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (true) {
            System.out.print("Key: ");
            String key = sc.nextLine();
            if (key.isEmpty())
                break;
            System.out.println(System.getProperty(key));
        }
        sc.close();

        Properties p = System.getProperties();
        Enumeration e = p.propertyNames();
        while (e.hasMoreElements()) {
            String key = (String) e.nextElement();
            System.out.println(key + " = " + System.getProperty(key));
        }
    }
}
```

## 15.5 Die Klasse Class

Jeder Typ (Klasse, Interface, Record, enum-Aufzählung, Array, einfacher Datentyp, void) einer laufenden Java-Anwendung wird durch ein Objekt der Klasse `Class` beschrieben.

### Typinformationen zur Laufzeit

`java.lang.Class` hat keine `public`-Konstruktoren. Objekte vom Typ `Class` entstehen automatisch, wenn Klassen geladen werden.

Zu jedem Typ existiert genau ein `Class`-Objekt.

`Class`-Objekte ermöglichen es, zur Laufzeit Informationen über Klassen zu beschaffen und Objekte von beliebigen Klassen zu erzeugen, deren Existenz zur Entwicklungszeit des Programms noch nicht bekannt war.

Die `Object`-Methode

```
getClass()
```

liefert das `Class`-Objekt zu dem Objekt, für das die Methode aufgerufen wurde.

Beispiel:

```
String s = "";
Class c = s.getClass();
```

c ist das `Class`-Objekt für die Klasse `String`.

### Klassenliterale

Für die einfachen Datentypen, Arrays und `void` gibt es Konstanten des Typs `Class`. Diese werden mit dem Typnamen und Suffix `.class` gebildet, z. B. `int.class`, `int[].class`, `void.class`.

Ebenso erhält man für eine Klasse, ein Interface, einen Record oder eine `enum`-Aufzählung A mit `A.class` eine Referenz auf das `Class`-Objekt von A.

Die statische `Class`-Methode `forName(String name)` liefert das `Class`-Objekt für die Klasse bzw. das Interface mit dem Namen `name`. Der Name muss vollständig spezifiziert sein, also z. B. `java.lang.String`.

Der Aufruf dieser Methode für eine Klasse mit Namen `name` führt zum Laden und Initialisieren dieser Klasse. Wird die Klasse nicht gefunden, so wird die kontrollierte Ausnahme `java.lang.ClassNotFoundException` ausgelöst.

```
String getName()
```

liefert den Namen des Typs, der vom `Class`-Objekt repräsentiert wird.

```
package klasse;

public class ClassTest1 {
    public static void main(String[] args) throws ClassNotFoundException {
        System.out.println(".".getClass().getName());
        System.out.println(String.class.getName());
        System.out.println(Class.forName("java.util.Vector").getName());
    }
}
```

Ausgabe des Programms:

```
java.lang.String
java.lang.String
java.util.Vector
```

### Einige Class-Methoden

`Class getClass()`

liefert das `Class`-Objekt für die Superklasse der Klasse, für deren `Class`-Objekt die Methode aufgerufen wurde. Repräsentiert dieses `Class`-Objekt die Klasse `Object`, ein Interface, einen einfachen Datentyp oder `void`, so wird `null` zurückgegeben.

`Class[] getInterfaces()`

liefert ein Array von `Class`-Objekten für Interfaces, die diejenige Klasse implementiert hat, für deren `Class`-Objekt die Methode aufgerufen wurde. Ähnliches gilt für ein abgeleitetes Interface.

`boolean isInterface()`

liefert `true`, wenn dieses `Class`-Objekt ein Interface repräsentiert.

`boolean isRecord()`

liefert `true`, wenn dieses `Class`-Objekt einen Record repräsentiert.

`boolean isArray()`

liefert `true`, wenn dieses `Class`-Objekt ein Array repräsentiert.

`boolean isPrimitive()`

liefert `true`, wenn dieses `Class`-Objekt einen einfachen Datentyp oder `void` repräsentiert.

`boolean isEnum()`

liefert `true`, wenn dieses `Class`-Objekt einen Aufzählungstyp repräsentiert.

### Ressourcen finden

`java.net.URL getResource(String name)`

Der Klassenlader, der die durch dieses `Class`-Objekt beschriebene Klasse lädt, benutzt dieselben Mechanismen, um auch die mit der Klasse gespeicherte Ressource `name` zu finden. Ressourcen können z. B. Texte oder Bilder sein.

Der *absolute Name* der zu suchenden Ressource wird wie folgt gebildet:

Beginnt `name` mit `'/'`, so wird der absolute Name aus den auf `'/'` folgenden Zeichen gebildet. Andernfalls wird der absolute Name aus dem Paketnamen der Klasse gebildet, wobei ein `'.'` durch ein Pfadtrennzeichen ersetzt wird.

Beispiel:

`de.test.persistence.ConnectionManager`

name	absoluter Name
<code>/dbparam.txt</code>	<code>dbparam.txt</code>
<code>dbparam.txt</code>	<code>de/test/persistence/dbparam.txt</code>

Ressourcen können auch in jar-Dateien zusammengefasst werden. Diese jar-Dateien müssen dann in den `CLASSPATH` eingebunden werden.

`getResource` gibt ein `URL`-Objekt für die Ressource zurück oder `null`, wenn die Ressource nicht gefunden wurde. Ein Objekt der Klasse `java.net.URL` repräsentiert einen *Uniform Resource Locator*. URLs werden später im Zusammenhang mit der Netzwerkkommunikation im Kapitel 30 behandelt.

In diesem Zusammenhang sei noch die Methode

```
java.io.InputStream getResourceAsStream(String name)
```

erwähnt, die ein `InputStream`-Objekt zum Lesen der vom Klassenlader gefundenen Ressource `name` liefert.

## Informationen über Klassen

Das folgende Beispielprogramm liefert Informationen über Klassen, deren Namen als Parameter beim Aufruf mitgegeben werden.

```
package klasse;

public class ClassTest2 {
    public static void main(String[] args) throws ClassNotFoundException {
        Class aClass = Class.forName(args[0]);
        System.out.println(aClass.getName());

        Class superclass = aClass.getSuperclass();
        if (superclass != null)
            System.out.println("Superklasse: " + superclass.getName());

        Class[] interfaces = aClass.getInterfaces();
        if (interfaces.length > 0) {
            System.out.println("implementierte Interfaces:");
            for (Class anInterface : interfaces) {
                System.out.println("    " + anInterface.getName());
            }
        }
    }
}
```

Beispiel (Aufruf im Projektverzeichnis):

```
java -cp out/production/kap15 klasse.ClassTest2 java.util.Vector3

java.util.Vector
Superklasse: java.util.AbstractList
implementierte Interfaces:
    java.util.List
    java.util.RandomAccess
    java.lang.Cloneable
    java.io.Serializable
```

---

3 Bei *IntelliJ IDEA* liegt der Bytecode standardmäßig im Verzeichnis `out/production/Projektname`

### Klassen dynamisch laden

Das folgende Programm zeigt, wie Klassen zur Laufzeit dynamisch geladen werden können, ohne dass ihre Namen im Quellcode genannt sind.

Die Klassen `Addition` und `Subtraktion` implementieren beide das Interface `Berechnung`. Die Klasse `Test` nutzt `Class`-Methoden, um ein Objekt zu erzeugen und hierfür die Schnittstellenmethode auszuführen.

```
package klasse;

public interface Berechnung {
    int berechne(int a, int b);
}

package klasse;

public class Addition implements Berechnung {
    public int berechne(int a, int b) {
        return a + b;
    }
}

package klasse;

public class Subtraktion implements Berechnung {
    public int berechne(int a, int b) {
        return a - b;
    }
}

package klasse;

public class Test {
    public static void main(String[] args) throws Exception {
        Class c = Class.forName(args[0]);
        Berechnung b = (Berechnung) c.getDeclaredConstructor().newInstance();
        System.out.println(b.berechne(10, 5));
    }
}
```

Der Aufruf von

```
c.getDeclaredConstructor().newInstance()
```

erzeugt ein neues Objekt der Klasse, die durch das `Class`-Objekt `c` repräsentiert wird. Es wird der parameterlosen Konstruktor der Klasse aufgerufen und eine Referenz auf das erzeugte Objekt zurückgegeben.

Aufrufbeispiele:

```
java -cp out/production/kap15 klasse.Test klasse.Addition
15

java -cp out/production/kap15 klasse.Test klasse.Subtraktion
5
```

## 15.6 Die Klasse Arrays

Die Klasse `java.util.Arrays` bietet Methoden zum komfortablen Arbeiten mit Arrays.

```
static boolean equals(Typ[] a, Typ[] b)
liefert true genau dann, wenn beide Arrays gleich lang sind und elementweise
gleich sind. Als Typ kann hier ein einfacher Datentyp oder Object eingesetzt
werden.

static void fill(Typ[] a, Typ val)
weist allen Elementen in a den Wert val zu. Als Typ kann hier ein einfacher
Datentyp oder Object eingesetzt werden.

static void fill(Typ[] a, int from, int to, Typ val)
weist allen Elementen in a ab Index from bis zum Index to - 1 den Wert val zu.
Als Typ kann hier ein einfacher Datentyp oder Object eingesetzt werden.

static void sort(Typ[] a)
sortiert die Elemente des Arrays a aufsteigend. Typ steht hier für einen einfachen
Datentyp mit Ausnahme von boolean oder für Object.

static void sort(Typ[] a, int from, int to)
sortiert die Elemente des Bereichs from bis to - 1. Typ steht hier für einen
einfachen Datentyp mit Ausnahme von boolean oder für Object.

static String toString(Typ[] a)
liefert die Zeichenkettendarstellung des Arrays a. Als Typ kann hier ein ein-
facher Datentyp oder Object eingesetzt werden.
```

### Comparable

Für das Sortieren im Fall von `Object` müssen alle Elemente des Arrays das Interface `java.lang.Comparable` mit der Methode

```
int compareTo(Object obj)
implementieren.
```

Je zwei Elemente `x` und `y` des Arrays müssen vergleichbar sein. Ein Aufruf von `x.compareTo(y)` muss einen negativen Wert, den Wert 0 oder einen positiven Wert liefern, je nachdem, ob `x` kleiner als `y`, `x` gleich `y` oder `x` größer als `y` ist.

`compareTo` sollte konsistent zu `equals` implementiert werden: `x.compareTo(y) == 0` liefert den gleichen Wert wie `x.equals(y)`.

`x.compareTo(null)` sollte eine `NullPointerException` auslösen.

Die Klasse `String` implementiert `Comparable` so, dass Strings lexikographisch miteinander verglichen werden.

### Binäre Suche

`static int binarySearch(Typ[] a, Typ key)`

durchsucht das Array `a` nach dem Wert `key` unter Verwendung des *Verfahrens der binären Suche*. Dazu müssen die Elemente in `a` aufsteigend sortiert sein. `Typ` steht für einen einfachen Datentyp mit Ausnahme von `boolean` oder für `Object`. Wird der Wert `key` gefunden, so wird sein Index zurückgegeben. Andernfalls ist der Rückgabewert negativ und zwar  $-i-1$ , wobei  $i$  der Index des ersten größeren Werts bzw. `a.length` (wenn alle Elemente in `a` kleiner als `key` sind) ist. Der Wert gibt also Aufschluss darüber, wo das fehlende Element einzufügen wäre. Der Rückgabewert ist  $\geq 0$  genau dann, wenn `key` gefunden wurde.

Das folgende Programm sortiert zum einen Zahlen, zum anderen Konten nach ihrer Kontonummer. Die Klasse `Konto` implementiert dazu das Interface `Comparable`.

```
package arrays;

public class Konto implements Comparable {
    private int id;
    private double saldo;

    public Konto() {
    }

    public Konto(int id, double saldo) {
        this.id = id;
        this.saldo = saldo;
    }

    public int getId() {
        return id;
    }

    public void setId(int nr) {
        id = nr;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

```
public void setSaldo(double betrag) {  
    saldo = betrag;  
}  
  
public void zahleEin(double betrag) {  
    saldo += betrag;  
}  
  
public void zahleAus(double betrag) {  
    saldo -= betrag;  
}  
  
@Override  
public String toString() {  
    return "Konto{" +  
        "id=" + id +  
        ", saldo=" + saldo +  
        '}';  
}  
  
@Override  
public boolean equals(Object o) {  
    if (this == o) return true;  
    if (!(o instanceof Konto)) return false;  
    return id == ((Konto) o).id;  
}  
  
@Override  
public int hashCode() {  
    return id;  
}  
}  
  
@Override  
public int compareTo(Object obj) {  
    return id - ((Konto) obj).id;  
}  
}  
  
package arrays;  
  
import java.util.Arrays;  
  
public class ArraysTest {  
    public static void main(String[] args) {  
        int[] z = { 9, 8, 7, 6, 5, 4, 2, 1, 0 };  
        Arrays.sort(z);  
        System.out.println(Arrays.toString(z));  
  
        int r = Arrays.binarySearch(z, 8);  
        System.out.println(r);  
        r = Arrays.binarySearch(z, 3);  
        System.out.println(r);  
  
        Konto k1 = new Konto(4711, 100);  
        Konto k2 = new Konto(6000, 200);  
        Konto k3 = new Konto(4044, 300);  
        Konto k4 = new Konto(1234, 400);  
        Konto[] konten = { k1, k2, k3, k4 };  
    }  
}
```

```

        Arrays.sort(konten);
        System.out.println(Arrays.toString(konten));
    }
}

```

Ausgabe des Programms:

```

[0, 1, 2, 4, 5, 6, 7, 8, 9]
7
-4
[Konto{id=1234, saldo=400.0}, Konto{id=4044, saldo=300.0},
Konto{id=4711, saldo=100.0}, Konto{id=6000, saldo=200.0}]

```

## 15.7 Mathematische Funktionen

Die Klasse `java.lang.Math` enthält die beiden `double`-Konstanten  $e$  (Eulersche Zahl  $e$ ) und  $\pi$  (Kreiskonstante  $\pi$ ) sowie grundlegende *mathematische Funktionen*, die als Klassenmethoden realisiert sind.

**Tabelle 15-3:** Einige Methoden der Klasse `Math`

Klassenmethode	Erläuterung
<code>Typ abs(Typ x)</code>	Absolutbetrag von $x$ . <code>Typ</code> steht für <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> .
<code>float signum(float x)</code> <code>double signum(double x)</code>	Vorzeichen: -1 für $x < 0$ , 0 für $x = 0$ , 1 für $x > 0$
<code>Typ min(Typ x, Typ y)</code>	Minimum von $x$ und $y$ . <code>Typ</code> steht für <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> .
<code>Typ max(Typ x, Typ y)</code>	Maximum von $x$ und $y$ . <code>Typ</code> steht für <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> .
<code>double ceil(double x)</code>	kleinste ganze Zahl größer oder gleich $x$
<code>double floor(double x)</code>	größte ganze Zahl kleiner oder gleich $x$
<code>int round(float x)</code> <code>long round(double x)</code>	ganze Zahl, die entsteht, wenn die Nachkommastellen von $x + 0.5$ abgeschnitten werden.
<code>double sqrt(double x)</code>	Quadratwurzel aus $x$
<code>double cbrt(double x)</code>	Kubikwurzel aus $x$
<code>double pow(double x, double y)</code>	Potenz $x^y$
<code>double hypot(double x, double y)</code>	Quadratwurzel aus $x^2 + y^2$
<code>double exp(double x)</code>	$e^x$
<code>double log(double x)</code>	Natürlicher Logarithmus von $x$
<code>double log10(double x)</code>	Logarithmus zur Basis 10 von $x$
<code>double sin(double x)</code>	Sinus von $x$

Klassenmethode	Erläuterung
double cos(double x)	Cosinus von x
double tan(double x)	Tangens von x
double sinh(double x)	Sinus hyberbolicus von x
double cosh(double x)	Cosinus hyberbolicus von x
double tanh(double x)	Tangens hyberbolicus von x
double asin(double x)	Arcussinus von x
double acos(double x)	Arcuscosinus von x
double atan(double x)	Arcustangens von x
double random()	Zufallszahl $\geq 0$ und $< 1$

```
package math;

public class MathTest {
    public static void main(String[] args) {
        double x = 3.5, y = 2.;
        double abstand = Math.hypot(x, y);
        System.out.println("Abstand: " + abstand);

        double radius = 2.;
        double flaeche = Math.PI * Math.pow(radius, 2);
        System.out.println("Fläche: " + flaeche);

        double anfangsBetrag = 5000.;
        double zinssatz = 7.5;
        double n = 10.;
        double endBetrag = anfangsBetrag * Math.pow(1. + zinssatz / 100., n);
        System.out.println("Endbetrag: " + endBetrag);

        System.out.println(Math.ceil(3.6));
        System.out.println(Math.floor(3.6));
        System.out.println(Math.round(3.6));
    }
}
```

### Ausgabe des Programms:

```
Abstand: 4.031128874149275
Fläche: 12.566370614359172
Endbetrag: 10305.157810823555
4.0
3.0
4
```

### DecimalFormat

Die Klasse `java.text.DecimalFormat` wird zur Darstellung von Dezimalzahlen genutzt.

```
DecimalFormat(String pattern)
```

erzeugt ein Objekt mit dem durch pattern vorgegebenen Format.

Die Zeichenkette pattern kann sich aus den folgenden Zeichen zusammensetzen:

- # Ziffer, führende Nullen werden nicht angezeigt
- 0 Ziffer, führende Nullen werden als 0 angezeigt
- .
- ,
- % Dezimalpunkt
- Symbol für Tausender-Gruppierung
- Darstellung als Prozentzahl (nach Multiplikation mit 100)

Als Präfix und Suffix des darzustellenden Wertes können beliebige Zeichen auftreten.

```
String format(double value)  
formatiert value gemäß der Vorgabe.
```

Beispiel:

```
DecimalFormat f = new DecimalFormat("###,##0.00 €");  
System.out.println(f.format(24522.4567));
```

Ausgabe:

```
24.522,46 €
```

## Random

Mit der Klasse `java.util.Random` können *Zufallszahlen* erzeugt werden.

Konstruktoren sind:

```
Random()  
Random(long seed)
```

seed liefert die Startbedingung für die Erzeugung der Zufallszahlen. Zwei Objekte, die mit dem gleichen Wert seed erzeugt werden, generieren gleiche Folgen von Zufallszahlen. Wird seed nicht übergeben, wird der Zufallszahlengenerator auf Basis der aktuellen Systemzeit initialisiert.

```
void setSeed(long seed)  
ändert den Startwert des Zufallszahlengenerators.
```

Die folgenden Methoden liefern gleichverteilte Zufallszahlen vom Typ `int`, `long`, `float` und `double`:

```
int nextInt()  
long nextLong()
```

```
float nextFloat()
double nextDouble()
```

In den beiden letzten Fällen sind die Zufallszahlen  $\geq 0$  und  $< 1$ .

```
int nextInt(int n)
    liefert eine gleichverteilte Zufallszahl  $\geq 0$  und  $< n$ .
double nextGaussian()
    liefert eine normalverteilte Zufallszahl mit dem Mittelwert 0 und der Standard-
    abweichung 1.
```

```
package math;

import java.util.Random;

public class RandomTest {
    public static void main(String[] args) {
        Random rand = new Random(1);
        for (int i = 0; i < 10; i++) {
            System.out.print(rand.nextInt(100) + " ");
        }
        System.out.println();

        rand.setSeed(2);
        for (int i = 0; i < 10; i++) {
            System.out.print(rand.nextInt(100) + " ");
        }
        System.out.println();

        rand = new Random();
        for (int i = 0; i < 10; i++) {
            System.out.print(rand.nextInt(100) + " ");
        }
    }
}
```

Ausgabe des Programms:

```
85 88 47 13 54 4 34 6 78 48
8 72 40 67 89 50 6 19 47 68
38 2 96 72 92 66 89 79 2 84
```

Die letzte Reihe von Zufallszahlen ist nicht reproduzierbar.

## **BigInteger**

Die Klasse `java.math.BigInteger` stellt ganze Zahlen mit beliebiger Stellenanzahl dar.

```
BigInteger(String s)
    erzeugt ein BigInteger-Objekt aus der dezimalen String-Darstellung s.
```

Es existieren die Konstanten

```
static final BigInteger ZERO  
static final BigInteger ONE  
static final BigInteger TWO  
static final BigInteger TEN
```

Einige Methoden:

`static BigInteger valueOf(long val)`  
liefert ein BigInteger-Objekt, dessen Wert val entspricht.

`BigInteger abs()`  
liefert den Absolutbetrag.

`BigInteger negate()`  
liefert die negative Zahl.

`BigInteger add(BigInteger val)`  
addiert this und val.

`BigInteger subtract(BigInteger val)`  
subtrahiert val von this.

`BigInteger divide(BigInteger val)`  
dividiert this durch val.

`BigInteger mod(BigInteger val)`  
liefert den Rest bei der Division von this durch val.

`BigInteger multiply(BigInteger val)`  
multipliziert this und val.

`String toString()`  
wandelt die Zahl in eine Zeichenkette um.

`boolean isProbablePrime(int p)`  
liefert true, wenn die Zahl mit einer Wahrscheinlichkeit größer als  $1 - 0.5^p$  eine Primzahl ist, sonst false.

Das nächste Programm demonstriert den Umgang mit großen Zahlen.

Ausgehend von einer zufälligen Zahl mit einer vorgegebenen Stellenanzahl wird jeweils die nächste Primzahl mit einer Wahrscheinlichkeit von mindestens  $1 - 0.5^{100}$  ermittelt. Große Primzahlen spielen z. B. bei der Verschlüsselung (*Public-Key-Verfahren*) eine wichtige Rolle.

```
package math;  
  
import java.math.BigInteger;  
import java.util.Random;  
  
public class Primzahlen {
```

```

// erzeugt eine zufällige n-stellige Zahl
public static BigInteger getZahl(int n) {
    Random r = new Random();
    StringBuilder s = new StringBuilder();

    s.append(1 + r.nextInt(9));
    for (int i = 1; i < n; i++) {
        s.append(r.nextInt(10));
    }

    return new BigInteger(s.toString());
}

// erzeugt ausgehend von start die nächste Primzahl > start
public static BigInteger nextPrimzahl(BigInteger start) {
    if (start.mod(BigInteger.TWO).equals(BigInteger.ZERO)) // gerade Zahl?
        start = start.add(BigInteger.ONE);
    else
        start = start.add(BigInteger.TWO);

    if (start.isProbablePrime(100))
        return start;
    else
        return nextPrimzahl(start); // rekursiver Aufruf
}

public static void main(String[] args) {
    int num = 60;
    if (args.length > 0)
        num = Integer.parseInt(args[0]);

    BigInteger start = getZahl(num);

    for (int i = 0; i < 10; i++) {
        start = nextPrimzahl(start);
        System.out.println(start);
    }
}
}

```

Das Programm erzeugt zehn 60-stellige Primzahlen (Beispiel):

```

750512869719538739792341585126777615584117113304267747669413
750512869719538739792341585126777615584117113304267747669439
750512869719538739792341585126777615584117113304267747669491
...

```

## 15.8 Datum und Uhrzeit

In diesem Kapitel werden zunächst die herkömmlichen Klassen `Date`, `TimeZone` und `GregorianCalendar` aus dem Paket `java.util` benutzt, um Datum und Uhrzeit anzugeben bzw. zu berechnen.

Mit Java-Version 8 wurde ein neues, umfangreiches *Date And Time API* eingeführt. Dies wird im Anschluss vorgestellt.

## Date

Ein Objekt der Klasse Date repräsentiert einen Zeitpunkt.

Konstruktoren sind:

`Date()`

erzeugt ein Objekt, das die aktuelle Systemzeit des Rechners repräsentiert.

`Date(long time)`

erzeugt ein Objekt, das den Zeitpunkt `time` Millisekunden nach dem 1.1.1970 00:00:00 Uhr GMT repräsentiert.

Methoden der Klasse Date:

`long getTime()`

liefert die Anzahl Millisekunden, die seit dem 1.1.1970 00:00:00 Uhr GMT vergangen sind.

`void setTime(long time)`

stellt das Date-Objekt so ein, dass es den Zeitpunkt `time` Millisekunden nach dem 1.1.1970 um 00:00:00 Uhr GMT repräsentiert.

`String toString()`

liefert eine Zeichenkette aus Wochentag, Monat, Tag, Stunde, Minute, Sekunde, lokale Zeitzone und Jahr. Beispiel: Thu Dec 29 15:44:55 CET 2022

`int compareTo(Date d)`

liefert 0, wenn this mit d übereinstimmt, einen Wert kleiner 0, wenn this vor d liegt und einen Wert größer 0, wenn this nach d liegt.

## SimpleDateFormat

Die Klasse `java.text.SimpleDateFormat` kann zur Formatierung von Datum und Zeit verwendet werden.

`SimpleDateFormat(String pattern)`

erzeugt ein Objekt mit dem durch `pattern` vorgegebenen Format.

Die Zeichenkette `pattern` kann sich aus den folgenden Zeichen zusammensetzen:

d Tag als Zahl, dd zweistellig

M Monat als Zahl, MM zweistellig, MMM abgekürzter Text, MMMM Langtext

yy Jahr (zweistellig), yyyy vierstellig

E Tag als abgekürzter Text,EEE Langtext

H Stunde (0 - 23), HH zweistellig

m Minute, mm zweistellig

s Sekunde, ss zweistellig

Beliebige in einfache Hochkommas eingeschlossene Zeichen können in pattern eingefügt werden. Sie werden nicht als Musterzeichen interpretiert. Zeichen, die nicht mit den Buchstaben A bis z und a bis z übereinstimmen, können direkt eingefügt werden.

```
String format(Date date)
    formatiert date gemäß der Vorgabe.
```

Beispiel:

```
Date now = new Date();
SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
String s = f.format(now);
```

s enthält die Zeichenkette (Beispiel): 02.06.2023 15:49:03

## TimeZone

Objekte der Klasse `TimeZone` repräsentieren Zeitzonen.

Methoden der Klasse `TimeZone`:

```
static String[] getAvailableIDs()
```

liefert ein Array aller Zeitzonennamen. So steht z. B. *GMT* für die *Greenwich Mean Time*, die der *Universal Time* (UT) entspricht, und *Europe/Berlin* für die in Deutschland geltende Zeitzone.

```
static TimeZone getTimeZone(String ID)
    liefert die Zeitzone für den Zeitzonennamen ID.
```

```
static TimeZone getDefault()
    liefert die lokale Zeitzone des Rechners.
```

```
StringgetID()
    liefert den Zeitzonennamen. Beispiel: Europe/Berlin.
```

```
StringgetDisplayName()
    liefert eine ausführliche Angabe zur Zeitzone.
    Beispiel: Mitteleuropäische Normalzeit
```

## GregorianCalendar

Die Klasse `GregorianCalendar` implementiert den gregorianischen Kalender, der in den meisten Ländern verwendet wird.

Konstruktoren:

```
GregorianCalendar()
GregorianCalendar(int year, int month, int day)
GregorianCalendar(int year, int month, int day, int hour, int minute)
GregorianCalendar(int year, int month, int day, int hour, int minute,
                  int second)
GregorianCalendar(TimeZone zone)
```

Die Konstruktoren verwenden entweder die aktuelle oder die angegebene Zeit und beziehen sich auf die lokale Zeitzone oder die angegebene Zeitzone.

*Achtung:*

Monate werden nicht von 1 bis 12, sondern von 0 bis 11 gezählt.

Die abstrakte Superklasse `java.util.Calendar` enthält eine Reihe von ganzzahligen konstanten Klassenvariablen, die als Feldbezeichner von den Methoden `set` und `get` benutzt werden (siehe Tabelle 15-4).

**Tabelle 15-4:** Calendar-Konstanten

Konstante	Bedeutung
ERA	Epoche
YEAR	Jahr
MONTH	Monat (0 ... 11)
WEEK_OF_MONTH	Woche innerhalb des Monats
WEEK_OF_YEAR	Kalenderwoche
DATE	Tag im Monat (1 ... 31)
DAY_OF_MONTH	Tag im Monat
DAY_OF_WEEK	Wochentag (1 = Sonntag)
DAY_OF_YEAR	Tag bezogen auf das Jahr
AM	Vormittag
PM	Nachmittag
AM_PM	AM bzw. PM
HOUR	Stunde (0 ... 12)
HOUR_OF_DAY	Stunde (0 ... 23)
MINUTE	Minute (0 ... 59)
SECOND	Sekunde (0 ... 59)
MILLISECOND	Millisekunde (0 ... 999)
ZONE_OFFSET	Zeitzonenabweichung in Millisekunden relativ zu GMT
DST_OFFSET	Sommerzeitabweichung in Millisekunden
JANUARY ... DECEMBER	Werte für Monat
MONDAY ... SUNDAY	Werte für Wochentag

Einige wichtige Methoden:

`int get(int field)`

liefert den Wert des Feldes, das durch die Konstante `field` bezeichnet wird.

Beispiel: `get(Calendar.YEAR)`

`void set(int field, int value)`

setzt den Wert des Feldes, das durch die Konstante `field` bezeichnet wird, auf `value`.

Beispiel: `set(Calendar.YEAR, 2010)`

Die drei folgenden Varianten der Methode `set` ändern gleich mehrere Felder:

`void set(int year, int month, int day)`

`void set(int year, int month, int day, int hour, int minute)`

`void set(int year, int month, int day, int hour, int minute, int second)`

`void add(int field, int amount)`

addiert den Wert `amount` zum Wert des Feldes, das durch die Konstante `field` bezeichnet wird.

`Date getTime()`

liefert den Zeitpunkt des `Calendar`-Objekts als `Date`-Objekt.

`void setTime(Date d)`

setzt den Zeitpunkt des `Calendar`-Objekts auf den durch `d` bestimmten Zeitpunkt.

`boolean equals(Object obj)`

liefert `true`, wenn die `Calendar`-Objekte `this` und `obj` gleich sind.

`boolean before(Object obj)`

liefert `true`, wenn dieses Objekt einen früheren Zeitpunkt darstellt als `obj`.

`boolean after(Object obj)`

liefert `true`, wenn dieses Objekt einen späteren Zeitpunkt darstellt als `obj`.

```
package date_time;
```

```
import java.text.SimpleDateFormat;
```

```
import java.util.Calendar;
```

```
import java.util.Date;
```

```
import java.util.GregorianCalendar;
```

```
import java.util.TimeZone;
```

```
public class DatumTest {
```

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

```
        SimpleDateFormat f1 = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
```

```
        SimpleDateFormat f2 = new SimpleDateFormat("dd.MM.yyyy");
```

```
        // aktuelles Rechnerdatum
```

```
        Date datum = new Date();
```

```
        System.out.println(f1.format(datum));
```

```
// Datum in der Zeitzone America/New_York
TimeZone tz = TimeZone.getTimeZone("America/New_York");
GregorianCalendar cal1 = new GregorianCalendar(tz);
cal1.setTime(datum);
System.out.println("Zeitzone: " + tz.getID());
System.out.println("Tag: " + cal1.get(Calendar.DATE));
System.out.println("Monat: " + (cal1.get(Calendar.MONTH) + 1));
System.out.println("Jahr: " + cal1.get(Calendar.YEAR));
System.out.println("Stunde: " + cal1.get(Calendar.HOUR_OF_DAY));
System.out.println("Minute: " + cal1.get(Calendar.MINUTE));
System.out.println("Sekunde: " + cal1.get(Calendar.SECOND));

// Datum auf den 1.1.2023 setzen
GregorianCalendar cal2 = new GregorianCalendar();
cal2.set(2023, 0, 1);
System.out.println(f2.format(cal2.getTime()));

// und 40 Tage dazu addieren
cal2.add(Calendar.DATE, 40);
System.out.println(f2.format(cal2.getTime()));
}
}
```

### Ausgabebeispiel:

```
02.06.2023 16:44:55
Zeitzone: America/New_York
Tag: 2
Monat: 6
Jahr: 2023
Stunde: 10
Minute: 44
Sekunde: 55
01.01.2023
10.02.2023
```

## Das Datum/Zeit-API aus java.time

Mit Java-Version 8 wurde ein neues, umfangreiches *Date and Time API* eingeführt, das es ermöglicht, einfach und komfortabel mit Datums- und Zeitwerten zu arbeiten.

Die Klasse `date_time.DateAndTime` gibt einen Einblick in die zahlreichen Möglichkeiten des neuen API.

### Datum/Uhrzeit festlegen

`LocalDate`, `LocalTime`, `LocalDateTime` aus dem Paket `java.time` stellen ein Datum, eine Uhrzeit und Datum/Uhrzeit jeweils ohne Angabe einer Zeitzone dar.

```
LocalDate heute = LocalDate.now();
System.out.println("Heute: " + heute);
```

```
LocalDate geburtstag = LocalDate.of(2006, 12, 28);
System.out.println("Geburtstag: " + geburtstag);
LocalTime tagesschau = LocalTime.of(20, 0);
System.out.println("Tagesschau: " + tagesschau);
LocalDateTime klausur = LocalDateTime.of(2023, 1, 20, 11, 30);
System.out.println("Klausur: " + klausur);
System.out.println();
```

**Ausgabe:**

```
Heute: 2023-06-02
Geburtstag: 2006-12-28
Tagesschau: 20:00
Klausur: 2023-01-20T11:30
```

**Datum/Uhrzeit parsen**

Die Klasse `java.time.format.DateTimeFormatter` erlaubt es, Zeichenketten in Datums- und Zeitwerte umzuwandeln bzw. Datums- und Zeitwerte als Zeichenketten zu formatieren.

```
DateTimeFormatter dtf1 = DateTimeFormatter.ofPattern("dd.MM.yyyy");
LocalDate d1 = LocalDate.parse("28.12.2006", dtf1);
System.out.println("Datum: " + d1);
DateTimeFormatter dtf2 = DateTimeFormatter.ofPattern("yyyyMMddHHmm");
LocalDateTime d2 = LocalDateTime.parse("202301201130", dtf2);
System.out.println("Datum und Uhrzeit: " + d2);
System.out.println();
```

**Ausgabe:**

```
Datum: 2006-12-28
Datum und Uhrzeit: 2023-01-20T11:30
```

**Datum/Uhrzeit formatiert ausgeben**

Hier werden Datums- und Zeitwerte formatiert ausgegeben. Dabei können Texte in der gewünschten Sprache ausgegeben werden.

```
LocalDateTime jetzt = LocalDateTime.now();
DateTimeFormatter dtf3 = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm:ss");
System.out.println(dtf3.format(jetzt));
DateTimeFormatter dtf4 = DateTimeFormatter.ofPattern("d. MMMM yyyy");
System.out.println(dtf4.format(jetzt));
DateTimeFormatter dtf5 = DateTimeFormatter.ofPattern("d. MMMM yyyy")
    .withLocale(Locale.of("en", "US"));
System.out.println(dtf5.format(jetzt));
System.out.println();
```

**Ausgabe:**

```
02.06.2023 16:47:25
2. Juni 2023
2. June 2023
```

## Rechnen mit Datums- und Zeitwerten

Mit den Methoden `plus` und `minus` können ausgehend von einem Zeitpunkt neue Zeitpunkte berechnet werden. `java.time.temporal.ChronoUnit`-Konstanten geben die Zeiteinheit an.

```
LocalDate in7Tagen = heute.plus(7, ChronoUnit.DAYS);
System.out.println("In 7 Tagen: " + in7Tagen);
LocalDate in3Monaten = heute.plus(3, ChronoUnit.MONTHS);
System.out.println("In 3 Monaten: " + in3Monaten);
LocalDate in1Jahr = heute.plus(1, ChronoUnit.YEARS);
System.out.println("In 1 Jahr: " + in1Jahr);
LocalDateTime in1Stunde = jetzt.plus(1, ChronoUnit.HOURS);
System.out.println("In 1 Stunde: " + in1Stunde);
LocalDateTime vor30Minuten = jetzt.minus(30, ChronoUnit.MINUTES);
System.out.println("Vor 30 Minuten: " + vor30Minuten);
System.out.println();
```

Ausgabe:

```
In 7 Tagen: 2023-06-09
In 3 Monaten: 2023-09-02
In 1 Jahr: 2024-06-02
In 1 Stunde: 2023-06-02T17:47:25.844869789
Vor 30 Minuten: 2023-06-02T16:17:25.844869789
```

## Zeitspannen ermitteln

Die Klasse `java.time.Duration` repräsentiert eine Zeitdauer. Die Klassenmethode `between` erzeugt ein `Duration`-Objekt. Dessen Methode `toMinutes` liefert die Dauer in Minuten. Die `Duration`-Methode `toString` gibt die Dauer in der Form `PTnHnMnS` aus. Im Beispiel `PT22H30M10S` (22 Stunden, 30 Minuten, 10 Sekunden).

```
LocalDateTime beginn = LocalDateTime.of(2016, 1, 4, 12, 0, 0);
LocalDateTime ende = LocalDateTime.of(2016, 1, 5, 10, 30, 10);
Duration dauer = Duration.between(beginn, ende);
System.out.println("Dauer: " + dauer);
System.out.println("Dauer: " + dauer.toMinutes());
LocalDateTime bestelltAm = LocalDateTime.of(2016, 1, 4, 0, 0);
LocalDateTime geliefertAm = LocalDateTime.of(2016, 2, 3, 0, 0);
Duration lieferZeit = Duration.between(bestelltAm, geliefertAm);
System.out.println("Lieferzeit: " + lieferZeit.toDays());
System.out.println();
```

Ausgabe:

```
Dauer: PT22H30M10S
Dauer: 1350
Lieferzeit: 30
```

Für längere Zeitspannen kann die `between`-Methode der `ChronoUnit`-Konstanten (beispielsweise `YEARS`) verwendet werden. Diese berechnet die Zeitspanne in der gewünschten Einheit.

## Einzelne Elemente extrahieren

Die einzelnen Bestandteile eines Datums- und Zeitwertes können einzeln extrahiert werden.

Die Klassen `java.time.Month` und `java.time.DayOfWeek` repräsentieren den Monat bzw. den Wochentag.

Mit `getDisplayName` können ihre Namen sprachabhängig (hier wird das *Default-Locale* benutzt) ausgegeben werden.

Die `Month`-Methode `length` liefert die Anzahl Tage dieses Monats, wobei der `boolean`-Parameter angibt, ob ein Schaltjahr vorliegt oder nicht.

Die `LocalDate`-Methode `isLeapYear` prüft, ob das zugrunde liegende Jahr ein Schaltjahr ist.

```
LocalDateTime d3 = LocalDateTime.of(2016, 2, 4, 12, 30, 15);
int jahr = d3.getYear();
System.out.println("Jahr: " + jahr);
Month monat = d3.getMonth();
System.out.println("Monat: " + monat.getDisplayName(TextStyle.FULL,
    Locale.getDefault()) + " " + monat.getValue());
int anz = monat.length(d3.toLocalDate().isLeapYear());
System.out.println("Anzahl Tage: " + anz);
int tag = d3.getDayOfMonth();
System.out.println("Tag: " + tag);
DayOfWeek wochentag = d3.getDayOfWeek();
System.out.println("Wochentag: " + wochentag.getDisplayName(TextStyle.FULL,
    Locale.getDefault()));
int stunde = d3.getHour();
System.out.println("Stunde: " + stunde);
int minute = d3.getMinute();
System.out.println("Minute: " + minute);
int sekunde = d3.getSecond();
System.out.println("Sekunde: " + sekunde);
System.out.println();
```

### Ausgabe:

```
Jahr: 2016
Monat: Februar 2
Anzahl Tage: 29
Tag: 4
Wochentag: Donnerstag
Stunde: 12
Minute: 30
Sekunde: 15
```

## Zeitzonen

Die Klasse `java.time.ZonedDateTime` repräsentiert eine Datums- und Zeitangabe mit Zeitzone.

`java.time.ZoneId` stellt eine Zeitzone dar.

Mit der `LocalDateTime`-Methode `atZone` kann aus einem `LocalDateTime`-Objekt ein `ZonedDateTime`-Objekt erzeugt werden.

Die Methode `withZoneSameInstant` der Klasse `ZonedDateTime` erzeugt ein neues `ZonedDateTime`-Objekt mit der vorgegebenen Zeitzone.

```
ZonedDateTime now = ZonedDateTime.now();
System.out.println("Jetzt: " + now);
ZoneId germanTimeZone = ZoneId.of("Europe/Berlin");
ZonedDateTime nowInGermany = ZonedDateTime.now(germanTimeZone);
System.out.println("Jetzt in Deutschland: " + nowInGermany);
ZoneId newYorkTimeZone = ZoneId.of("America/New_York");
ZonedDateTime nowInNewYork = ZonedDateTime.now(newYorkTimeZone);
System.out.println("Jetzt in New York: " + nowInNewYork);
LocalDateTime termin = LocalDateTime.of(2016, 2, 4, 12, 30, 15);
ZonedDateTime terminInGermany = termin.atZone(germanTimeZone);
System.out.println("Termin in Deutschland: " + terminInGermany);
ZonedDateTime terminInNewYork = terminInGermany
    .withZoneSameInstant(newYorkTimeZone);
System.out.println("Termin in New York: " + terminInNewYork);
System.out.println();
```

**Ausgabe:**

```
Jetzt: 2023-06-02T16:47:25.906876689+02:00[Europe/Berlin]
Jetzt in Deutschland: 2023-06-02T16:47:25.907019326+02:00[Europe/Berlin]
Jetzt in New York: 2023-06-02T10:47:25.908105744-04:00[America/New_York]
Termin in Deutschland: 2016-02-04T12:30:15+01:00[Europe/Berlin]
Termin in New York: 2016-02-04T06:30:15-05:00[America/New_York]
```

## Laufzeit messen

`java.time.Instant`-Objekte speichern die Anzahl Nanosekunden ab dem 1.1.1970. Die Dauer zwischen zwei Zeitpunkten kann mit Hilfe der `Duration`-Methode `between` ermittelt werden.

```
Instant instant1 = Instant.now();
double x = 0;
for (int i = 0; i < 1_000_000; i++) {
    x += Math.sin(i) + Math.cos(i);
}
Instant instant2 = Instant.now();
System.out.println("Wert: " + x);
System.out.println("Laufzeit: " + Duration.between(instant1, instant2).toMillis());
```

**Ausgabe:**

```
Wert: -0.05582148989519953
Laufzeit: 98
```

## Die `Date`-Methode

`Instant toInstant()`

konvertiert ein `Date`-Objekt in ein `Instant`-Objekt.

Die `LocalDate`-Methode

```
static LocalDate ofInstant(Instant instant, ZoneId zone)
```

erzeugt ein `LocalDate`-Objekt aus einem `Instant`-Objekt und einer Zeitzone.

Ähnliches gilt für `LocalDateTime`.

Mit Hilfe dieser Methoden kann nun ein `Date`-Objekt in ein `LocalDate`- bzw. `LocalDateTime`-Objekt umgewandelt werden.

### Umwandlung von `Date` zu `LocalDate` bzw. `LocalDateTime`

```
package date_time;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Date;

public class Converter {
    public static LocalDate convertToLocalDate(Date date) {
        return LocalDate.ofInstant(date.toInstant(), ZoneId.systemDefault());
    }

    public static LocalDateTime convertToLocalDateTime(Date date) {
        return LocalDateTime.ofInstant(date.toInstant(), ZoneId.systemDefault());
    }

    public static void main(String[] args) {
        Date date = new Date();
        System.out.println(convertToLocalDate(date));
        System.out.println(convertToLocalDateTime(date));
    }
}
```

## 15.9 Aufgaben

### Aufgabe 1

Schreiben Sie ein Programm, das die Grundrechenarten Addition, Subtraktion, Multiplikation und Division für Gleitkommazahlen ausführen kann. Dabei soll das Programm jeweils mit drei Parametern (z. B.  $114.5 + 2.5$ ) aufgerufen werden.

*Lösung: Paket rechner*

### Aufgabe 2

Schreiben Sie ein Programm, das die Qualität eines Passworts mit Punkten bewertet. Für jede der folgenden erfüllten Regeln soll ein Punkt vergeben werden:

- Länge mindestens 8 Zeichen,
- enthält Klein- und Großbuchstaben,
- enthält Ziffern,
- enthält Sonderzeichen.

Das Passwort soll als Kommandozeilen-Parameter eingelesen werden.

*Lösung: Paket password*

### Aufgabe 3

Schreiben Sie ein Programm, das für eine vorgegebene Zeichenkette die Anzahl der in ihr vorkommenden Kleinbuchstaben, Großbuchstaben, Leerzeichen und sonstigen Zeichen ermittelt.

*Lösung: Paket analyse*

### Aufgabe 4

Die ganzen Zahlen von 1 bis `Integer.MAX_VALUE` sollen in einer Schleife aufaddiert werden. Die Variable `sum` soll (versehentlich) den Typ `Long` und nicht den elementaren Typ `long` haben.

Messen Sie die Laufzeit in beiden Fällen und vergleichen Sie.

*Lösung: Paket laufzeit*

### Aufgabe 5

Mit `clone()` können auch Arrays kopiert werden. Hier handelt es sich um flache Kopien. Demonstrieren Sie diesen Sachverhalt.

*Lösung: Paket clone*

### Aufgabe 6

Definieren Sie die Klasse `IntegerStack`, die ganze Zahlen in einem Array einer vorgegebenen Länge speichern kann und die folgenden Methoden enthält:

`public void push(int value)`

legt den Wert `value` oben auf den Stapel.

`public int pop()`

liefert das oberste Element des Stapels und entfernt es aus dem Stapel.

`IntegerStack` soll das Interface `Cloneable` und die Methode `clone` so implementieren, dass "tiefe Kopien" möglich sind.

*Lösung: Paket deep\_copy*

**Aufgabe 7**

Implementieren Sie eine Klasse `MailAdressen`, die E-Mail-Adressen zu Namen speichern kann und den Zugriff über Namen ermöglicht. Die Einträge sollen aus einer Datei geladen werden können. Benutzen Sie die Klasse `Properties`.

*Lösung: Paket adressen*

**Aufgabe 8**

Die Klasse `Artikel` soll die Instanzvariablen `id`, `preis` und `menge` enthalten. Hierzu sind ein Konstruktor und die jeweiligen get/set-Methoden zu definieren.

Die Klasse `Warenkorb` soll mit Hilfe eines `vector`-Objekts `Artikel`-Objekte aufnehmen können.

Implementieren Sie die folgenden Methoden:

```
public void add(Artikel artikel)  
fügt ein Artikel-Objekt in den Warenkorb ein.
```

```
public double bestellwert()  
liefert den Bestellwert (preis * menge) aller Artikel im Warenkorb.
```

*Lösung: Paket warenkorb*

**Aufgabe 9**

Arrays sind eine Sonderform von Klassen. Ermitteln Sie den Klassennamen von `int`-, `double`- und `String`-Arrays.

*Lösung: Paket arrays*

**Aufgabe 10**

Implementieren Sie die Methode

```
public static String[] extract(String text, String delim),  
die einen Text in die einzelnen Wörter zerlegt und diese in sortierter Reihenfolge in  
einem Array zurück gibt.
```

Tipp: Nutzen Sie ein `vector`-Objekt als Zwischenspeicher.

*Lösung: Paket woerter1*

**Aufgabe 11**

Entwickeln Sie eine Variante der Lösung zu Aufgabe 10. Das Array soll jedes Wort genau einmal, ergänzt um die Anzahl des Auftretens dieses Wertes im Text, enthalten. Tipp: Nutzen Sie ein `Hashtable`-Objekt als Zwischenspeicher.

*Lösung: Paket woerter2*

**Aufgabe 12**

Erzeugen Sie ein Array aus Objekten der Klasse `Kunde` (siehe Kapitel 15.2) und sortieren Sie diese aufsteigend nach dem Kundennamen mit Hilfe der Methode `Arrays.sort`. Hierzu ist für die Klasse `Kunde` das Interface `Comparable` in geeigneter Form zu implementieren.

*Lösung: Paket kunde*

**Aufgabe 13**

Nutzen Sie die Klasse `BigInteger`, um beliebig große Fakultäten  $n! = 1 * 2 * \dots * n$  zu berechnen.

*Lösung: Paket fakultaet*

**Aufgabe 14**

Implementieren Sie die folgenden Methoden zur Rundung von `float`- bzw. `double`-Zahlen:

```
public static float round(float x, int digits)  
public static double round(double x, int digits)
```

Beispiel:

`round(123.456, 2)` liefert `123.46`

Tipp: Nutzen Sie die `Math`-Methoden `pow` und `round`.

*Lösung: Paket runden*

**Aufgabe 15**

Entwickeln Sie ein Programm, das Ziehungen von Lottozahlen (6 aus 49) simuliert und zeigt, wie viele Ziehungen nötig sind, damit sechs richtige Zahlen eines zu Beginn vorgegebenen Tipps erzielt werden. Um eine Ziehung zu simulieren, werden sechs Zufallszahlen in einem Array gespeichert und sortiert.

*Lösung: Paket lotto*

**Aufgabe 16**

Die Methode

```
static String random(int n)
```

soll eine Zeichenkette mit `n` zufällig ausgewählten Kleinbuchstaben (a - z) zurückgeben.

*Lösung: Paket zufall1*

**Aufgabe 17**

Die Methode

```
static String random(String[] array)
```

soll ein zufällig ausgewähltes Element liefern.

*Lösung: Paket zufall2*

**Aufgabe 18**

Die Methode

```
static void shuffleArray(int[] a)
```

soll die Elemente eines Arrays zufällig neu anordnen (mischen).

*Lösung: Paket shuffle*

**Aufgabe 19**

Ein Text soll in einem neu erzeugten Text so versteckt werden, dass er nur von einer "eingeweihten" Person gefunden werden kann. Der geheime Text wird zunächst in Großbuchstaben umgewandelt. Hinter jedem Zeichen werden n zufällige Großbuchstaben gesetzt.

Beispiel (mit n = 3):

Geheimer Text: *Wir treffen uns um acht hinter der Hecke*

Erzeugter Text: WXBJIDVXRUAANMMTEUERVDLEMVMFAWDFJNSESSKNHRN QTWUCNMNDPYSVUP  
GSKUPGGMLJZ LBNANUECVKOHJZJTMOD DCCHWYCIVMFNIKZTHXEJECKDW  
BYTDLBYEEJVRKRF PFVHSJGEEEUCMCIKRNSEUVK

*Lösung: Paket secret*

**Aufgabe 20**

Implementieren Sie eine Klasse `AktuellesDatum` mit einer Klassenmethode, die das aktuelle Datum wie in folgendem Beispiel als Zeichenkette liefert:

Dienstag, den 17.10.2023

*Lösung: Paket datum*

**Aufgabe 21**

Berechnen Sie die Zeit in Tagen, Stunden, Minuten und Sekunden, die seit dem 1.1.2020 um 00:00:00 Uhr vergangen ist.

Tipp: Verwenden Sie hierzu die Duration-Methoden `toDaysPart()`, `toHoursPart()`, `toMinutesPart()` und `toSecondsPart()`.

*Lösung: Paket duration*

**Aufgabe 22**

Realisieren Sie eine Methode, die die Differenz zwischen zwei Datumsangaben in Tagen ermittelt:

```
static int getDaysBetween(int startYear, int startMonth, int startDay,  
                         int endYear, int endMonth, int endDay)
```

Verwenden Sie `GregorianCalendar`-Objekte für das Start- und das Enddatum. Nutzen Sie die `GregorianCalendar`-Methode `add`, um solange jeweils einen Tag auf das Startdatum zu addieren, bis das Enddatum überschritten ist.

*Lösung: Paket diff*

**Aufgabe 23**

Isabelle ist am 28.12.2006 geboren. Wie alt ist sie heute? Wie viele Tage dauert es noch bis zum nächsten Geburtstag?

*Lösung: Paket geburtstag*



# 16 Internationalisierung

Häufig soll ein und dieselbe Softwareversion in mehreren Ländern und unterschiedlichen Sprachregionen eingesetzt werden. Dabei ist es wichtig, dass die Programme leicht und möglichst ohne den Quellcode ändern zu müssen an die regionalen Gegebenheiten angepasst werden können.

Java bietet eine sprach- und länderspezifische Datumsformatierung, Dezimaldarstellung und Einstellung von Währungssymbolen und Textelementen. Diese Programmgestaltung nennt man *Internationalisierung* (engl. *internationalization*), abgekürzt mit *I18N*.<sup>1</sup>

## Lernziele

In diesem Kapitel lernen Sie

- wie Datum und Zahlen sprach- und länderspezifisch dargestellt und
- wie sprachabhängige Texte verwaltet werden können.

## 16.1 Die Klasse Locale

### Sprach- und Ländercode

Die Bezeichnung von Sprachen, Ländern und Regionen ist standardisiert (ISO-639 und ISO-3166). Der *Sprachcode* besteht aus zwei kleingeschriebenen Buchstaben, der *Ländercode* aus zwei großgeschriebenen Buchstaben.

Beispiele:

- de AT Deutsch (Österreich)
- de CH Deutsch (Schweiz)
- de DE Deutsch (Deutschland)
- en AU Englisch (Australien)
- en CA Englisch (Kanada)
- en GB Englisch (Vereinigtes Königreich)
- en IE Englisch (Irland)
- en IN Englisch (Indien)
- en MT Englisch (Malta)
- en NZ Englisch (Neuseeland)
- en PH Englisch (Philippinen)
- en SG Englisch (Singapur)
- en US Englisch (Vereinigte Staaten von Amerika)
- en ZA Englisch (Südafrika)

---

<sup>1</sup> Im englischen Wort befinden sich 18 Buchstaben zwischen *I* und *N*.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_16](https://doi.org/10.1007/978-3-658-43574-5_16).

## Locale

Die Klasse `java.util.Locale` verwaltet Sprach- und Länderangaben und repräsentiert damit eine geografische, politische oder kulturelle Region (`Locale` = Gebietsschema).

Die Konstruktoren von `Locale` sind ab Java 19 veraltet (*deprecated*). Stattdessen bieten die folgenden statischen Methoden entsprechende `Locale`-Instanzen.

```
static Locale of(String language)
static Locale of(String language, String country)
```

Sprach- und Ländereinstellungen werden standardmäßig von den Systemeigenschaften `user.language` und `user.country` übernommen.

Weitere Methoden:

```
static Locale getDefault()
liefert die aktuelle Standardeinstellung.
```

```
static void setDefault(Locale newLocale)
setzt das Default-Locale für die aktuelle Anwendung.
```

```
String getLanguage()
String getCountry()
liefert den Sprach- bzw. Ländercode.
```

```
String getDisplayLanguage()
String getDisplayCountry()
String getDisplayName()
liefert die Kurzbeschreibung von Sprache, Land bzw. beidem.
```

```
static Locale[] getAvailableLocales()
liefert alle verfügbaren Locales.
```

`Locale` enthält auch eine Reihe von Konstanten für Sprachen und Länder, z. B. `Locale.GERMAN`, `Locale.GERMANY`.

```
package locale;

import java.util.Locale;

public class LocaleTest1 {
    public static void main(String[] args) {
        System.out.println("user.language: " + System.getProperty("user.language"));
        System.out.println("user.country: " + System.getProperty("user.country"));
        System.out.println();

        Locale locale = Locale.getDefault();
        print(locale);

        locale = Locale.of("en");
        print(locale);
```

```
locale = Locale.of("en", "US");
print(locale);

print(Locale.GERMAN);
print(Locale.GERMANY);
}

public static void print(Locale locale) {
    System.out.println("Locale: " + locale);
    System.out.println("Language: " + locale.getLanguage());
    System.out.println("Country: " + locale.getCountry());
    System.out.println("DisplayLanguage: " + locale.getDisplayLanguage());
    System.out.println("DisplayCountry: " + locale.getDisplayCountry());
    System.out.println("DisplayName: " + locale.getDisplayName());
    System.out.println();
}
}
```

### Ausgabe des Programms:

user.language: de

user.country: DE

Locale: de\_DE

Language: de

Country: DE

DisplayLanguage: Deutsch

DisplayCountry: Deutschland

DisplayName: Deutsch (Deutschland)

Locale: en

Language: en

Country:

DisplayLanguage: Englisch

DisplayCountry:

DisplayName: Englisch

Locale: en\_US

Language: en

Country: US

DisplayLanguage: Englisch

DisplayCountry: Vereinigte Staaten

DisplayName: Englisch (Vereinigte Staaten)

Locale: de

Language: de

Country:

DisplayLanguage: Deutsch

DisplayCountry:

DisplayName: Deutsch

Locale: de\_DE

Language: de

Country: DE

DisplayLanguage: Deutsch

DisplayCountry: Deutschland

DisplayName: Deutsch (Deutschland)

Durch Aufruf von z. B.

```
java -Duser.language=en -Duser.country=US ...
```

kann die Standardeinstellung geändert werden.

Das nächste Programm zeigt alle verfügbaren Locales.

```
package locale;

import java.util.Locale;

public class LocaleTest2 {
    public static void main(String[] args) {
        Locale[] locales = Locale.getAvailableLocales();
        for (Locale locale : locales) {
            System.out.println(locale.getLanguage() + " " + locale.getCountry()
                + " " + locale.getDisplayName());
        }
    }
}
```

## 16.2 Zeit und Zahlen darstellen

Mit Hilfe der Klasse `java.text.DateFormat` können Datum und Uhrzeit sprachabhängig formatiert werden.

Die statischen `DateFormat`-Methoden

`getDateInstance`, `getTimeInstance` und `getDateTimeInstance` liefern `DateFormat`-Instanzen für Datum, Uhrzeit bzw. beides.

Hierbei kann der Formatierungsstil in Form von `DateFormat`-Konstanten als Argument mitgegeben werden (siehe folgendes Programm).

Die `DateFormat`-Methode

```
String format(Date date)
formatiert Datum/Uhrzeit.
```

Folgendes Programm zeigt, wie sich die Formatierung von Dezimalzahlen und Datum/Uhrzeit an das jeweilige Gebietsschema anpasst (hier: de\_DE).

`DecimalFormat` wurde bereits in Kapitel 15 beschrieben.

```
package display;

import java.text.DateFormat;
import java.text.DecimalFormat;
import java.util.Date;

public class FormatTest {
```

```
public static void main(String[] args) {
    DecimalFormat f = new DecimalFormat("###,##0.00");
    System.out.println(f.format(24522.4567));

    DateFormat[] formats = {
        DateFormat.getDateInstance(),
        DateFormat.getDateInstance(DateFormat.SHORT),
        DateFormat.getDateInstance(DateFormat.MEDIUM),
        DateFormat.getDateInstance(DateFormat.LONG),
        DateFormat.getDateInstance(DateFormat.FULL),

        DateFormat.getTimeInstance(),
        DateFormat.getTimeInstance(DateFormat.SHORT),
        DateFormat.getTimeInstance(DateFormat.MEDIUM),
        DateFormat.getTimeInstance(DateFormat.LONG),
        DateFormat.getTimeInstance(DateFormat.FULL),

        DateFormat.getDateTimeInstance(),
        DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.SHORT)
    };

    Date now = new Date();
    for (DateFormat df : formats) {
        System.out.println(df.format(now));
    }
}
```

Ausgabe des Programms (Beispiel):

```
24.522,46
04.06.2023
04.06.23
04.06.2023
4. Juni 2023
Sonntag, 4. Juni 2023
17:18:32
17:18
17:18:32
17:18:32 MESZ
17:18:32 Mitteleuropäische Sommerzeit
04.06.2023, 17:18:32
04.06.2023, 17:18
```

Das nächste Programm zeigt entsprechende Ausgaben von Datum und Uhrzeit für das mit Java 8 eingeführte neue *Date and Time API*. Die Stil-Varianten `LONG` and `FULL` für Zeitangaben stehen nur für Objekte mit Zeitzone zur Verfügung.

```
package display;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;
```

```

public class DateTimeFormatterTest {
    public static void main(String[] args) {
        DateTimeFormatter[] formatters = {
            DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT),
            DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM),
            DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG),
            DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL),

            DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT),
            DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM),

            DateTimeFormatter.ofLocalizedName(FormatStyle.MEDIUM,
                FormatStyle.SHORT),
        };

        LocalDateTime now = LocalDateTime.now();
        for (DateTimeFormatter dtf : formatters) {
            System.out.println(dtf.format(now));
        }

        ZonedDateTime zonedDateTime = now.atZone(ZoneId.systemDefault());
        DateTimeFormatter dtf1 = DateTimeFormatter.ofLocalizedTime(FormatStyle.LONG);
        System.out.println(dtf1.format(zonedDateTime));
        DateTimeFormatter dtf2 = DateTimeFormatter.ofLocalizedName(FormatStyle.FULL);
        System.out.println(dtf2.format(zonedDateTime));
    }
}

```

`java.text.NumberFormat` ermöglicht die Verarbeitung von Fließkommazahlen in länder spezifischer Schreibweise.

`static NumberFormat getInstance(Locale locale)`  
liefert ein `NumberFormat`-Objekt für das angegebene `Locale`-Objekt.  
`getInstance()` ohne Argument liefert ein solches für das aktuell eingestellte Default-`Locale`.

`Number parse(String source) throws java.text.ParseException`  
erzeugt ein Objekt vom Typ `java.lang.Number`. Dieses kann mit `doubleValue()` in eine double-Zahl gewandelt werden.

```

package display;

import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class ParseTest {
    public static void main(String[] args) throws ParseException {
        Locale locale = Locale.of("en", "US");
        NumberFormat nf = NumberFormat.getInstance(locale);
        Number number = nf.parse("12,345.67");
        System.out.println(number.doubleValue());
    }
}

```

```
locale = Locale.of("de", "DE");
nf = NumberFormat.getInstance(locale);
number = nf.parse("12.345,67");
System.out.println(number.doubleValue());

nf = NumberFormat.getInstance();
number = nf.parse("12.345,67");
System.out.println(number.doubleValue());
}
}
```

Ausgabe des Programms:

```
12345.67
12345.67
12345.67
```

## 16.3 Sprachspezifische Sortierung

Die Klasse `java.text.Collator` unterstützt zusammen mit `Arrays.sort(...)` die sprachspezifische Sortierung von Zeichenketten.

`Collator.getInstance()` liefert ein `Collator`-Objekt für das aktuell eingestellte Default-Locale. Optional kann an `getInstance` auch ein `Locale`-Objekt übergeben werden.

```
package sort;

import java.text.Collator;
import java.util.Arrays;

public class Sortieren {
    public static void main(String[] args) {
        String[] words1 = { "Auto", "Ärger", "Anton" };
        Arrays.sort(words1);
        System.out.println(Arrays.toString(words1));

        String[] words2 = { "Auto", "Ärger", "Anton" };
        Arrays.sort(words2, Collator.getInstance());
        System.out.println(Arrays.toString(words2));
    }
}
```

Ausgabe des Programms:

```
[Anton, Auto, Ärger]
[Anton, Ärger, Auto]
```

## 16.4 Ressourcenbündel

Die Klasse `java.util.ResourceBundle` repräsentiert ein sogenanntes *Ressourcenbündel*. Ein Ressourcenbündel fasst Dateien zusammen, die Schlüssel und Texte in Übersetzung enthalten.

Die Klasse `ResourceBundle` bietet eine Methode, um über den eindeutigen Schlüssel auf den zugehörigen Text abhängig vom eingestellten Gebietsschema zugreifen zu können:

```
String getString(String key)
```

Im Quellcode werden anstelle der sprachabhängigen Texte nur die Schlüssel verwendet. Ändert sich die `Locale`-Einstellung, so gibt das Programm automatisch die Texte in der eingestellten Sprache aus.

### Namenskonventionen für die Textdateien eines Bündels

```
Buendelname.properties (Default)  
Buendelname_Sprachcode.properties  
Buendelname_Sprachcode_Laendercode.properties
```

Diese Dateien werden von der `ResourceBundle`-Methode `getBundle im aktuellen Klassenpfad` gesucht:

```
static ResourceBundle getBundle(String baseName)  
static ResourceBundle getBundle(String baseName, Locale locale)
```

Die erste Methode nutzt das *Default-Locale*. Die zweite Methode nutzt das Gebietsschema `locale` zur Festlegung von Sprache und Land.

In unserem Beispiel befindet sich das Bündel im Verzeichnis (Paket) `resources` und hat den Namen `Bundle`.

`baseName` enthält den voll qualifizierten Namen: `resources.Bundle`

Inhalt von `Bundle.properties`:

```
greeting=Welcome!  
firstname=First name  
lastname=Last name  
email=Email address
```

Inhalt von `Bundle_de.properties`:

```
greeting=Willkommen!  
firstname=Vorname  
lastname=Nachname  
email=E-Mail
```

```
package resource_bundle;

import java.util.ResourceBundle;

public class BundleTest {
    public static void main(String[] args) {
        ResourceBundle res = ResourceBundle.getBundle("resources.Bundle");

        String first = "Hugo";
        String last = "Meier";
        String mail = "hugo.meier@web.de";

        System.out.println(res.getString("greeting"));
        System.out.println(res.getString("firstname") + ": " + first);
        System.out.println(res.getString("lastname") + ": " + last);
        System.out.println(res.getString("email") + ": " + mail);
    }
}
```

Ausgabe des Programms (Standardeinstellung ist hier *de*):

```
Willkommen!
Vorname: Hugo
Nachname: Meier
E-Mail: hugo.meier@web.de
```

Der Aufruf von

```
java -Duser.language=en -cp out/production/kap16 resource_bundle.BundleTest
```

führt zur Ausgabe:

```
Welcome!
First name: Hugo
Last name: Meier
Email address: hugo.meier@web.de
```

## 16.5 Aufgaben

### Aufgabe 1

Schreiben Sie ein Programm, das einen Begrüßungstext für drei verschiedene Gebietsschemata ausgibt: *de\_DE*, *en\_US* und *fr\_FR*. Kommandozeilen-Parameter sind Sprachcode und Ländercode.

*Lösung: Paket welcome*

### Aufgabe 2

Schreiben Sie ein Programm, das für die Gebietsschemata *de\_DE*, *en\_US* und *fr\_FR* das aktuelle Datum in Langform hintereinander ausgibt.

*Lösung: Paket date*



# 17 Services

Ein Ziel bei der Entwicklung größerer Systeme ist, die Abhängigkeiten zwischen den Typen (Klassen, Interfaces) auf ein Mindestmaß zu reduzieren und sich nicht an konkrete Implementierungen zu binden.

## Lernziele

In diesem Kapitel lernen Sie

- wie Service Provider genutzt werden können, um die Abhängigkeit von konkreten Implementierungen zu reduzieren,
- wie Provider konfiguriert werden und
- wie Services genutzt werden können.

## 17.1 Service Provider

### Programmieren gegen Schnittstellen

Implementiert die Klasse A das Interface X und nutzt man nur die in X definierten Methoden, so sollte man mit

```
X ref = new A();
```

eine Instanz der Klasse A erzeugen und folglich später alle Methoden über diese Referenz aufrufen.

Das erleichtert den Austausch gegen eine alternative Implementierung B, da nur die obere Anweisung auszutauschen ist:

```
X ref = new B();
```

Etwas abstrakter formuliert:

Eine Schnittstelle beschreibt einen Service, der von einem *Service Provider* angeboten wird. Dabei kann es mehrere Provider (unterschiedliche Implementierungen) geben.

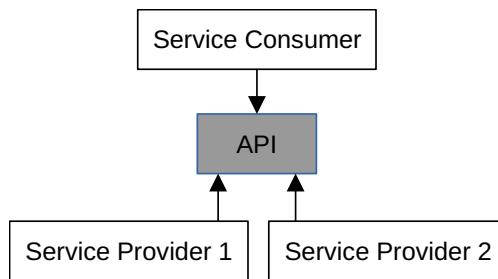
Dieses Kapitel beschreibt einen Mechanismus, wie Implementierungen (wie oben A bzw. B) vollständig aus dem Quellcode des *Service Consumer* herausgehalten werden können.

Einige Voraussetzung:

Die implementierenden Klassen müssen `public` sein und den parameterlosen Konstruktor enthalten.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_17](https://doi.org/10.1007/978-3-658-43574-5_17).



**Abbildung 17-1:** Service Consumer und Service Provider

### ServiceLoader

Die statische Methode `load` der Klasse `java.util.ServiceLoader` liefert einen `ServiceLoader` für die angegebene Service-Schnittstelle.

Im Beispiel (`MyService` ist ein Interface):

```
ServiceLoader serviceLoader = ServiceLoader.load(MyService.class);
```

Alle verfügbaren Service-Implementierungen kann man nun mit einer `foreach`-Schleife erhalten. Wir interessieren uns nur für die erste gefundene Implementierung.

### Provider-Konfiguration

Damit eine Service-Implementierung gefunden werden kann, muss eine Zuordnung von Service-Schnittstelle zu Service-Implementierung erfolgen.

Hierzu wird im Wurzelverzeichnis des Klassenpfades ein Ordner `META-INF` mit einem Unterordner `services` angelegt. `services` enthält eine Textdatei mit dem voll qualifizierten Namen der Service-Schnittstelle. Die Textdatei enthält in einer Zeile den voll qualifizierten Namen der Klasse, die den Service implementiert. Diese Datei muss UTF-8-codiert sein.

Im Beispiel:

```
META-INF/
  services/
    de.test.api.MyService
```

Inhalt der Textdatei:

```
de.test.impl.MyServiceImpl
```

## Service Provider

Das Interface MyService:

```
package de.test.api;

import java.util.ServiceLoader;

public interface MyService {
    void show();

    static MyService newInstance() {
        ServiceLoader<MyService> serviceLoader = ServiceLoader.load(MyService.class);
        for (Object service : serviceLoader) {
            return (MyService) service;
        }
        return null;
    }
}
```

Zur Demonstration nutzen wird nur eine einfache Service-Methode:

```
void show()
```

Den oben beschriebenen *Lookup*-Mechanismus nehmen wir gleich in das Interface mit auf, denn Interfaces können ab Java-Version 8 statische Methoden enthalten.

Wenn mehrere Implementierungen in `META-INF/services/de.test.api.MyService` eingetragen sind, wird die zuerst gefundene zurückgeliefert.

Mit Hilfe von *Generics* (siehe Kapitel 19) kann diese Methode kompakter formuliert werden:

```
static MyService newInstance() {
    ServiceLoader<MyService> serviceLoader = ServiceLoader.load(MyService.class);
    for (MyService service : serviceLoader) {
        return service;
    }
    return null;
}
```

Die Service-Implementierung:

```
package de.test.impl;

import de.test.api.MyService;

public class MyServiceImpl implements MyService {
    @Override
    public void show() {
        System.out.println("show aus MyServiceImpl");
    }
}
```

Implementierung und Konfiguration können in einer jar-Datei zusammengefasst werden:<sup>1</sup>

```
jar --create --file myservice.jar \
-C out/production/kap17 de -C out/production/kap17 META-INF
```

Diese Datei muss dann zur Compilierung und Ausführung des *Consumers* in den Klassenpfad aufgenommen werden.

## 17.2 Service Consumer

```
import de.test.api.MyService;

public class Consumer {
    public static void main(String[] args) {
        MyService myService = MyService.newInstance();
        if (myService != null)
            myService.show();
    }
}
```

Ausführung des Consumers:

```
java -cp out/production/kap17:myservice.jar Consumer
```

### Fazit

- Service Consumer können Service Provider nutzen, ohne dass eine Abhängigkeit zwischen den jeweiligen Klassen im Quellcode besteht.
- Service Provider, die zum Zeitpunkt der Compilierung des Consumers noch nicht vorhanden waren, können zur Laufzeit (in Form von jar-Dateien) hinzugefügt werden.

## 17.3 Aufgaben

### Aufgabe 1

Das Interface `Service` deklariert die Methode `void work()`. Erstellen Sie eine Klasse `ServiceFactory` mit der folgenden Methode:

```
public static Service createService(String className)
```

---

1 In IntelliJ IDEA wird der Byte-Code standardmäßig im Verzeichnis `out/production/Projektname` gespeichert.

Diese Methode kann für eine Klasse `className` aufgerufen werden, die das Interface `Service` implementiert. `createService` erzeugt eine Instanz dieser Klasse. Nutzen Sie hierzu `Class`-Methoden.

Erstellen Sie zwei Implementierungen für das Interface `Service` und testen Sie `createService`. Dieses Erzeugungsmuster *Fabrik-Methode* zeigt, dass Implementierungen einer bestimmten Schnittstelle ausgetauscht werden können, ohne dass das Programm geändert werden muss.

*Lösung: Paket factory*

### Aufgabe 2

Entwickeln Sie mit Hilfe des `ServiceLoader`-Konzepts einen Service, der einen Text in Anführungszeichen einkleidet.

Das Interface `Quoter` soll die Methode `String quote(String text)` enthalten. Die Service-Implementierung `GermanQuoter` soll tief- und hochgestellte Anführungszeichen nutzen, Unicodezeichen '\u201E' und '\u201D'.

*Lösung: Paket quoter.demo*

### Aufgabe 3

Zeigen Sie anhand eines einfachen Beispiels, dass ein Service Consumer *in einem Programmdurchlauf* auch mehrere Implementierungen desselben Service nutzen kann.

*Lösung: Paket service.demo*



# 18 Javadoc

Mit dem JDK-Tool `javadoc` kann aus Java-Quelltexten, die mit Dokumentationskommentaren versehen sind, automatisch Dokumentation im HTML-Format generiert werden.

## Lernziele

In diesem Kapitel lernen Sie

- wie das Werkzeug `javadoc` genutzt werden kann, um Dokumentation zu generieren.

## 18.1 javadoc-Syntax

Innerhalb von Kommentaren der Form `/** ... */` können `javadoc`-Tags zur Festlegung spezieller Informationen verwendet werden. Tabelle 18-1 enthält die wichtigsten Tags.

**Tabelle 18-1:** Wichtige `javadoc`-Tags

<code>{@summary text}</code>	Zusammenfassende Beschreibung
<code>{@code text}</code>	Einzeiliger Code-Abschnitt
<code>@param parameter description</code>	Parameterbeschreibung
<code>@return description</code>	Beschreibung des Rückgabewertes
<code>@throws exception description</code>	Beschreibung einer Ausnahme
<code>@see package.class#member label</code>	Link mit der Bezeichnung <code>label</code>

Der Quellcode der Klassen des Pakets `primzahlen` des folgenden Programms wird mit geeigneten Kommentaren versehen. Somit kann dann eine Programmdokumentation automatisch erstellt werden.

```
package primzahlen;

import java.math.BigInteger;
import java.util.Random;

/**
 * {@summary Hilfsmethoden zur Erzeugung großer Primzahlen.}
 */
public class Utils {
    private static final BigInteger NULL = new BigInteger("0");
    private static final BigInteger EINS = new BigInteger("1");
    private static final BigInteger ZWEI = new BigInteger("2");
```

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_18](https://doi.org/10.1007/978-3-658-43574-5_18).

```
/**  
 * {@summary Erzeugt per Zufallsgenerator eine n-stellige ganze Zahl.}  
 *  
 * @param n Anzahl Stellen  
 * @return n-stellige ganze Zahl  
 */  
public static BigInteger getZahl(int n) {  
    Random r = new Random();  
    StringBuilder s = new StringBuilder();  
  
    s.append(1 + r.nextInt(9));  
    for (int i = 1; i < n; i++)  
        s.append(r.nextInt(10));  
  
    return new BigInteger(s.toString());  
}  
  
/**  
 * {@summary Erzeugt ausgehend von der Zahl {@code start} die nächste Primzahl.}  
 *  
 * @param start Zahl, mit der die Berechnung beginnt  
 * @return die nächste Primzahl nach {@code start}  
 */  
public static BigInteger nextPrimzahl(BigInteger start) {  
    if (start.mod(ZWEI).equals(NULL))  
        start = start.add(EINS);  
    else  
        start = start.add(ZWEI);  
  
    if (start.isProbablePrime(100))  
        return start;  
    else  
        return nextPrimzahl(start);  
}  
}  
  
package primzahlen;  
  
import java.math.BigInteger;  
  
/**  
 * {@summary Test der Klasse Utils.}  
 * @see primzahlen.Utils Utils  
 */  
public class Test {  
    /**  
     * {@summary Erzeugt ausgehend von einer 30-stelligen Zahl  
     * die nächste Primzahl.}  
     *  
     * @param args wird nicht benötigt  
     */  
    public static void main(String[] args) {  
        int num = 30;  
  
        BigInteger start = Utils.getZahl(num);  
        System.out.println("start:\t" + start);  
    }  
}
```

```
        BigInteger next = Utils.nextPrimzahl(start);
        System.out.println("next:\t" + next);
    }
}
```

## 18.2 Das Werkzeug javadoc

### **Methodendetails**

#### **getZahl**

```
public static BigInteger getZahl(int n)
```

Erzeugt per Zufallsgenerator eine n-stellige ganze Zahl.

**Parameter:**

n - Anzahl Stellen

**Gibt zurück:**

n-stellige ganze Zahl

#### **nextPrimzahl**

```
public static BigInteger nextPrimzahl(BigInteger start)
```

Erzeugt ausgehend von der Zahl start die nächste Primzahl.

**Parameter:**

start - Zahl, mit der die Berechnung beginnt

**Gibt zurück:**

die nächste Primzahl nach start

**Abbildung 18-1:** Mit javadoc generierte HTML-Dokumentation

Die Programmdokumentation kann mit dem folgenden Kommando erstellt werden:

```
javadoc -public -d doc -sourcepath src -subpackages primzahlen
```

Die Ausgabedateien werden im Verzeichnis doc abgelegt. Mit `javadoc -help` werden alle Optionen für `javadoc` aufgelistet.

Viele Entwicklungsumgebungen (wie z. B. *IntelliJ IDEA*) bieten das Tool `javadoc` mit einer grafischen Oberfläche zur Auswahl von Optionen an.

## 18.3 Aufgaben

### Aufgabe 1

Versehen Sie den Quellcode des Service Provider aus Kapitel 17 mit javadoc-Kommentaren und erzeugen Sie die Dokumentation.

*Lösung: Paket de.test*



# 19 Generische Typen und Methoden

Klassen, Records, Interfaces und Methoden können mit Hilfe von formalen *Typparametern* (Platzhaltern) implementiert werden, die erst bei der Verwendung durch einen konkreten Typ ersetzt werden.

Der Typparameter repräsentiert zum Zeitpunkt der Implementierung noch einen unbekannten Typ. Man definiert also *Schablonen*, die erst durch Angabe von konkreten Typen bei ihrer Verwendung zu normalen Klassen, Records, Interfaces bzw. Methoden ausgeprägt werden.

Diese Möglichkeit nennt man *Generizität*. Der Begriff *Generics* ist ein Synonym hierfür. Generics werden in Java ausschließlich vom Compiler verarbeitet. Das Laufzeitsystem (JVM) arbeitet weiterhin mit "normalen" Klassen und Interfaces.

## Lernziele

In diesem Kapitel lernen Sie

- welche Vorteile generische Typen und Methoden haben,
- wie generische Klassen, Records, Interfaces und Methoden definiert werden und
- wie diese eingesetzt werden können.

## 19.1 Einführung

Um das Konzept "Generics" anschaulich zu machen, verwenden wir im Folgenden eine Klasse, die als Behälter für verschiedenartige Werte dienen soll.

Eine `Box`-Instanz kann beliebige Objekte aufnehmen, wie das folgende Beispiel zeigt:

```
package non_generic;

public class Box {
    private Object value;

    public void setValue(Object value) {
        this.value = value;
    }

    public Object getValue() {
        return value;
    }
}

package non_generic;

public class BoxTest {
```

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_19](https://doi.org/10.1007/978-3-658-43574-5_19).

```

public static void main(String[] args) {
    Box box1 = new Box();
    box1.setValue("Hugo Meier");

    Box box2 = new Box();
    box2.setValue(4711);

    System.out.println(((String) box1.getValue()).length());
    System.out.println(((Integer) box2.getValue()) + 1);
}
}

```

Um objektspezifische Methoden auf `Box`-Inhalte anwenden zu können, muss in der Regel der *Cast-Operator* verwendet werden.

## Zielsetzung

Ziel ist es, eine "Box" zu definieren, die nur Objekte eines *bestimmten* vorgegebenen Typs (z. B. immer nur Strings oder immer nur `int`-Werte) aufnehmen kann.

Die Einhaltung dieser Regel soll schon beim Compilieren des Programms geprüft werden. Für jeden denkbaren Typ einen eigenen `Box`-Typ (`StringBox`, `IntegerBox` usw.) zu definieren, ist als Lösung indiskutabel.

Man will zum Zeitpunkt der Übersetzung bereits sicherstellen, dass bei der Ausführung des Programms dieses *typsicher* abläuft.

Hierzu zunächst einige Definitionen.

## Generische Typen

Eine *generische Klasse* ist eine Klassen-Definition, in der unbekannte Typen (nur Referenztypen, keine einfachen Datentypen) durch Typparameter (Platzhalter) vertreten sind.

Dies gilt analog auch für *Records*.

Ein *generisches Interface* ist eine Interface-Definition, in der unbekannte Typen durch Typparameter vertreten sind.

Allgemein spricht man hier von *generischen Typen*.

Um *Typparameter* zu bezeichnen, werden üblicherweise einzelne Großbuchstaben verwendet (im Beispiel: `T`). Typparameter werden dem Klassen-, Record- bzw. Interface-Namen in spitzen Klammern hinzugefügt.

Beispiel:

```
class Box<T> { ... }
```

### Parametrisierter Typ

Sind die Typparameter einer generischen Klasse durch konkrete *Typargumente* ersetzt, spricht man von einer *parametrisierten Klasse*, allgemein von einem *parametrisierten Typ*. Ähnliches gilt für Records und Interfaces.

Beispiel:

```
Box<String> sbox = ...
```

Das folgende Programm ist die Lösung zu der obigen Problemstellung. Die hier erzeugte Box kann nur Objekte eines bestimmten Typs aufnehmen.

```
package generic;

public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

package generic;

public class BoxTest {
    public static void main(String[] args) {
        Box<String> box1 = new Box<>();
        box1.setValue("Hugo Meier");

        Box<Integer> box2 = new Box<>();
        box2.setValue(4711);

        System.out.println(box1.getValue().length());
        System.out.println(box2.getValue() + 1);

        // box1.setValue(4711);
    }
}
```

Die auf Kommentar gesetzte Anweisung

```
box1.setValue(4711);
```

führt zu einem Übersetzungsfehler.

Das Programm ist sogar einfacher geworden, da der *Downcast* auf *String* bzw. *Integer* nun nicht mehr nötig ist.

## Diamond Operator

Der Typparameter in der new-Anweisung kann bis auf die spitzen Klammern <> (*Diamond Operator*) weggelassen werden, z. B.

```
Box<String> box1 = new Box<>();
```

Der Compiler ermittelt den korrekten Typ aus dem Kontext. Verwendet man var, muss der Typ natürlich vorgegeben werden:

```
var box1 = new Box<String>();
```

Aus einer generischen Klasse werden auf diesem Weg viele unterschiedliche parametisierte Klassen "generiert". Zu einer generischen Klasse gehört genau eine Bytecode-Datei mit Endung .class. Diese wird von allen zugehörigen parametrisierten Klassen genutzt.

Generische Klassen können parametisierte oder generische Interfaces implementieren.

Beispiel:

```
class Box<T> implements Markierbar<String>
class Box<T> implements Markierbar<T>
```

Im letzten Fall wird der Typparameter T der Klasse an den Typparameter des Interfaces gekoppelt.

Mehrere Typparameter sind auch möglich.

Beispiel: class Demo<T, U>

## 19.2 Typeinschränkungen

Typen, die ein Typparameter T annehmen kann, können mit extends auf bestimmte Klassen bzw. Interfaces eingeschränkt werden.

### Typebound

Der rechts von extends stehende Typ wird als *Typebound* bezeichnet.

Beispiel:

```
class Box<T extends Number> { ... }
```

Hier kann Number<sup>1</sup> selbst und jede Subklasse von Number eingesetzt werden, also z. B. Integer.

<sup>1</sup> Subklassen der abstrakten Klasse Number sind z. B. Byte, Short, Integer, Long, Float, Double, BigInteger.

---

`extends` wird gleichermaßen für Klassen und Interfaces genutzt.

### Was heißt kompatibel?

Ein Typ  $\tau$  heißt *kompatibel* zum Typ  $u$ , wenn eine Instanz vom Typ  $\tau$  einer Variablen vom Typ  $u$  zugewiesen werden kann.

Ist  $\tau$  Subklasse von  $u$ , so ist  $\tau$  kompatibel zu  $u$ .

Beispiel:

```
Integer ist Subklasse von Number, also ist Integer kompatibel zu Number.  
Number number = Integer.valueOf(4711);
```

Typargumente in parametrisierten Typen müssen zum Typebound kompatibel sein.

Beispiel:

```
class Box<T extends Number> { ... }  
Box<Integer> box1 = new Box<>();  
Box<Double> box2 = new Box<>();
```

Falls eine explizite Angabe des Typebounds fehlt, ist `Object` der voreingestellte *Default-Typebound*.

`class Box<T>` und `class Box<T extends Object>` sind also äquivalent.

Der Typebound kann auch den Typparameter mit einbeziehen:

Beispiel:

```
class NewBox<T extends Comparable<T>>
```

Das Interface `Comparable<T>` deklariert die Methode

```
int compareTo(T obj)
```

Das folgende Programmbeispiel zeigt den Umgang mit Typebounds.

```
package typebound;  
  
public class Box<T extends Number> {  
    private T value;  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
  
    public T getValue() {  
        return value;  
    }  
}
```

```
package typebound;

public class NewBox<T extends Comparable<T>> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

package typebound;

public class Konto implements Comparable<Konto> {
    private int id;
    private int saldo;

    public Konto(int id, int saldo) {
        this.id = id;
        this.saldo = saldo;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getSaldo() {
        return saldo;
    }

    public void setSaldo(int saldo) {
        this.saldo = saldo;
    }

    ...

    @Override
    public int compareTo(Konto k) {
        return Integer.compare(id, k.id);
    }
}

package typebound;

public class BoxTest {
    public static void main(String[] args) {
        Box<Integer> ibox = new Box<>();
        ibox.setValue(4711);
        System.out.println(ibox.getValue());
```

```

Box<Double> dbox = new Box<>();
dbox.setValue(123.45);
System.out.println(dbox.getValue());

NewBox<Konto> box1 = new NewBox<>();
box1.setValue(new Konto(4711, 10000));

NewBox<Konto> box2 = new NewBox<>();
box2.setValue(new Konto(4712, 20000));

System.out.println(box1.getValue().compareTo(box2.getValue()));
}
}

```

Ausgabe:

```

4711
123.45
-1

```

Mehrfache Typebounds sind möglich:

`T extends Type1 & Type2 & Type3 & ...`

*Type1* kann ein beliebiger Typ (Klasse oder Interface) sein, *Type2*, *Type3* usw. dürfen nur Interfaces sein.

Der für *T* eingesetzte Typ ist von *Type1* abgeleitet und implementiert die Interfaces *Type2*, *Type3* usw.

bzw.

der für *T* eingesetzte Typ implementiert die Interfaces *Type1*, *Type2*, *Type3* usw.

## Invarianz

Aus "A ist kompatibel zu B" folgt *nicht* "C<A> ist kompatibel zu C<B>".

Die Kompatibilität der Typargumente überträgt sich also *nicht* auf die parametrisierten Typen (*Invarianz*).<sup>2</sup>

So ist beispielsweise die Zuweisung

```
Box<Number> box = new Box<Integer>();
```

*nicht* möglich. Wäre dies möglich, so könnte man z. B. eine double-Zahl der Box hinzufügen:

```
box.setValue(1.23);
```

Somit würde die Box vom Typ `Integer` einen unerlaubten Wert enthalten.

---

<sup>2</sup> Das Begleitmaterial zu diesem Buch enthält Codebeispiele zur Definition der hier vorgestellten Varianzarten (siehe Paket *demo*).

Invarianz verhindert hier also, dass Objekte, die nicht vom Typ `Integer` sind, durch bloße "Umetikettierung" der Box hinzugefügt werden können.

Bei Arrays ist das Verhalten jedoch anders:

`Integer` ist kompatibel zu `Number`, ebenso ist ein `Integer`-Array kompatibel zu einem `Number`-Array.

Beispiel:

```
Number[] a = new Integer[1];
a[0] = 3.14;
```

Dieser Code lässt sich fehlerfrei compilieren. Erst zur Laufzeit wird die Ausnahme `ArrayStoreException` ausgelöst. Die Elementtypen von Arrays werden also erst zur Laufzeit überprüft.

## 19.3 Raw Types

### Type-Erasure

Beim Übersetzen wird generischer Code mit Typparametern und Typargumenten auf normalen, nicht-generischen Code reduziert (*Type-Erasure*).

Typparameter werden durch `Object` oder den ersten Typebound ersetzt.

Informationen über Typparameter und Typebounds werden als Metadaten in den Bytecode eingebettet. Diese Metadaten werden beim Übersetzen der Anwendung vom Compiler für Typprüfungen wieder ausgewertet. Code, der mit der Instanz eines parametrisierten Typs arbeitet, wird vom Compiler automatisch um die notwendigen Typumwandlungen (Casts) erweitert.

Das Laufzeitsystem verarbeitet zum Zeitpunkt der Ausführung eines Programms also ganz normalen nicht-generischen Code.

### Raw Type

Durch *Type-Erasure* entsteht der sogenannte *Raw Type* einer generischen Klasse.

Generische Klassen und Interfaces können aus Gründen der Abwärtskompatibilität ohne Typparameter auch "nicht-generisch" genutzt werden.

Beispiel:

```
class Box<T> { ... }
Box box = new Box();
```

Der Compiler erzeugt lediglich eine Warnung.

Ein parametrisierter Typ ist zu seinem *Raw Type* kompatibel.

Beispiel:

```
Box box = new Box<Integer>();
```

---

```
package raw_type;

public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

---

```
package raw_type;

public class BoxTest {
    public static void main(String[] args) {
        Box box = new Box();
        box.setValue("Hugo");
        System.out.println(box.getValue());

        Box<Integer> ibox = new Box<>();
        ibox.setValue(5000);

        box = ibox;
        box.setValue("Emil");
        System.out.println(ibox.getValue());
    }
}
```

Die Integer-Box ibox enthält am Ende den String "Emil". Von der Nutzung des *Raw Type* ist also abzuraten:

*Unchecked call to 'setValue(T)' as a member of raw type 'raw\_type.Box'*

## 19.4 Wildcards

Ein parametrisierter Typ kann ein *unbestimmtes* Typargument nennen, z. B.

```
Box<?> box;
```

Dieses unbestimmte Typargument wird durch das *Wildcard-Zeichen* ? symbolisiert.

Die Verwendung von *Wildcard-Typen* ermöglicht flexiblen Code:

Wird ein Wildcard-Typ als Parametertyp einer Methode genutzt, so ist diese Methode nicht auf Argumente nur eines einzigen parametrisierten Typs beschränkt.

Auch für Wildcard-Typen können Einschränkungen formuliert werden. Dabei sind diverse Regeln zu berücksichtigen. Diese werden im Folgenden anhand von Beispielen erläutert:

### Bivarianz, Covarianz und Contravarianz

#### Bivarianz

Zu einem unbeschränkten *Wildcard-Typ* `C<?>` sind alle parametrisierten Typen des gleichen generischen Typs `C<T>` kompatibel (*Bivarianz*).

Beispiel:

Alle parametrisierten "Box-Typen" (z. B. `Box<Integer>`, `Box<String>`) sind zu `Box<?>` kompatibel.

Aber: Weder ein *lesender* noch ein *schreibender* Zugriff auf eine Instanz eines solchen Wildcard-Typs ist erlaubt, wie das nächste Programm zeigt.

Einige Ausnahmen sind:

Empfangstyp `Object` beim Lesen, Zuweisung von `null` beim Schreiben.

```
package bivarianz;

public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

package bivarianz;

public class BoxTest {
    public static void main(String[] args) {
        Box<String> sbox = new Box<>();
        sbox.setValue("Hugo Meier");
        print(sbox);

        Box<Integer> ibox = new Box<>();
        ibox.setValue(4711);
        print(ibox);

        Box<?> box = ibox;
        box.setValue(null);
        print(box);
    }
}
```

---

```

public static void print(Box<?> box) {
    Object obj = box.getValue();
    if (obj == null)
        System.out.println("Box ist leer");
    else
        System.out.println("Inhalt: " + obj);
}
}

```

## Obere Grenze und Covarianz

Wildcard-Typen können durch eine *obere Grenze* eingeschränkt werden (*Upper-Typebound*):

`C<? extends B>`

Ist `B` eine Klasse, so muss das Typargument entweder `B` selbst oder eine von `B` abgeleitete Klasse sein. Ist `B` ein Interface, so muss das Typargument `B` implementieren.

Mehrfache Upper-Typebounds sind auch möglich, z. B.

`C<? extends K1 & I1 & I2>`

Ist `A` kompatibel zu `B`, so ist `C<A>` kompatibel zu `C<? extends B>` (*Covarianz*).

Auf eine Instanz eines solchen Wildcard-Typs ist kein *schreibender* Zugriff erlaubt. Einzige Ausnahme ist die Zuweisung von `null`. *Lesende* Operationen können sich auf den Upper-Typebound als Empfangstyp beziehen.

```

package covarianz;

public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public void copyFrom(Box<? extends T> other) {
        value = other.getValue(); // lesender Zugriff
    }
}

```

```

package covarianz;

public class BoxTest {
    public static void main(String[] args) {
        Box<Integer> ibox = new Box<>();

```

```

ibox.setValue(4711);

Box<? extends Number> box = ibox;
Number n = box.getValue();
System.out.println(n);
box.setValue(null);

Box<Number> nbox = new Box<>();

ibox.setValue(5000);
nbox.copyFrom(ibox);
System.out.println(nbox.getValue());

Box<Double> dbox = new Box<>();
dbox.setValue(123.45);
nbox.copyFrom(dbox);
System.out.println(nbox.getValue());
}
}

```

`Box<Integer>` und `Box<Double>` sind beide kompatibel zu `Box<? extends Number>`. Der Parameter `other` der `Box`-Methode `copyFrom` ist ein *T-Produzent*, d. h. es wird ein Wert aus der Box `other` gelesen, um einen T-Wert zu erzeugen.

## Untere Grenze und Contravarianz

Wildcard-Typen können durch eine untere Grenze eingeschränkt werden (*Lower-Typebound*):

```
C<? super A>
```

Als Typargument sind hier nur die Klasse `A` selbst oder Klassen, die Superklassen von `A` sind, zulässig.

Ist `A` kompatibel zu `B`, so ist `C<B>` kompatibel zu `C<? super A>` (*Contravarianz*).

Auf eine Instanz eines solchen Wildcard-Typs ist kein *lesender* Zugriff erlaubt. Einzige Ausnahme ist das Lesen mit Empfangstyp `Object`.

*Schreibende* Operationen können sich auf den Lower-Typebound beziehen.

```

package contravarianz;

public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }
}
```

```

public T getValue() {
    return value;
}

public void copyTo(Box<? super T> other) {
    other.setValue(value); // schreibender Zugriff
}
}

package contravarianz;

public class BoxTest {
    public static void main(String[] args) {
        Box<Object> obox = new Box<>();
        obox.setValue("Hugo");

        Box<? super Integer> box = obox;
        Object obj = box.getValue();
        System.out.println(obj);
        box.setValue(4711);

        Box<Integer> ibox = new Box<>();
        ibox.setValue(4711);

        Box<Number> nbox = new Box<>();
        ibox.copyTo(nbox);
        System.out.println(nbox.getValue());

        ibox.copyTo(obox);
        System.out.println(obox.getValue());
    }
}

```

`Box<Number>` und `Box<Object>` sind beide kompatibel zu `Box<? super Integer>`.

Der Parameter `other` der `Box`-Methode `copyTo` ist ein *T-Konsument*, d. h. es wird ein `T`-Wert an eine `other`-Methode übergeben und "konsumiert".

### Zwei Merkregeln:

*PECS-Prinzip: Producer extends, Consumer super*

Repräsentiert ein parametrisierter Typ einen *T-Produzenten*, wird `<? extends T>` verwendet, repräsentiert er einen *T-Konsumenten*, wird `<? super T>` verwendet.<sup>3</sup>

*LESS-Prinzip: Lesen = extends, Schreiben = super*

Wird `<? extends T>` verwendet, ist Lesen erlaubt, Schreiben jedoch nicht. Bei `<? super T>` ist Schreiben erlaubt, Lesen jedoch nicht.

---

<sup>3</sup> Bloch, J.: Effective Java. dpunkt.verlag, 3. Auflage 2018

Tabelle 19-1 gibt einen Überblick über die verschiedenen Regeln und die möglichen Lese- und Schreibzugriffe.

**Tabelle 19-1:** Wildcard-Typen

Wildcard-Typ	Varianz	Lesen	Schreiben
?	Bivarianz	- <sup>1)</sup>	- <sup>2)</sup>
? extends T	Covarianz	+ <sup>3)</sup>	- <sup>2)</sup>
? super T	Contravarianz	- <sup>1)</sup>	+ <sup>3)</sup>

1) Ausnahme: Empfangstyp Object 2) Ausnahme: Zuweisung von null 3) bezogen auf T

### Bemerkung zu Arrays

Arrays für generische Klassen können ausschließlich mit dem Wildcard-Zeichen ? erzeugt werden.

Beispiel:

```
Box<?>[] boxes = new Box<?>[3];
```

## 19.5 Generische Methoden

Methoden können nicht nur die Typparameter ihrer Klasse nutzen, sondern auch eigene Typparameter definieren. Solche Methoden werden als *generische Methoden* bezeichnet.

Im Methodenkopf einer generischen Methode steht die Liste der eigenen Typparameter (ggf. mit Typebounds) in spitzen Klammern vor dem Rückgabetyp.

Beispiel:

```
static <E> void exchange(Box<E> a, Box<E> b)
```

Beim Aufruf einer generischen Methode wird das Typargument unmittelbar vor den Methodennamen gesetzt:

```
Box.<Integer>exchange(box1, box2);
```

Bei Instanzmethoden muss die Objektreferenz angegeben werden. Statische Methoden müssen mit dem Klassennamen angesprochen werden.

In vielen Fällen können konkrete Typargumente weggelassen werden, weil der Compiler die Typargumente aus dem Kontext des Methodenaufrufs bestimmen kann (*Typ-Inferenz*):

```
Box.exchange(box1, box2);
```

Die Klasse `Utils` im folgenden Beispiel enthält eine generische Klassenmethode, die die Inhalte zweier Boxen untereinander austauscht.

```
package methods;

public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

package methods;

public class Utils {
    public static <E> void exchange(Box<E> a, Box<E> b) {
        E value = b.getValue();
        b.setValue(a.getValue());
        a.setValue(value);
    }
}

package methods;

public class BoxTest {
    public static void main(String[] args) {
        Box<Integer> box1 = new Box<>();
        box1.setValue(1000);
        Box<Integer> box2 = new Box<>();
        box2.setValue(2000);

        System.out.println("Box1: " + box1.getValue());
        System.out.println("Box2: " + box2.getValue());

        Utils.exchange(box1, box2);

        System.out.println("Box1: " + box1.getValue());
        System.out.println("Box2: " + box2.getValue());
    }
}
```

## 19.6 Grenzen des Generics-Konzepts

Zusammenfassend werden einige *Einschränkungen im Umgang mit Generics* aufgezeigt:

- Als Typargumente sind ausschließlich Referenztypen (wie Klassen, Interfaces, Arrays) geeignet. Einfache Datentypen wie `int` und `double` sind nicht erlaubt.
- Klassenattribute und Klassenmethoden einer generischen Klasse können keine Typparameter der eigenen Klasse verwenden. Eigene Typparameter sind jedoch möglich (generische Methoden).
- Der Operator `instanceof` kann nicht mit Typparametern bzw. parametrisierten Typen verwendet werden.
- Konstruktoraufnäufe mit Typparametern können nicht verwendet werden: `new T()` bzw. `new T[100]` ist nicht erlaubt.
- Typparameter können nicht als Basistyp bei Vererbung benutzt werden: `class Xxx<T> extends T` ist nicht erlaubt.
- Typparameter sind in `catch`-Klauseln (Exception-Handling) nicht erlaubt.

## 19.7 Aufgaben

### Aufgabe 1

Definieren Sie die generische Klasse `Pair<T,U>`, die Werte zweier unterschiedlicher Typen aufnehmen kann. Über passende get-Methoden sollen diese Werte abgefragt werden können. Definieren Sie für spätere Zwecke die beiden Instanzvariablen als `protected`.

*Lösung: Paket pair*

### Aufgabe 2

Definieren Sie das generische Interface `Markable<S>`, das die beiden folgenden Methoden beinhaltet:

```
void setMark(S m);
S getMark();
```

Die Klasse `Pair<T,U>` aus Aufgabe 1 soll dieses Interface mit dem Typargument `String` implementieren.

Die Klasse `Box<T>` aus Kapitel 19.1 soll ebenso dieses Interface implementieren, wobei der Typparameter `T` an den Typparameter des Interfaces gekoppelt werden soll.

*Lösung: Paket mark*

**Aufgabe 3**

Definieren Sie die generische Klasse `PairOfNumbers`, die von der Klasse `Pair<T,U>` aus Aufgabe 1 abgeleitet ist und deren Typargumente zu `Number` kompatibel sein müssen. Die abstrakte Klasse `Number` enthält u. a. die abstrakte Methode

```
double doubleValue(),
```

die z. B. von den Klassen `Integer`, `Double` und `BigInteger` implementiert wird.

Die Klasse `PairOfNumbers` soll eine Methode enthalten, die die Summe der beiden Zahlen des Paars als double-Zahl zurückgibt.

*Lösung: Paket numbers*

**Aufgabe 4**

Implementieren Sie die Methode

```
public static void print(Pair<?,?> p),
```

die die beiden Werte `a` und `b` eines Paars in der Form `(a, b)` ausgibt.

Nutzen Sie die Klasse `Pair<T,U>` aus Aufgabe 1.

*Lösung: Paket print*

**Aufgabe 5**

Implementieren Sie die Methode

```
public static double sum(Pair<? extends Number, ? extends Number> p),
```

die die beiden Werte eines Paars addiert. Nutzen Sie die Klasse `Pair<T,U>` aus Aufgabe 1 und die `Number`-Methode `double doubleValue()`.

*Lösung: Paket sum*

**Aufgabe 6**

Lösen Sie Aufgabe 5 durch Implementierung einer generischen Methode (ohne Verwendung von Wildcard-Typen).

*Lösung: Paket sum2*

**Aufgabe 7**

Entwickeln Sie eine Klasse `Kontakt` mit den Attributen `vorname`, `nachname` und `telnr`. Die Klasse soll `Comparable<Kontakt>` so implementieren, dass Kontakte nach Nachnamen und Vornamen sortiert werden können.

*Lösung: Paket kontakt*

**Aufgabe 8**

Definieren Sie analog zu Aufgabe 1 einen Record `Pair<A, B>`. Implementieren Sie zusätzlich die folgenden generischen, statischen Methoden:

```
static <X, Y> Pair<X, Y> of(X x, Y y)
static <T> Pair<T, T> fromList(List<? extends T> list)
static <T> List<T> toList(Pair<T, T> pair)
```

Die ersten beiden Methoden erzeugen ein Paar aus zwei Werten bzw. einer 2-elementigen Liste. Die letzte Methode konvertiert ein Paar in eine Liste `List<T>`.

*Lösung: Paket pair2*

**Aufgabe 9**

Das Interface `Ordered` erweitert das Interface `Comparable`:

```
interface Ordered<T> extends Comparable<T>
```

und definiert die Vergleichsmethoden `less`, `lessEqual`, `greater` und `greaterEqual` als Default-Methoden.

Testen Sie diese Methoden mit der Klasse `Person`:

```
class Person implements Ordered<Person>
```

*Lösung: Paket ordered*

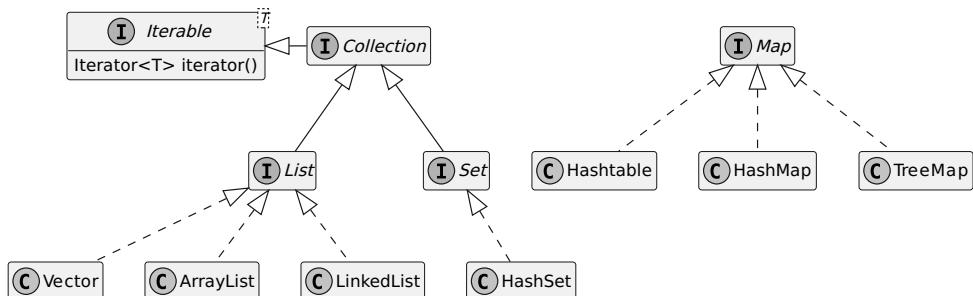
## 20 Collections

Alle Klassen und Interfaces des *Collections Framework* sind als generische Typen definiert. Hierdurch erreicht man, dass alle Elemente eines Daten-Containers vom gleichen vorgegebenen Typ sind. *Collections* bieten einen typsicherer und effizienten Zugriffe auf die gespeicherten Objekte.

*Collections* können grob in drei Arten unterteilt werden:

- Listen,
- Mengen und
- Schlüsseltabellen (Key-Value-Speicher).

Alle Listen und Mengen implementieren das Interface `java.util.Collection<T>`. Schlüsseltabellen implementieren das Interface `java.util.Map<K, V>`.



**Abbildung 20-1:** Wichtige Collections-Klassen

### Lernziele

In diesem Kapitel lernen Sie

- verschiedene Interfaces des Collections Framework und die zugehörigen Klassen kennen sowie
- den Umgang mit Listen, Mengen und Schlüsseltabellen.

### 20.1 Die Interfaces Collection, Iterable und Iterator

Zunächst einige `Collection`-Methoden, die für Listen und Mengen zur Verfügung stehen:

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_20](https://doi.org/10.1007/978-3-658-43574-5_20).

```
boolean add(E e)
    fügt e in den Container ein.

void clear()
    entfernt alle Elemente.

boolean contains(Object obj)
    prüft, ob obj im Container enthalten ist.

boolean isEmpty()
    prüft, ob der Container keine Elemente enthält.

boolean remove(Object obj)
    entfernt obj.

int size()
    liefert die Anzahl der Elemente im Container.

Object[] toArray()
    liefert die Elemente des Containers als Array zurück.

<T> T[] toArray(T[] a)
    liefert die Elemente des Containers als Array zurück. Wenn alle Elemente in das angegebene Array a passen, werden sie darin zurückgegeben. Andernfalls wird ein neues Array mit der Größe des Containers zugewiesen.
```

`Iterator<T> iterator()`  
liefert einen `Iterator` zum schrittweisen Durchlaufen des Containers. Diese Methode ist im Interface `java.lang.Iterable<T>` enthalten, das von `Collection` erweitert wird.

Das Interface `java.util.Iterator<T>` hat die folgenden Methoden:

```
boolean hasNext()
    prüft, ob es noch weitere nicht besuchte Elemente gibt.

T next()
    liefert das nächste Element oder löst eine NoSuchElementException aus.

void remove()
    entfernt das zuletzt besuchte Element.
```

Ein `Iterator`-Objekt kann nur einmal verwendet werden. Um einen Container erneut zu durchlaufen, muss ein neues `Iterator`-Objekt erzeugt werden.

Alle Container, die das Interface `Iterable` implementieren, können mit *foreach*-Schleifen durchlaufen werden:

```
for (Typ name : c) { ... }

Typ entspricht hier dem Elementtyp der Collection c.
```

## 20.2 Listen

Eine *Liste* ist eine sequentielle Anordnung von Elementen in einer festen Reihenfolge.

### Das Interface List

Das Interface `java.util.List<E>` erweitert `Collection<E>`.

`add` aus `Collection` fügt ein Element am Ende der Liste ein.

Einige `List`-Methoden:

`void add(int i, E e)`

fügt `t` an der Position `i` ein.

`E get(int i)`

liefert das Element an der Position `i`.

`E set(int i, E e)`

ersetzt das Element an der Position `i` durch `e` und liefert das alte Element zurück.

`E remove(int i)`

entfernt das Element an der Position `i` und liefert es zurück.

`boolean remove(Object obj)`

entfernt das erste passende Element und liefert `true`, falls `obj` in der Liste enthalten war.

`int indexOf(Object obj)`

liefert die Position des ersten Auftritts von `obj` in der Liste; liefert den Wert `-1`, wenn `obj` nicht in der Liste enthalten ist.

`int lastIndexOf(Object obj)`

liefert die Position des letzten Auftritts von `obj` in der Liste; liefert den Wert `-1`, wenn `obj` nicht in der Liste enthalten ist.

`static <T> List<T> copyOf(Collection<? extends T> c)`

erzeugt eine unveränderliche Kopie von `c`.

### Vector

Die bereits im Kapitel 15.3 ohne Berücksichtigung der Generizität behandelte Klasse `java.util.Vector<E>` implementiert `List<E>`.

`Vector(Collection<? extends E> c)`

erzeugt ein `vector`-Objekt mit den Elementen der Collection `c`.

## ArrayList

Die Klasse `java.util.ArrayList<E>` implementiert `List<E>`, wobei intern ein Array verwendet wird.

Der Konstruktor `ArrayList()` erzeugt eine leere Liste.

`ArrayList(Collection<? extends E> c)`  
erzeugt eine Liste aus den Elementen von `c`.

Die `java.util.Arrays`-Methode

```
static <T> List<T> asList(T... a)
```

erzeugt eine Liste aus einem Array. Nachträgliche Elementänderungen im Array sind in der Liste sichtbar und umgekehrt.

Das folgende Programm zeigt zwei Varianten zum Durchlaufen einer Liste, das Kopieren einer Liste und Änderungen am Original sowie die Konvertierung zwischen Liste und Array.

```
package list;

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

public class ListTest {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Anton");
        list.add("Emil");
        list.add("Fritz");
        list.add("Hugo");

        // Iteration Variante 1
        Iterator<String> it = list.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
        System.out.println();

        // Iteration Variante 2
        for (String name : list) {
            System.out.println(name);
        }
        System.out.println();

        // Kopieren und Original-Liste ändern
        System.out.println(list);
        List<String> copy = List.copyOf(list);
        list.add(3, "Gustav");
        list.remove(0);
        System.out.println(list);
        System.out.println(copy);
        System.out.println();
    }
}
```

```
// Konvertieren: List --> Array
String[] array = list.toArray(new String[0]);
System.out.println(Arrays.toString(array));

// Konvertieren: Array --> List
List<String> list2 = Arrays.asList(array);
System.out.println(list2);
array[0] = "Tim";
System.out.println(list2);
list2.set(0, "Pit");
System.out.println(Arrays.toString(array));
}
}
```

Ausgabe des Programms:

```
Anton
Emil
Fritz
Hugo
```

```
Anton
Emil
Fritz
Hugo
```

```
[Anton, Emil, Fritz, Hugo]
[Emil, Fritz, Gustav, Hugo]
[Anton, Emil, Fritz, Hugo]

[Emil, Fritz, Gustav, Hugo]
[Emil, Fritz, Gustav, Hugo]
[Tim, Fritz, Gustav, Hugo]
[Pit, Fritz, Gustav, Hugo]
```

## LinkedList

Die Klasse `java.util.LinkedList<E>` implementiert eine doppelt verkettete Liste. Jedes Element hat einen Verweis auf seinen Vorgänger und seinen Nachfolger. Diese doppelte Verkettung macht das Einfügen und Entfernen von Elementen besonders effizient.

Es stehen für diese Klasse besondere Methoden zur Verfügung:

```
void addFirst(E e), void addLast(E e), E getFirst(), E getLast(),
E removeFirst(), E removeLast()
```

E `peek()`  
liefert das erste Element der Liste.

E `poll()`  
liefert und löscht das erste Element der Liste.

E `pop()`  
liefert das erste Element der Liste.

```
void push(E e)
```

fügt das Element e am Anfang der Liste ein.

LinkedList kann als *FIFO*-Speicher (First In – First Out) oder als *LIFO*-Speicher (Last In – First Out, auch *Stack* genannt) genutzt werden.

Hierzu gibt es eine Aufgabe am Ende dieses Kapitels.

## 20.3 Mengen

Eine *Menge* enthält im Gegensatz zur Liste keine Duplikate und es gibt keine bestimmte Reihenfolge für ihre Elemente.

### Das Interface Set

Das Interface `java.util.Set<E>` erweitert `Collection<E>`.

`add` aus `Collection` fügt ein Element in die Menge ein, falls es nicht bereits vorhanden ist.

```
boolean addAll(Collection<? extends E> c)
```

fügt alle Elemente aus c ein, falls sie noch nicht vorhanden sind.

```
static <E> Set<E> copyOf(Collection<? extends E> c)
```

erzeugt eine unveränderliche Kopie von c (ohne Duplikate).

### HashSet

Die Klasse `java.util.HashSet<E>` implementiert `Set<E>`.

Der Konstruktor `HashSet()` erzeugt eine leere Menge.

```
HashSet(Collection<? extends E> c)
```

erzeugt eine neue Menge aus den Elementen von c.

In einer Menge existieren keine zwei verschiedenen Elemente `e1` und `e2` mit `e1.equals(e2) == true`.

#### Achtung:

Beim Einfügen von Elementen in eine Menge können Doppelgänger nur erkannt und dann eliminiert werden, wenn die Klasse der Elemente geeignete Methoden `equals` und `hash` enthalten.

Set-Objekte können keine nachträglichen Änderungen an schon eingefügten Objekten überwachen. Wird ein eingefügtes Objekt nachträglich so geändert, dass es mit einem anderen Objekt der Menge (im Sinne von `equals`) übereinstimmt, gibt es einen Doppelgänger.

Hierzu gibt es jeweils eine Aufgabe am Ende dieses Kapitels.

Das folgende Programm erzeugt Lottozahlen (6 aus 49). Hier wird die "Mengeneigenschaft" von `add` ausgenutzt: Eine bereits in der Menge vorhandene Zahl wird nicht nochmals eingefügt.

```
package set;

import java.util.Arrays;
import java.util.HashSet;
import java.util.Random;
import java.util.Set;

public class Lottozahlen {
    public static void main(String[] args) {
        Random random = new Random();
        Set<Integer> lottozahlen = new HashSet<>();

        do {
            lottozahlen.add(1 + random.nextInt(49));
        } while (lottozahlen.size() < 6);

        Object[] lotto = lottozahlen.toArray();
        Arrays.sort(lotto);
        System.out.println(Arrays.toString(lotto));
    }
}
```

Ausgabe des Programms (Beispiel):

```
[2, 6, 7, 32, 35, 36]
```

## 20.4 Schlüsseltabellen

Eine *Schlüsseltabelle* ist eine Menge von Schlüssel-Wert-Paaren. Schlüssel müssen eindeutig sein.

### Das Interface Map

Alle Schlüsseltabellen implementieren das Interface `java.util.Map<K,V>`. Der Typparameter `K` steht für den Schlüsseltyp, `V` für den Werttyp.

Um die Gleichheit von Schlüsselobjekten festzustellen, wird intern die Methode `equals` genutzt. Die Methode `hashCode` wird genutzt, um die internen Speicherstrukturen effizient zu organisieren.

Einige Methoden:

```
void clear()
    entfernt alle Einträge.
```

```
boolean containsKey(Object key)
    prüft, ob key als Schlüssel in der Schlüsseltabelle enthalten ist.
```

```

boolean containsValue(Object value)
    prüft, ob value als Wert in der Schlüsseltabelle enthalten ist.

V get(Object key)
    liefert den Wert zum Schlüssel key.

default V getOrDefault(Object key, V defaultValue)
    liefert den Wert zum angegebenen Schlüssel oder defaultValue, wenn der Wert
    in der Schlüsseltabelle nicht enthalten ist.

V put(K key, V value)
    fügt das Schlüssel-Wert-Paar (key, value) ein. Falls der Schlüssel bereits in der
    Tabelle vorhanden ist, wird der alte Wert zurückgeliefert, sonst null.

V remove(Object key)
    entfernt den Schlüssel key und den zugeordneten Wert und liefert diesen Wert
    zurück oder null, falls key als Schlüssel nicht existiert.

boolean isEmpty()
    prüft, ob die Schlüsseltabelle leer ist.

int size()
    liefert die Anzahl der vorhandenen Schlüssel-Wert-Paare.

Set<K> keySet()
    liefert alle Schlüssel als Menge.

Collection<V> values()
    liefert die Werte der Schlüsseltabelle als Collection-Objekt.

static <K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> map)
    erzeugt eine unveränderliche Kopie der Einträge von map.

```

Die von den Methoden `keySet` und `values` zurückgegebenen Objekte sind *Sichten* auf die Schlüsseltabelle.

Änderungen der Schlüsseltabelle führen zu Änderungen dieser Sichten. Entfernen aus der Schlüssel- bzw. Wertemenge führt zum Entfernen des Eintrags aus der Schlüsseltabelle.

```

Set<Map.Entry<K,V>> entrySet()
    liefert die Menge der Einträge (Schlüssel-Wert-Paare) in der Schlüsseltabelle.

```

Das Interface `java.util.Map.Entry<K,V>` ist inneres Interface von `Map` und enthält die Methoden:

```

K getKey()
    liefert den Schlüssel des Eintrags.

V getValue()
    liefert den Wert des Eintrags.

```

## Hashtable

Die bereits im Kapitel 15.3 ohne Berücksichtigung der Generizität behandelte Klasse `java.util.Hashtable<K,V>` implementiert `Map<K,V>`.

## HashMap

Die Klasse `java.util.HashMap<K,V>` implementiert `Map<K,V>`.

Der Konstruktor `HashMap()` erzeugt eine leere Tabelle.

## TreeMap

Die Klasse `java.util.TreeMap<K,V>` implementiert eine Map, die nach Schlüsseln aufsteigend sortiert ist. Bei Verwendung des einfachen Konstruktors muss der Schlüssel das Interface `Comparable` implementieren.

Das folgende Programm baut eine Gehaltstabelle aus den Schlüssel-Wert-Paaren (Name, Gehalt) auf. Diese Tabelle wird auf zwei unterschiedliche Arten durchlaufen.

```
package map;

import java.util.Map;
import java.util.Set;

public class MapTest {
    public static void main(String[] args) {
        Map<String, Double> map = new TreeMap<>();
        map.put("Meier", 5000.);
        map.put("Schmitz", 4500.);
        map.put("Balder", 4700.);
        map.put("Schulze", 4500.);

        System.out.println(map);

        // Variante 1
        for (String key : map.keySet())
            System.out.println(key + ": " + map.get(key));

        // Variante 2
        for (Map.Entry<String, Double> e : map.entrySet())
            System.out.println(e.getKey() + ": " + e.getValue());
    }
}
```

Ausgabe des Programms:

```
{Balder=4700.0, Meier=5000.0, Schmitz=4500.0, Schulze=4500.0}
Balder: 4700.0
Meier: 5000.0
Schmitz: 4500.0
Schulze: 4500.0
```

```
Balder: 4700.0
Meier: 5000.0
Schmitz: 4500.0
Schulze: 4500.0
```

## 20.5 Erzeugung von Collections für vordefinierte Werte

Um einige vordefinierte Werte in eine Liste aufzunehmen, gibt es mehrere Möglichkeiten:

- Man nutzt die `List`-Methode `add` für jeden einzelnen Wert oder
- man verwendet die Arrays-Methode `static <T> List<T> asList(T... a)`.

Seit Java-Version 9 stehen weitere Methoden zur Verfügung:

`List`-Methode

```
static <E> List<E> of(E... elements)
```

`Set`-Methode

```
static <E> Set<E> of(E... elements)
```

Enthält die Menge Duplikate, wird eine `IllegalArgumentException` ausgelöst.

`Map`-Methode

```
static <K,V> Map<K,V> ofEntries(
    Map.Entry<? extends K,? extends V>... entries)
```

Bei doppelten Schlüsseln wird eine `IllegalArgumentException` ausgelöst.

Ein `Map`-Eintrag kann mit der `Map`-Methode `entry` erzeugt werden:

```
static <K,V> Map.Entry<K,V> entry(K k, V v)
```

```
package of_test;

import java.util.List;
import java.util.Map;
import java.util.Set;

public class Test {
    public static void main(String[] args) {
        List<String> list = List.of("Hugo", "Emil", "Tim");
        System.out.println(list);

        Set<String> set = Set.of("Hugo", "Emil", "Tim");
        System.out.println(set);
```

```
    Map<Integer, String> map = Map.ofEntries(  
        Map.entry(1, "Hugo"),  
        Map.entry(2, "Emil"),  
        Map.entry(3, "Tim"));  
    System.out.println(map);  
}  
}
```

Generell gilt:

Nachträgliche Änderungen an den mit `of` bzw. `ofEntries` erzeugten Containern sind nicht erlaubt.

## 20.6 Sequenced Collections

In Java 21 wurde das Interface `SequencedCollection` eingeführt, das in der Hierarchie zwischen den Interfaces `Collection` und `List` liegt.

`SequencedCollection` definiert u. a. Methoden

- zum Zugriff auf das erste und letzte Element:  
`E getFirst()` und `E getLast()`
- zum Einfügen und Anhängen eines Elements am Anfang bzw. an das Ende der Liste:  
`void addFirst(E)` und `void addLast(E)`
- zum Entfernen des ersten und letzten Elements:  
`E removeFirst()` und `E removeLast()`

Für geordnete Maps - wie z. B. `TreeMap` - wurde das vom Interface `Map` abgeleitete Interface `SequencedMap` eingeführt.

Es bietet u. a. Methoden

- zum Zugriff auf den ersten und letzten Eintrag:  
`Entry<K, V> firstEntry()` und `Entry<K, V> lastEntry()`
- zum Einfügen und Anhängen eines Eintrags am Anfang bzw. an das Ende der Map:  
`V putFirst(K, V)` und `V putLast(K, V)`
- zum Entfernen des ersten und letzten Eintrags:  
`Entry<K, V> pollFirstEntry()` und `Entry<K, V> pollLastEntry()`

## 20.7 Aufgaben

### Aufgabe 1

Erzeugen Sie eine Instanz der generischen Klasse `ArrayList`, die Objekte vom Typ `Konto` (mit `int id` und `double saldo`) speichert. `Konto` muss `Comparable` implementieren.

Fügen Sie verschiedene Instanzen in die Liste ein und sortieren Sie diese dann aufsteigend nach Kontonummern mit Hilfe der generischen Klassenmethode `sort` der Klasse `java.util.Collections`:

```
static <E extends Comparable<? super E>> void sort(List<E> list)
```

*Lösung: Paket konten*

### Aufgabe 2

Nutzen Sie die Klasse `LinkedList` als *FIFO*-Speicher sowie als *LIFO*-Speicher. Implementieren Sie zwei Testszenarien.

*Lösung: Paket fifo\_lifo*

### Aufgabe 3

Simulieren Sie eine einfache Druckerwarteschlange (`PrintQueue`). Ein Dokument (Klasse `Document` mit den Attributen `Titel` und `Seitenzahl`) kann in die Warteschlange eingefügt werden. Die `PrintQueue`-Methode `process` nimmt das erste Element aus der Schlange und druckt es (hier nur Meldung auf den Bildschirm).

*Lösung: Paket queue*

### Aufgabe 4

Beim Einfügen von Elementen in eine Menge können Doppelgänger nur erkannt und dann eliminiert werden, wenn die Klasse der Elemente geeignete Methoden `equals` und `hash` enthalten. Zeigen Sie das an einem Beispiel.

*Lösung: Paket set1*

### Aufgabe 5

Set-Objekte können keine nachträglichen Änderungen an schon eingefügten Objekten überwachen. Wird ein eingefügtes Objekt nachträglich so geändert, dass es mit einem anderen Objekt der Menge (im Sinne von `equals`) übereinstimmt, gibt es einen Doppelgänger. Demonstrieren Sie diesen Sachverhalt anhand eines Programmbeispiels.

*Lösung: Paket set2*

**Aufgabe 6**

Konvertieren Sie eine Liste in eine Menge und umgekehrt.

*Lösung: Paket convert*

**Aufgabe 7**

Schreiben Sie ein Programm,

- das herausfindet, ob eine Liste von Zahlen Duplikate hat, und
- die Anzahl gleicher Elemente ermittelt.

Nutzen Sie die Lösung von Aufgabe 6 sowie die `Map`-Methode `getOrDefault`.

*Lösung: Paket doppelt*

**Aufgabe 8**

Implementieren Sie die Mengenoperationen Vereinigung, Durchschnitt und Differenz zweier Mengen:

```
static <E> Set<E> union(Set<E> setA, Set<E> setB)
static <E> Set<E> intersection(Set<E> setA, Set<E> setB)
static <E> Set<E> difference(Set<E> setA, Set<E> setB)
```

Verwenden Sie hierzu die folgenden `Set`-Methoden:

`boolean retainAll(Collection<?> c)`

entfernt alle Elemente dieser Menge, die nicht in `c` enthalten sind.

`boolean removeAll(Collection<?> c)`

entfernt alle Elemente dieser Menge, die in `c` enthalten sind.

*Lösung: Paket set\_op*

**Aufgabe 9**

Die Einträge in einer `Map` sollen auf sechs verschiedene Arten durchlaufen werden. Nutzen Sie die `foreach`-Schleife, die `for`-Zählschleife und das `Iterator`-Interface jeweils mit `map.keySet()` und `map.entrySet()`.

*Lösung: Paket map1*

**Aufgabe 10**

Erstellen Sie eine `Map`, deren Einträge aus Wortpaaren bestehen. Ein Wortpaar besteht aus einem deutschen Wort (Schlüssel) und der zugehörigen englischen Übersetzung (Wert).

Implementieren Sie jeweils eine Methode, die ein Wortpaar einfügt, den Inhalt der Map als Zeichenkette ausgibt, zu einem deutschen Wort die englische Übersetzung ausgibt, alle deutschen Wörter (sortiert) ausgibt, alle englischen Wörter ausgibt

sowie alle englischen Wörter in aufsteigender Reihenfolge ausgibt (siehe Aufgabe 1).

*Lösung: Paket dictionary*

### Aufgabe 11

Instanzen der Klasse User mit dem Attribut int id sollen als Schlüssel in einer HashMap gespeichert werden. Die Methode equals soll implementiert werden, nicht aber hashCode.

Betrachten Sie den folgenden Codeausschnitt:

```
Map<User, String> map = new HashMap<>();
User user1 = new User(4711);
map.put(user1, "Hugo Meier");
User user2 = new User(4711);
System.out.println(map.get(user2));
```

map.get(user2) gibt null zurück und nicht den String Hugo Meier.

Wir können Sie sich das erklären?

*Lösung: Paket map2*

### Aufgabe 12

Implementieren Sie das folgende Interface MyList mit Hilfe eines Arrays.

```
public interface MyList<E> extends Iterable<E> {
    // Neues Element an erster Stelle einfügen
    void addFirst(E obj);

    // Neues Element an letzter Stelle einfügen
    void addLast(E obj);

    // Prüft, ob ein Element in der Liste vorhanden ist
    boolean contains(E obj);

    // Prüft, ob die Liste leer ist
    boolean isEmpty();

    // Liefert die Anzahl Elemente in der Liste
    int size();

    // Liefert das erste Element der Liste
    E getFirst();

    // Liefert das letzte Element der Liste
    E getLast();

    // Liefert das Element an der Position index
    E get(int index);

    // Entfernt ein Element aus der Liste
    boolean remove(E obj);
}
```

Die Implementierung `MyArrayList` soll drei Attribute enthalten:

- Die Länge des Arrays:

```
private static final int BLOCK_SIZE = 20;
```

- Das Array, das die Daten enthält:

```
private Object[] data = new Object[BLOCK_SIZE];
```

- Die nächste freie Position, um ein neues Element zu speichern:

```
private int freePosition = 0;
```

Wenn das Array voll ist und ein neues Element hinzugefügt werden soll, muss ein neues größeres Array erzeugt und das alte Array in das neue kopiert werden. Hierzu ist bei `addFirst` und `addLast` die folgende Hilfsmethode aufzurufen:

```
private void expandIfNeeded() {
    if (freePosition >= data.length) {
        Object[] newData = new Object[data.length + BLOCK_SIZE];
        System.arraycopy(data, 0, newData, 0, size());
        data = newData;
    }
}
```

Beim Entfernen eines Elements aus der Mitte mit `remove` muss die entstehende Lücke geschlossen werden. Alle nachfolgenden Elemente müssen nach links um eine Position aufrücken. Das kann wieder mit Hilfe von `System.arraycopy(...)` realisiert werden. Da das Interface `MyList` das Interface `Iterable` erweitert, muss auch

```
public Iterator<E> iterator()
```

implementiert werden. So können dann Listen vom Typ `MyList` in einer `foreach`-Schleife durchlaufen werden.

*Lösung: Paket list*



## 21 Lambda-Ausdrücke

Bis zur Java-Version 7 gab es keine Möglichkeit, direkt eine Methode als Argument einer anderen Methode beim Aufruf zu übergeben und damit diese mit einer bestimmten Funktionalität auszustatten.

Um Letzteres zu erreichen, kann eine Instanz einer anonymen Klasse definiert werden, die die in Frage stehende Methode enthält. Diese Instanz wird dann als Argument beim Aufruf übergeben. Das nachfolgende Programm demonstriert diese Vorgehensweise.

Ab Java-Version 8 können sogenannte *Lambda-Ausdrücke* anonyme Klassen in vielen Fällen ersetzen. Der Begriff *Lambda-Ausdruck* stammt aus dem Lambda-Kalkül, einer formalen Sprache zur Definition und Anwendung von Funktionen.<sup>1</sup>

Da Lambda-Ausdrücke in Java mit speziellen Interfaces (sogenannten Funktionsinterfaces) zusammenhängen, werden diese zunächst erläutert.

### Lernziele

In diesem Kapitel lernen Sie

- was Lambda-Ausdrücke in Java sind,
- wie Funktionsinterfaces mittels Lambda-Ausdrücken und Methodenreferenzen implementiert werden können und
- wie Lambda-Ausdrücke die Programmierung vereinfachen können.

Im folgenden Beispiel wird die Methode `calculate` an die Methode `someMethod` über den Umweg "anonyme Klasse" übergeben.

```
package anonym;
public interface X {
    int calculate(int a, int b);
}

package anonym;

public class OldFashioned {
    private final int i;

    public OldFashioned(int i) {
        this.i = i;
    }
}
```

---

1 Der Lambda-Kalkül wurde in den 1930er Jahren von den US-amerikanischen Mathematikern Alonzo Church und Stephen Cole Kleene eingeführt.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_21](https://doi.org/10.1007/978-3-658-43574-5_21).

```

public void someMethod(X x) {
    int result = i + x.calculate(2, 3);
    System.out.println(result);
}

public static void main(String[] args) {
    OldFashioned old = new OldFashioned(1);

    old.someMethod(new X() {
        public int calculate(int a, int b) {
            return a + b;
        }
    });
}
}

```

Im `main`-Programm wird der Methode `someMethod` ein Objekt einer anonymen Klasse, die das Interface `X` implementiert, übergeben.

## 21.1 Funktionsinterfaces

Ein *Funktionsinterface (funktionale Schnittstelle)* ist ein Interface mit *genau einer* abstrakten Methode. Default-Methoden, private Methoden und statische Methoden dürfen zusätzlich vorhanden sein.

Bei der Feststellung, ob das Interface genau eine abstrakte Methode enthält, spielt ein abstrakte Methode, deren Methodenkopf mit einer `public Object`-Methode wie z. B. `equals` übereinstimmt, keine Rolle.

Ist das Interface mit der Annotation `@FunctionalInterface` versehen, prüft der Compiler, ob es sich um ein Funktionsinterface handelt.

Beispiel:

```

@FunctionalInterface
public interface X {
    void m();

    boolean equals(Object obj);

    default void a() {
        System.out.println(hallo());
    }

    private String hallo() {
        return "Hallo";
    }

    static void b() {
        System.out.println("Welt");
    }
}

```

Hier handelt es sich um ein Funktionsinterface, da die Methode `m` die einzige abstrakte Methode ist und `equals` dabei keine Rolle spielt.

Bekannte Interfaces wie `java.lang.Iterable`, `java.lang.Runnable` und `java.awt.event.ActionListener` sind Funktionsinterfaces.

Das Paket `java.util.function` enthält eine Reihe weiterer Funktionsinterfaces. Angegeben ist im Folgenden jeweils die "funktionale" Methode.

`Predicate<T>`  
    `boolean test(T t)`  
    prüft, ob eine Bedingung für `t` erfüllt ist.

`Supplier<T>`  
    `T get()`  
    liefert ein Ergebnis.

`Consumer<T>`  
    `void accept(T t)`  
    verarbeitet das Objekt `t`.

`Function<T, R>`  
    `R apply(T t)`  
    bildet das Argument auf einen anderen Wert ab.

`BiFunction<T, U, R>`  
    `R apply(T t, U u)`  
    erzeugt aus den Argumenten `t` und `u` ein Ergebnis.

`UnaryOperator<T>`  
    `T apply(T t)`  
    bildet `t` auf ein `T`-Objekt ab.

`BinaryOperator<T>`  
    `T apply(T t1, T t2)`  
    verknüpft zwei Argumente zu einem Wert.

`Predicate` kann zum Filtern von Elementen einer Liste verwendet werden, wie das folgende Programm zeigt.

```
package filter;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FilterTest {
    public static <T> List<T> filter(List<T> list, Predicate<T> predicate) {
        List<T> result = new ArrayList<>();
```

```

        for (T item : list) {
            if (predicate.test(item)) {
                result.add(item);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i = 1; i < 10; i++) {
            list.add(i);
        }

        List<Integer> result = filter(list, new Predicate<Integer>() {
            public boolean test(Integer n) {
                return n % 2 == 0;
            }
        });
        System.out.println(result);
    }
}

```

Der zweite Aufrufparameter der Methode `filter` ist ein Objekt einer anonymen Klasse vom Typ `Predicate`. Es werden nur die geraden Zahlen in die Ergebnisliste übernommen.

Ausgabe des Programms:

[2, 4, 6, 8]

## 21.2 Lambdas

Ein *Lambda-Ausdruck* (kurz *Lambda*) repräsentiert eine "anonyme Funktion". Er besteht aus einer Parameterliste, dem Operator `->` und einem Methodenrumpf:

`(parameter) -> { body }`

Beispiele:

```

(int x) -> x * x
(x) -> x * x
x -> x * x
(x, y) -> x + y
(x, y) -> { return x + y; }
(String s1, String s2) -> System.out.println(s1.length() + s2.length())
() -> 4711

```

## Syntax-Regeln

Für die Bildung von Lambda-Ausdrücken gelten die folgenden Regeln:

- Ein Lambda-Ausdruck kann keinen, einen oder mehrere Parameter haben.
- Die Parametertypen können explizit angegeben sein oder sie werden aus dem Kontext ermittelt.
- var kann für Parameter angegeben werden.
- Mehrere Parameter werden durch Kommas getrennt und sind geklammert.
- Bei nur einem Parameter dürfen die Klammern fehlen.
- Der Rumpf des Lambda-Ausdrucks kann keine, eine oder mehrere Anweisungen enthalten.
- Bei nur einer Anweisung dürfen die geschweiften Klammern fehlen.
- Enthält der Rumpf nur eine einzige return-Anweisung, so kann er durch den Ausdruck alleine ersetzt werden.

## Funktionsinterface mit Lambda implementieren

Funktionsinterfaces können mit Hilfe von Lambda-Ausdrücken implementiert werden. Ein Lambda-Ausdruck ist zuweisungskompatibel zu jedem Funktionsinterface, dessen "funktionale" Methode die passende Parameterliste und den passenden Rückgabetyp hat.

Beispiel:

Das Funktionsinterface `java.util.Comparator<T>` definiert die abstrakte Methode

```
int compare(T obj1, T obj2)
```

Sie vergleicht obj1 und obj2 und liefert eine negative Zahl, 0 oder eine positive Zahl, je nachdem obj1 kleiner, gleich oder größer als obj2 ist.

Der Lambda-Ausdruck

```
(x, y) -> x - y
```

ist zuweisungskompatibel zu `Comparator<Integer>`, denn zwei Integer-Zahlen als Parameter liefern eine Integer-Zahl als Ergebnis:

```
Comparator<Integer> comparator = (x, y) -> x - y;
```

Weitere Beispiele:

```
Runnable r = () -> System.out.println("Hallo");
Predicate<Integer> p = n -> n % 2 == 0;
Function<Integer, Integer> f = x -> x * x;
BinaryOperator<Integer> op = (x, y) -> x + y;
```

```
Consumer<String> c = s -> System.out.println(s);
Supplier<Integer> s = () -> 4711;
```

Lambda-Ausdrücke werden gebraucht, um anonyme Klassen zu ersetzen. Sie können immer dort angegeben werden, wo ein Objekt einer funktionalen Schnittstelle benötigt wird.

Hier die Lambda-Version des letzten Programmbeispiels:

```
package lambdas;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FilterTest {
    public static <T> List<T> filter(List<T> list, Predicate<T> predicate) {
        List<T> result = new ArrayList<>();
        for (T item : list) {
            if (predicate.test(item)) {
                result.add(item);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i = 1; i < 10; i++) {
            list.add(i);
        }

        List<Integer> result = filter(list, n -> n % 2 == 0);
        System.out.println(result);
    }
}
```

Ein Lambda-Ausdruck kann in verschiedenen Kontexten genutzt werden. Die Parametertypen werden aus dem Zuweisungskontext ermittelt (*Typ-Inferenz*).

```
package lambdas;

import java.util.function.BinaryOperator;

public class FoldTest {
    public static <T> T fold(BinaryOperator<T> op, T first, T... items) {
        T result = first;
        for (T item : items) {
            result = op.apply(result, item);
        }
        return result;
    }
}
```

```
public static void main(String[] args) {
    String s = fold((s1, s2) -> s1 + s2, "", "a", "b", "c");
    System.out.println(s);

    int x = fold((s1, s2) -> s1 + s2, 0, 1, 2, 3);
    System.out.println(x);
}
```

Hier wird der Lambda-Ausdruck `(s1, s2) -> s1 + s2` verwendet, mit dessen Hilfe Strings verkettet bzw. Zahlen aufaddiert werden.

Ausgabe des Programms:

```
abc
6
```

### Zugriff auf Variablen, this und super

Der Rumpf eines Lambda-Ausdrucks hat Zugriff auf alle Attribute (Instanz- und Klassenvariablen) der umgebenden Klasse und auf die *unveränderlichen* lokalen Variablen der Definitionsumgebung.

Unveränderlich bedeutet, dass die Variable ohne Fehlermeldung des Compilers mit `final` versehen werden könnte.

Das Schlüsselwort `this` bezieht sich auf Attribute der umgebenden Klasse, `super` auf deren Basisklasse.

```
package lambdas;

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

public class ClosureTest {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        List<Runnable> list = new ArrayList<>();

        for (int n : numbers) {
            list.add(() -> System.out.println(n));
        }

        for (Runnable r : list) {
            r.run();
        }
    }
}
```

Ausgabe des Programms:

```
1
2
3
```

## Closure

Die im Rumpf des Lambda-Ausdrucks benutzten Variablenwerte der Definitionsumgebung (hier die Werte der lokale Variablen `n`) werden "eingeschlossen" (*Closure*) und stehen dann in einer anderen Umgebung, in der die Methode (hier `run`) aufgerufen wird, wieder zur Verfügung.

## 21.3 Methodenreferenzen

Ein Funktionsinterface kann mit Hilfe eines Lambda-Ausdrucks oder einer Methodenreferenz implementiert werden.

Eine *Methodenreferenz* ist eine Referenz auf eine Klassenmethode, einen Konstruktor oder eine Instanzmethode (siehe Tabelle 21-1).

**Tabelle 21-1:** Arten von Methodenreferenzen

Referenz auf ...	Beispiel	Lambda-Äquivalent
Klassenmethode	<code>MyPredicates::isEven</code>	<code>n -&gt; MyPredicates.isEven(n)</code>
Konstruktor	<code>Person::new</code>	<code>n -&gt; new Person(n)</code>
Instanzmethode (ungebunden)	<code>Person::getName</code>	<code>p -&gt; p.getName()</code>
Instanzmethode (gebunden)	<code>ConsumingTest t = new ConsumingTest("#"); t::output</code>	<code>ConsumingTest t = new ConsumingTest("#"); s -&gt; t.output(s)</code>

In den folgenden Programmen werden die verschiedenen Arten von Methodenreferenzen verwendet.

```
package method_reference;

public class MyPredicates {
    public static boolean isEven(int n) {
        return n % 2 == 0;
    }

    public static boolean isOdd(int n) {
        return n % 2 != 0;
    }
}

package method_reference;

import java.util.ArrayList;
import java.util.List;
```

```
import java.util.function.Predicate;

public class FilterTest {
    public static <T> List<T> filter(List<T> list, Predicate<T> predicate) {
        List<T> result = new ArrayList<>();
        for (T item : list) {
            if (predicate.test(item)) {
                result.add(item);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i = 1; i < 10; i++) {
            list.add(i);
        }

        List<Integer> result1 = filter(list, MyPredicates::isEven);
        System.out.println(result1);

        List<Integer> result2 = filter(list, MyPredicates::isOdd);
        System.out.println(result2);
    }
}
```

Der Konstruktor der Klasse Person mit dem Parametertyp String im folgenden Beispiel ist kompatibel zur abstrakten Methode des Funktionsinterface Function. Aus einem String wird mit Hilfe des Kontruktors ein Person-Objekt erzeugt.

Der zweite Teil des Beispieldes zeigt, dass auch (ungebundene) Referenzen auf Instanzmethoden genutzt werden können. Argument der apply-Methode des Funktionsinterface Function ist die Referenz auf ein Person-Objekt.

Person::getName bildet ein Person-Objekt auf einen String ab.

```
package method_reference;

public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
package method_reference;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class MappingTest {
    public static <T, R> List<R> map(List<T> list, Function<T, R> function) {
        List<R> result = new ArrayList<>();
        for (T item : list) {
            R value = function.apply(item);
            result.add(value);
        }
        return result;
    }

    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Hugo");
        names.add("Emil");
        names.add("Anton");

        List<Person> persons = map(names, Person::new);
        names = map(persons, Person::getName);
        System.out.println(names);
    }
}
```

Das nächste Programm zeigt, dass auch die Referenz auf die Instanzmethode für ein spezielles Objekt (`System.out` bzw. `t`) genutzt werden kann (gebundene Instanzmethoden-Referenz).

```
package method_reference;

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

public class ConsumingTest {
    private final String header;

    public ConsumingTest(String header) {
        this.header = header;
    }

    public void output(String s) {
        System.out.println(header + " " + s);
    }

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Hugo");
        list.add("Emil");
        list.add("Anton");
```

```
list.forEach(System.out::println);

ConsumingTest t = new ConsumingTest("#");
list.forEach(t::output);
}
```

Das Interface `java.util.List<T>` erweitert das Interface `java.util.Iterable<T>`. `Iterable` enthält die Default-Methode

```
default void forEach(Consumer<? super T> action)
```

Diese wendet die Methode `accept` des Arguments `action` für jedes Element der Reihe nach an.

Die `PrintStream`-Methode `println` ist kompatibel zur Methode `accept` des Funktionsinterface `Consumer`, gleiches gilt für die Instanzmethode `output`.

Ausgabe des Programms:

```
Hugo
Emil
Anton
# Hugo
# Emil
# Anton
```

## 21.4 Weitere Beispiele

### Eine Alternative zur Template-Methode abstrakter Klassen

Das Beispiel in Kapitel 8.5 zeigt, wie abstrakte *Template-Methoden*, die in Subklassen überschrieben werden, eingesetzt werden können, wenn bereits eine Teilimplementierung in der abstrakten Superklasse existiert.

Das folgende Beispiel zeigt eine *Lösungsalternative*, die den gleichen Effekt erzielt, indem auf Vererbung verzichtet wird. Es wird ein Konstruktor genutzt, der einen *Supplier* als Funktionsobjekt übernimmt.

```
package supplier;

import java.util.function.Supplier;

public class Zeit {
    private final Supplier<Long> minuten;

    public Zeit(Supplier<Long> minuten) {
        this.minuten = minuten;
    }
}
```

```
public long getSekunden() {
    return minuten.get() * 60;
}
}

package supplier;

public class Tage {
    private final long tage;

    public Tage(long tage) {
        this.tage = tage;
    }

    public long getMinuten() {
        return tage * 24 * 60;
    }
}

package supplier;

public class StundenMinuten {
    private final long stunden;
    private final long minuten;

    public StundenMinuten(long stunden, long minuten) {
        this.stunden = stunden;
        this.minuten = minuten;
    }

    public long getMinuten() {
        return stunden * 60 + minuten;
    }
}

package supplier;

public class Test {
    public static void main(String[] args) {
        Tage t = new Tage(3);
        Zeit z1 = new Zeit(t::getMinuten);
        System.out.println(z1.getSekunden());

        StundenMinuten sm = new StundenMinuten(8, 30);
        Zeit z2 = new Zeit(sm::getMinuten);
        System.out.println(z2.getSekunden());
    }
}
```

### BiFunction zum Füllen einer Matrix

Die folgende Klasse `Matrix<T>` repräsentiert eine zweidimensionale Matrix mit `T`-Werten.

Die Elemente der Matrix werden intern in einem Array zeilenweise aneinander gereiht gespeichert. So besteht z. B. eine  $3 \times 4$ -Matrix aus 3 Zeilen und 4 Spalten. Das Array hat demnach  $12 = 4 + 4 + 4$  Einträge.

`BiFunction<Integer, Integer, T>` wird genutzt, um die Matrix zu initialisieren. Zeilenindex und Spaltenindex erzeugen den `T`-Wert.

Die Matrix implementiert das Interface `Iterable<Matrix.Element<T>>`, sodass die Matrixelemente `Matrix.Element<T>` in einer `foreach`-Schleife bequem durchlaufen werden können.

```
package matrix;

import java.util.Iterator;
import java.util.function.BiFunction;

public class Matrix<T> implements Iterable<Matrix.Element<T>> {
    private final int rows;
    private final int cols;
    private final T[] values;

    public Matrix(int rows, int cols, BiFunction<Integer, Integer, T> filler) {
        this.rows = rows;
        this.cols = cols;
        values = (T[]) new Object[rows * cols];

        for (int x = 0; x < rows; x++) {
            for (int y = 0; y < cols; y++) {
                values[x * cols + y] = filler.apply(x, y);
            }
        }
    }

    public T get(int x, int y) {
        return values[x * cols + y];
    }

    public void set(int x, int y, T value) {
        values[x * cols + y] = value;
    }

    public static class Element<T> {
        public int x;
        public int y;
        public T value;
    }
}
```

```
public void display() {
    for (Element<T> element : this) {
        System.out.print(element.value + " ");
        if (element.y == cols - 1)
            System.out.println();
    }
}

@Override
public Iterator<Element<T>> iterator() {
    return new Iterator<T>() {
        private int x = 0;
        private int y = 0;

        @Override
        public boolean hasNext() {
            return x != rows;
        }

        @Override
        public Element<T> next() {
            Element<T> element = new Element<T>();
            element.x = x;
            element.y = y;
            element.value = get(x, y);

            y++;
            if (y == cols) {
                y = 0;
                x++;
            }
            return element;
        }
    };
}

package matrix;

import java.util.function.BiFunction;

public class Test {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, String> sFiller =
            (x, y) -> "(" + x + "," + y + ")";
        Matrix<String> sMatrix = new Matrix<String>(3, 4, sFiller);
        sMatrix.display();

        BiFunction<Integer, Integer, Double> dFiller =
            (x, y) -> Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
        Matrix<Double> dMatrix = new Matrix<Double>(3, 4, dFiller);
        dMatrix.display();
    }
}
```

Ausgabe des Programms:

```
(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
0.0 1.0 2.0 3.0
1.0 1.4142135623730951 2.23606797749979 3.1622776601683795
2.0 2.23606797749979 2.8284271247461903 3.605551275463989
```

## Comparator-Konstruktionsmethoden

In Kapitel 21.2 wurde das Funktionsinterface `java.util.Comparator` vorgestellt.

Die folgende `java.util.Collections`-Methode nutzt dieses Interface, um Objekte in einer Liste zu sortieren:

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

Comparator können mit Hilfe von geeigneten *Schlüsselextraktionsfunktionen* konstruiert werden.

Zu diesem Zweck enthält das Interface `Comparator` die beiden statischen Methoden `comparing` und `thenComparing`.

```
static <T, U extends Comparable<? super U>> Comparator<T> comparing(
    Function<? super T, ? extends U> keyExtractor)
```

$T$  ist der Typ der Objekte, die verglichen werden sollen.  $U$  ist der Typ des Sortierschlüssels.

Beispiel:

Die Klasse `Person` enthält das Attribut `int age` mit der entsprechenden `get`-Methode. Mit Hilfe des folgenden `Comparator` kann dann nach Alter sortiert werden:

```
Comparator<Person> byAge = Comparator.comparing(Person::getAge);
```

```
default <U extends Comparable<? super U>> Comparator<T> thenComparing(
    Function<? super T, ? extends U> keyExtractor)
```

Diese Instanz-Methode wendet erst den ursprünglichen `Comparator` an und verwendet dann den extrahierten Schlüssel.

Beispiel:

```
Comparator<Person> byAgeAndSalary = Comparator
    .comparing(Person::getAge)
    .thenComparing(Person::getSalary);
```

Hier wird zuerst nach Alter sortiert und dann innerhalb von Personen desselben Alters nach Gehalt.

Das Interface `java.util.List` implementiert die Default-Methode `sort`:

```
default void sort(Comparator<? super E> c)

package compare;

public class Person {
    private final String name;
    private final int age;
    private final double salary;

    public Person(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public double getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return "Person{" +
            "name='" + name + '\'' +
            ", age=" + age +
            ", salary=" + salary +
            '}';
    }
}

package compare;

import java.util.Comparator;
import java.util.List;

public class Test {
    public static void main(String[] args) {
        List<Person> list = new java.util.ArrayList<>();
        list.add(new Person("Peter", 25, 2000.));
        list.add(new Person("Emil", 30, 3000.));
        list.add(new Person("Hugo", 40, 7000.));
        list.add(new Person("Paul", 30, 2500.));
        list.add(new Person("David", 40, 6500.));

        Comparator<Person> byAge = Comparator.comparing(Person::getAge);
        list.sort(byAge);
        System.out.println(list);
    }
}
```

```

        Comparator<Person> byAgeAndSalary = Comparator
            .comparing(Person::getAge)
            .thenComparing(Person::getSalary);
        list.sort(byAgeAndSalary);
        System.out.println(list);
    }
}

```

### Ausgabe des Programms:

```

[Person{name='Peter', age=25, salary=2000.0}, Person{name='Emil', age=30,
salary=3000.0}, Person{name='Paul', age=30, salary=2500.0}, Person{name='Hugo',
age=40, salary=7000.0}, Person{name='David', age=40, salary=6500.0}]
[Person{name='Peter', age=25, salary=2000.0}, Person{name='Paul', age=30,
salary=2500.0}, Person{name='Emil', age=30, salary=3000.0}, Person{name='David',
age=40, salary=6500.0}, Person{name='Hugo', age=40, salary=7000.0}]

```

### Memoisierung

Um die wiederholte Ausführung aufwändiger Berechnungen zu beschleunigen, können Rückgabewerte von Funktionen zwischengespeichert und wiederverwendet werden (*Memoisierung*, engl. *Memoization*).

Die folgende Methode stattet eine Funktion `fn` mit einer `Map<T, R>` als Cache aus:

```
static <T, R> Function<T, R> memoize(Function<T, R> fn)
```

Die Variable `cache` wird im erzeugten Funktionsobjekt in ein *Closure* verpackt.

```

package memoization;

import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public class MemoizationTest {
    public static int fibonacci(int n) {
        if (n < 2) return 1;
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static <T, R> Function<T, R> memoize(Function<T, R> fn) {
        Map<T, R> cache = new HashMap<>();
        return t -> {
            R r = cache.get(t);
            if (r == null) {
                r = fn.apply(t);
                cache.put(t, r);
                System.out.println("Cache: " + cache.entrySet());
            }
            return r;
        };
    }
}

```

```

public static void main(String[] args) {
    Function<Integer, Integer> fibWithCache = memoize(
        MemoizationTest::fibonacci);
    System.out.println(fibWithCache.apply(3));
    System.out.println(fibWithCache.apply(5));
    System.out.println(fibWithCache.apply(3));
    System.out.println(fibWithCache.apply(5));
    System.out.println(fibWithCache.apply(8));
}
}

```

Ausgabe:

```

Cache: [3=3]
3
Cache: [3=3, 5=8]
8
3
8
Cache: [3=3, 5=8, 8=34]
34

```

## Funktionstabellen erzeugen

Das folgende Beispiel zeigt, wie Funktion vom Typ `Function<Double, Double>` erzeugt und die Funktionswerte in einer Tabelle angezeigt werden können.

Parameter der Methode `print` sind die Funktion als Lambda-Ausdruck oder Methodenreferenz, ein Start- und ein Endwert und die Anzahl von zu berechnenden Funktionswerten.

```

package functions;

import java.util.function.Function;

public class FunctionTest {
    public static void print(Function<Double, Double> f,
                           double from, double to, int n) {

        System.out.println("      x |      f(x)");
        System.out.println("----- + -----");
        double d = (to - from) / (n - 1);
        for (int i = 0; i < n; i++) {
            double x = from + d * i;
            String row = "%10.5f | %10.5f".formatted(x, f.apply(x));
            System.out.println(row);
        }
        System.out.println();
    }

    public static Function<Double, Double> genPolynom(
        double a, double b, double c) {
        return x -> a * x * x + b * x + c;
    }
}

```

```

public static void main(String[] args) {
    print(Math::sin, 0, Math.PI, 10);
    print(genPolynom(2, -1, 2), -1, 1, 11);
}
}

```

Ausgabe (für das erzeugte Polynom):

x	f(x)
-1,00000	5,00000
-0,80000	4,08000
-0,60000	3,32000
-0,40000	2,72000
-0,20000	2,28000
0,00000	2,00000
0,20000	1,88000
0,40000	1,92000
0,60000	2,12000
0,80000	2,48000
1,00000	3,00000

## 21.5 Aufgaben

### Aufgabe 1

Eine Liste enthält diverse Strings. Nutzen Sie einen Lambda-Ausdruck, um die Strings in Kleinbuchstaben auf dem Bildschirm auszugeben.

*Lösung: Paket lower*

### Aufgabe 2

Nutzen Sie eine Methodenreferenz, um eine Liste von Zahlen auf dem Bildschirm auszugeben.

*Lösung: Paket method\_ref*

### Aufgabe 3

Ein Array aus unterschiedlich langen Zeichenketten soll nach Länge der Zeichenketten sortiert werden. Nutzen Sie die `java.util.Arrays`-Methode

```
static <T> void sort(T[] a, Comparator<? super T> c)
```

sowie einen geeigneten Lambda-Ausdruck.

*Lösung: Paket sort*

### Aufgabe 4

Implementieren Sie das Funktionsinterface

```
Function<Consumer<String>, Consumer<String>>
```

mit Hilfe der geschachtelten Lambda-Ausdrücke:

```
c -> { return t -> { c.accept(t); c.accept(t); }; };
```

oder kurz

```
c -> t -> { c.accept(t); c.accept(t); };
```

Testen Sie das Funktionsinterface mit unterschiedlichen Implementierungen, z. B. Ausgabe eines Strings oder Hinzufügen eines Strings in eine Liste.

*Lösung: Paket function*

### Aufgabe 5

Die Klasse `Termin` soll die Attribute `beschreibung`, `ort` und `zeit` haben, wobei `zeit` vom Datentyp `LocalDateTime` ist. Speichern Sie mehrere Termine in einem Array und sortieren Sie anschließend nach den Zeitangaben.

Verwenden Sie zum Sortieren die Methode `Arrays.sort` (siehe Aufgabe 3). Implementieren Sie eine Methode, die alle Termine ausgibt, auf die ein bestimmtes Kriterium zutrifft (z. B. Termine in einem vorgegebenen Monat):

```
void printTermine(Termin[] termine, Predicate<Termin> kriterium)
```

Nutzen Sie einen Lambda-Ausdruck, um diese Methode aufzurufen.

*Lösung: Paket termine*

### Aufgabe 6

Implementieren Sie die statische Methode `map`, die Elemente einer Liste in Elemente einer neuen Liste transformiert:

```
static <E, R> List<R> map(List<E> list, Function<E, R> mapper)
```

*Lösung: Paket map*

### Aufgabe 7

Eine Liste enthält `Artikel`-Objekte. Erzeugen Sie eine Map, die einer Warenguppe die Anzahl der Artikel, die zu dieser Warengruppe gehören, zuordnet. Verwenden Sie hierzu die folgende Default-Methode des Interface `Map`:

```
default V merge(K key, V value,
BiFunction<? super V, ? super V, ? extends V> func)
```

Mit `key` wird `value` assoziiert, falls der Schlüssel noch nicht vorhanden ist, ansonsten wird aus `value` und dem vorhandenen Wert mit Hilfe der `BiFunction` ein neuer Wert berechnet und zugeordnet.

*Lösung: Paket merge*



## 22 Streams

Mit Java-Version 8 wurde das *Stream*-Konzept eingeführt. *Streams* erleichtern die Ausführung von Operationen auf endlichen oder unendlichen Folgen von Daten-elementen. Dabei sind auch mehrstufige Berechnungen möglich. Streams können beispielsweise für alle Klassen, die das Interface `Collection` implementieren, erzeugt werden.

### Lernziele

In diesem Kapitel lernen Sie

- wie Streams erstellt, Berechnungen durchgeführt und Ergebnisse erzeugt werden können,
- welche wichtigen Stream-Operationen existieren und
- wie Streams die Verarbeitung von Daten "am Fließband" erheblich vereinfachen.

### 22.1 Das Stream-Konzept

Das Interface `java.util.stream.Stream` repräsentiert eine Folge von Elementen, die in mehreren Schritten verarbeitet werden können:

- Transformationen der Elemente (*intermediäre Operationen*)
- Erzeugung eines Endergebnisses (*terminale Operation*)

Ein Stream speichert keine Daten, sondern transportiert Daten durch eine *Pipeline von Operationen*. Die Verarbeitung der einzelnen Schritte wird erst mit Beginn der *terminalen Operation* ausgelöst. Hier werden die Elemente von der jeweils vorhergehenden Verarbeitungsstufe aktiv geholt (*Pull-Prinzip*).

Ein Stream kann das Ergebnis *nur einmal* bereitstellen und somit nicht wieder-verwendet werden. Die zugrunde liegende Datenquelle wird nicht verändert.

Bei Verwendung von Streams wird nur beschrieben, *was zu tun ist*, nicht aber *wie*.



**Abbildung 22-1:** Bearbeitung am Fließband

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_22](https://doi.org/10.1007/978-3-658-43574-5_22).

## 22.2 Erzeugung von Streams

Die `java.util.Collection`-Methode

```
default Stream<T> stream()
```

erzeugt einen Stream für die Elemente in der Collection.

Einige `Stream`-Methoden:

```
static <T> Stream<T> of(T... values)
```

liefert einen Stream aus den Elementen `values`.

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

erzeugt aus einem Startwert `seed` und durch wiederholte Anwendung von `f` einen unendlichen Stream.

```
static <T> Stream<T> generate(Supplier<? extends T> s)
```

erzeugt einen unendlichen Stream durch wiederholte Anwendung von `s`.

Für einfache Datentypen gibt es die spezifischen Interfaces `IntStream`, `LongStream` und `DoubleStream`.

Das folgende Beispiel demonstriert die Erzeugung von Streams.

```
package create;

import java.util.List;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class CreateStreams {
    public static void main(String[] args) {
        List<Integer> list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
        list.stream().limit(5).forEach(System.out::println);

        Integer[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        Stream.of(numbers).limit(5).forEach(System.out::println);

        Stream.iterate(1, n -> n + 1).limit(5).forEach(System.out::println);
        IntStream.iterate(1, n -> n + 1).limit(5).forEach(System.out::println);

        Stream.generate(Math::random).limit(5).forEach(System.out::println);
    }
}
```

Die *intermediäre Operation* `limit` begrenzt den Stream auf eine bestimmte Anzahl. Die *terminale Operation* `forEach` führt eine Aktion für jedes Element des Streams aus.

## 22.3 Transformation und Auswertung

Es existieren zahlreiche intermediäre und terminale Operationen. In den Beispielen dieses Abschnitts werden die folgenden Stream-Methoden demonstriert.

### Intermediäre Operationen

```
Stream<T> filter(Predicate<? super T> predicate)
<R> Stream<R> map(Function<? super T,>? extends R> mapper)
<R> Stream<R> flatMap(Function<? super T,
    ? extends Stream<? extends R>> mapper)
Stream<T> peek(Consumer<? super T> action)
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
Stream<T> takeWhile(Predicate<? super T> predicate)
Stream<T> dropWhile(Predicate<? super T> predicate)
```

`filter` filtert einzelne Elemente anhand eines `Predicate`-Objekts.

`map` wendet die angegebene Operation (`Function`-Objekt) auf jedes einzelne Element an.

`flatMap` liefert einen einzigen Stream, der sich aus den Elementen der von `mapper` erzeugten einzelnen Streams zusammensetzt.

`peek` führt für jedes Element den bereitgestellten `Consumer` aus.

`sorted` erwartet einen `Comparator` als Argument und sortiert die Elemente des Streams. Ist kein Argument angegeben, wird gemäß der natürlichen Ordnung sortiert.

`limit` begrenzt den Stream auf `maxSize` Elemente.

`skip` liefert einen Stream ohne die ersten `n` Elemente.

`takeWhile` verarbeitet Elemente, solange die Bedingung erfüllt ist.

`dropWhile` überspringt Elemente, solange die Bedingung erfüllt ist.

### Terminale Operationen

```
void forEach(Consumer<? super T> action)
List<T> toList()
long count()
T reduce(T identity, BinaryOperator<T> accumulator)
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

forEach führt für jedes Element den Consumer aus.

toList überführt den Stream in eine Liste (seit Java 16).

count liefert die Anzahl Elemente im Stream.

reduce wendet den BinaryOperator assoziativ auf die Elemente an, wobei identity der Startwert ist.

anyMatch, allMatch und noneMatch prüfen, ob die Bedingung für mindestens ein Element, für alle Elemente bzw. für kein Element erfüllt ist.

```
package operations;

public class Artikel {
    private int id;
    private char typ;
    private double preis;

    public Artikel(int id, char typ, double preis) {
        this.id = id;
        this.typ = typ;
        this.preis = preis;
    }

    public int getId() {
        return id;
    }

    // getter/setter

    // toString
}

package operations;

import java.util.Comparator;
import java.util.List;

public class StreamTest1 {
    public static void main(String[] args) {
        List<Artikel> list = List.of(
            new Artikel(4712, 'A', 30.),
            new Artikel(4714, 'A', 20.),
            new Artikel(4713, 'B', 10.),
            new Artikel(4715, 'B', 10.),
            new Artikel(4711, 'A', 10.)
        );
        // Sortieren und ausgeben
        list.stream().sorted(Comparator.comparing(Artikel::getId))
            .forEach(System.out::println);

        // Filtern und zählen
        long count = list.stream().filter(a -> a.getTyp() == 'A').count();
        System.out.println("Anzahl Artikel vom Typ 'A': " + count);
    }
}
```

```
// Filtern, ändern und Ergebnis als Liste erzeugen
List<Artikel> filtered = list.stream().filter(a -> a.getTyp() == 'A')
    .peek(a -> a.setPreis(1.1 * a.getPreis())).toList();
System.out.println(filtered);

// Map und reduce
double z = filtered.stream().map(Artikel::getPreis)
    .reduce(0., (x, y) -> x + y) / filtered.size();
System.out.println("Durchschnittspreis für Artikel vom Typ A: " + z);
}

}
```

Ausgabe des Programms:

```
4711 A 10.0
4712 A 30.0
4713 B 10.0
4714 A 20.0
4715 B 10.0
Anzahl Artikel vom Typ 'A': 3
[4712 A 33.0, 4714 A 22.0, 4711 A 11.0]
Durchschnittspreis für Artikel vom Typ A: 22.0
```

```
package operations;

import java.util.stream.IntStream;

public class StreamTest2 {
    public static void main(String[] args) {
        IntStream stream1 = IntStream.of(1, 2, 3, 4, 5, 6, 1, 2, 3);
        IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 1, 2, 3);
        IntStream stream3 = IntStream.of(1, 2, 3, 4, 5, 6, 1, 2, 3);
        stream1.takeWhile(n -> n < 4)
            .forEach(n -> System.out.print(n + " "));
        System.out.println();
        stream2.dropWhile(n -> n < 4)
            .forEach(n -> System.out.print(n + " "));
        System.out.println();
        stream3.dropWhile(n -> n < 4).takeWhile(n -> n >= 4)
            .forEach(n -> System.out.print(n + " "));
        System.out.println();

        IntStream stream4 = IntStream.of(1, 3, 4, 5, 6);
        stream4.skip(3).forEach(n -> System.out.print(n + " "));
        System.out.println();

        IntStream stream5 = IntStream.of(1, 3, 4, 5, 6);
        System.out.println(stream5.anyMatch(n -> n == 4));

        IntStream stream6 = IntStream.of(1, 3, 4, 5, 6);
        System.out.println(stream6.allMatch(n -> n <= 4));

        IntStream stream7 = IntStream.of(1, 3, 4, 5, 6);
        System.out.println(stream7.noneMatch(n -> n > 6));
    }
}
```

Ausgabe des Programms:

```
1 2 3  
4 5 6 1 2 3  
4 5 6  
5 6  
true  
false  
true
```

Das nächste Programm demonstriert die Wirkungsweise der intermediären Operation `flatMap`. Ein geschachtelter Stream (Stream von Streams) wird zu einem einfachen Stream transformiert.

```
package operations;  
  
import java.util.List;  
import java.util.stream.Stream;  
  
public class StreamTest3 {  
    public static void main(String[] args) {  
        List<Integer> list = List.of(1, 2);  
  
        Stream<Stream<Integer>> stream1 = list.stream()  
            .map(n -> Stream.of(n, n + 1));  
        stream1.forEach(s -> System.out.println(s.toList()));  
  
        Stream<Integer> stream2 = list.stream().flatMap(n -> Stream.of(n, n + 1));  
        stream2.forEach(System.out::println);  
    }  
}
```

Ausgabe des Programms:

```
[1, 2]  
[2, 3]  
1  
2  
2  
3
```

In Verbindung mit Lambda-Ausdrücken ist das Stream-API ein sehr wirkungsvolles Werkzeug für den Entwickler, das nicht zuletzt auch den Quellcode besser lesbar macht.

## 22.4 Aufgaben

### Aufgabe 1

Eine Reihe von Guthaben soll verzinst werden. Ermitteln Sie die Guthaben nach Verzinsung und geben Sie diese der Reihe nach aus. Lösen Sie die Aufgabe mit Hilfe der Stream-Methoden `map` und `forEach`.

*Lösung: Paket zinsen1*

### Aufgabe 2

Modifizieren Sie die Lösung zu Aufgabe 1, um als Ergebnis die Summe der verzinsten Guthaben zu erhalten. Lösen Sie die Aufgabe mit Hilfe der Stream-Methoden `map` und `reduce`.

*Lösung: Paket zinsen2*

### Aufgabe 3

Mit Hilfe von Stream-Methoden sollen als Kommandozeilen-Parameter erfasste Zahlen nach numerischer Größe sortiert und am Bildschirm ausgegeben werden. Nutzen Sie die `java.util.Arrays`-Methode

```
static <T> Stream<T> stream(T[] array)
```

*Lösung: Paket sort*

### Aufgabe 4

Gegeben sei eine Liste von Person-Objekten. Mit Hilfe von Stream-Methoden sollen die Namen dieser Personen in Großbuchstaben gewandelt und dann sortiert in einer Liste ausgegeben werden.

*Lösung: Paket upper*

### Aufgabe 5

Mit Hilfe von Stream-Methoden soll zu einer vorgegeben Zahl  $n$  die Fakultät  $n! = 1 * 2 * \dots * n$  ermittelt werden. Gehen Sie wie folgt vor:

Erzeugen Sie mit `Stream.iterate(1, i -> i + 1)` die Folge der Zahlen 1, 2, 3, ...

Begrenzen Sie mit `limit(n)` den Stream auf  $n$  Elemente.

Bilden Sie mit `map(i -> BigInteger.valueOf(i))` die Zahl  $i$  auf den entsprechenden `BigInteger`-Wert ab.

Nutzen Sie schließlich `reduce`, um die `BigInteger`-Werte der Reihe nach zu multiplizieren.

*Lösung: Paket fak*

**Aufgabe 6**

Ein Artikel enthält die Artikelnummer, den Preis, die Warengruppe und den Lagerbestand. Erzeugen Sie eine Liste mehrerer Artikel-Objekte und geben Sie alle Artikel, die zu einer bestimmten Warengruppe gehören und deren Lagerbestand eine bestimmte Menge übersteigt, am Bildschirm aus. Nutzen Sie hierzu mehrfach die Stream-Methode `filter`.

*Lösung: Paket filter*

**Aufgabe 7**

Berechnen Sie mit Hilfe der Stream-Methode `reduce` den Wert einer Dezimalzahl aus einer Liste von Ziffern.

*Lösung: Paket reduce1*

**Aufgabe 8**

Erzeugen Sie eine Liste aus Person-Objekten. Eine Person hat einen Namen und ein Geburtsdatum. Nutzen Sie die Stream-Methoden `filter`, `map` und `reduce`, um die Namen aller Personen, die in einem bestimmten Monat geboren sind, in einer Zeile durch jeweils ein Komma getrennt auszugeben.

Implementieren Sie für `reduce` das Interface `BinaryOperator<String>` so, dass zwei Strings kombiniert werden.

*Lösung: Paket reduce2*

**Aufgabe 9**

Die Klasse `Bankkonto` soll die Attribute `inhaber` und `guthaben` besitzen. Legen Sie mehrere Konten an und speichern Sie diese in einer `TreeMap`. Als Zugriffsschlüssel soll die Kontonummer dienen.

- Zeigen Sie für alle Konten Kontonummer und Guthaben an.
- Berechnen Sie das durchschnittliche Guthaben über alle Konten.
- Ermitteln Sie alle Kontoinhaber, die über mehr als 500 Euro Guthaben verfügen.

Verwenden Sie Streams und Lambda-Ausdrücke.

Beachte: `Map` enthält die Methode

```
default void forEach(BiConsumer<? super K, ? super V> action)
```

zur Verarbeitung aller Map-Einträge (Schlüssel, Wert).

*Lösung: Paket bank*

**Aufgabe 10**

Eine Folge von endlich vielen ganzen Zahlen soll sortiert und doppelte Einträge sollen entfernt werden. Nutzen Sie hierzu die Methode `sorted` sowie

```
Stream<T> distinct()
```

Zeigen Sie das Ergebnis am Bildschirm an.

*Lösung: Paket doppelt*

### Aufgabe 11

Erzeugen Sie mit Hilfe der Stream-Methoden iterate und map einen Stream von Quadratzahlen. Diese sollen in einer Schleife schubweise ausgegeben werden: pro Schleifendurchgang 10 Zahlen, dann die nächsten 10 usw. Nutzen Sie die Stream-Methoden

```
Stream<T> skip(long n) und Stream<T> limit(long maxSize)
```

*Lösung: Paket paging*

### Aufgabe 12

Das Interface `java.util.IntStream` repräsentiert einen Strom aus ganzen Zahlen und besitzt u. a. die *terminale Operation*

```
int sum()
```

Berechnen Sie die Summe von 1 bis 100.

*Lösung: Paket sum*

### Aufgabe 13

Teilen Sie eine Liste von Zahlen in gerade und ungerade Zahlen auf.

Beispiel:

```
List<Integer> list = List.of(2, 13, 4, 25, 8, 27);
```

Ergebnis:

```
{ungerade=[13, 25, 27], gerade=[2, 4, 8]}
```

Verwenden Sie hierzu die `java.util.stream.Collectors`-Methode `groupingBy` in Kombination mit der Stream-Methode `collect`:

```
Map<String, List<Integer>> grouped = list.stream()
    .collect(Collectors.groupingBy(
        n -> n % 2 == 0 ? "gerade" : "ungerade"));
```

*Lösung: Paket grouping1*

### Aufgabe 14

Erzeugen Sie eine Liste von Person-Objekten. Jede Person hat einen Namen und einen Wohnort. Die Personen in der Liste sollen nach Wohnorten gruppiert werden.

Hierzu kann die `java.util.stream.Collectors`-Methode `groupingBy` in Kombination mit der `Stream`-Methode `collect` verwendet werden:

```
Map<String, List<Person>> grouped = persons.stream()
    .collect(Collectors.groupingBy(Person::getOrt));
```

Das Ergebnis ist eine Map, die als Schlüssel den Ort und als Wert die Liste der dort wohnenden Personen enthält.

Geben Sie auf dem Bildschirm die Anzahl Personen pro Ort sowie alle Namen der Personen eines jeden Orts aus.

*Lösung: Paket grouping2*

### Aufgabe 15

Ein aus 100 Wörtern bestehender Text soll analysiert werden. Den Text können Sie sich auf der Website <https://www.loremipsum.de> mit dem  *Lorem-Ipsum-Generator* erzeugen lassen. Dieser Text besteht aus pseudo-lateinischen Wörtern und hat keine Bedeutung.

Führen Sie zunächst eine Bereinigung des Textes durch: Umwandlung in Kleinbuchstaben, Entfernung der Zeichen ".", "," und "\n".

Wandeln Sie dann den bereinigten Text in eine Liste mit der `String`-Methode `split` um (Trennzeichen: " ").

Führen Sie eine Gruppierung nach Wortlänge durch. `groupingBy` (siehe die vorherigen Aufgaben) erzeugt eine Map vom Typ `Map<Integer, List<String>>`.

Erzeugen Sie aus dieser Map eine neue Map, die nach Schlüsseln aufsteigend sortiert ist und zu jedem Schlüssel (Wortlänge) die Anzahl der unterschiedlichen Wörter dieser Länge enthält.

*Lösung: Paket grouping3*

### Aufgabe 16

Ein String, der aus mehreren einzelnen mit \n getrennten Zeichenketten besteht, soll zeilenweise ausgegeben werden. Nutzen Sie hierzu die `String`-Methode

```
Stream<String> lines()
```

*Lösung: Paket lines*

### Aufgabe 17

Die Klasse `Developer` enthält als Attribute den Namen des Entwicklers und die Programmiersprachen, die er beherrscht, als Menge von Strings. Erzeugen Sie ein Team von Entwicklern als Liste und geben Sie mit Hilfe von `Stream`-Operationen die beherrschten Programmiersprachen am Bildschirm aus.

Verwenden Sie hierzu die Stream-Methode `map`, die den Stream aus Developer-Objekten in einen Stream aus Objekten vom Typ `Set<String>` transformiert. Wenden Sie dann hierauf die Stream-Methode `flatMap` an, die jedes Element des Streams in einen Stream aus Strings abbildet und alle diese neuen Streams zu einem einzigen Stream verbindet.

Die Ausgabe soll sortiert sein und keine doppelten Zeichenketten enthalten (siehe Aufgabe 10).

*Lösung: Paket flatten*

### Aufgabe 18

Jeder Stream kann für die Parallelverarbeitung genutzt werden.

Mit `parallelStream` kann aus einer `Collection` ein paralleler Stream erzeugt werden. Mit der Methode `parallel` kann ein (sequentieller) Stream in einen parallelen Stream umgewandelt werden.

Hierzu wird der Stream in genügend kleine Teile zerlegt. Diese werden dann in mehreren Threads (siehe Kapitel 26) sequentiell verarbeitet. Zum Schluss werden die Teilergebnisse zu einem Gesamtergebnis kombiniert.

Hinweise:

- Nur bei großen Datenmengen und rechenintensiven Aufgaben sollten parallele Streams genutzt werden.
- Gemeinsam benutzte Ressourcen sollten wegen möglicher Konflikte beim parallelen Zugriff vermieden werden.
- Die Datenquelle sollte sich gut für die Aufteilung in parallel auszuführende Teile eignen.

Das Programm zu dieser Aufgabe zeigt die Rechenzeit in Millisekunden für die sequentielle und die parallele Verarbeitung einer größeren Berechnung an.

Studieren Sie dieses Programm und überzeugen Sie sich davon, dass in diesem Beispiel die parallele Verarbeitung Vorteile hat.

*Lösung: Paket parallel*



## 23 Optionale Werte

Die Behandlung von `null`-Werten kann mitunter lästig sein. Ein allzu leichtsinniger Umgang hiermit führt aber oft zu unangenehmen Laufzeitfehlern.

Gibt eine Methode `null` zurück, so ist nicht immer klar, was hiermit gemeint ist. Bei einer Suchfunktion könnte `null` "nicht gefunden" bedeuten oder aber auch, dass ein Fehler während der Suche aufgetreten ist. Eine gute Dokumentation der Methode ist also unbedingt erforderlich.

### Lernziele

In diesem Kapitel lernen Sie

- warum `null` als Rückgabewert von Methoden problematisch sein kann,
- wie die Klasse `Optional` zur besseren Behandlung optionaler Werte verwendet werden kann und
- wie damit Ausnahmen vom Typ `NullPointerException` reduziert werden können.

### 23.1 Motivation

Das folgende Programm setzt `null` beim Suchen ein, um zu signalisieren, dass etwas nicht gefunden wurde. Es zeigt auch, dass ein leichtfertiger Umgang mit solchen Suchmethoden die Stabilität zur Laufzeit gefährdet.

```
package artikel;

public record Artikel(int id, double preis) { }

package motivation;

import artikel.Artikel;
import java.util.List;

public class Test {
    public static Artikel findById(int id, List<Artikel> artikelListe) {
        for (Artikel artikel : artikelListe) {
            if (artikel.id() == id)
                return artikel;
        }
        return null;
    }

    public static void main(String[] args) {
        List<Artikel> artikelListe = List.of(
            new Artikel(4711, 10.),
```

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_23](https://doi.org/10.1007/978-3-658-43574-5_23).

```

        new Artikel(4712, 20.),
        new Artikel(4713, 30.));

    Artikel artikel = findById(4712, artikelListe);
    System.out.println(artikel.preis());

    artikel = findById(5000, artikelListe);
    System.out.println(artikel.preis());
}
}

```

Ausgabe des Programms:

```

20.0
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"artikel.Artikel.preis()" because "artikel" is null
at motivation.Test.main(Test.java:26)

```

Hier muss also darauf geachtet werden, dass der Rückgabewert auf `null` geprüft wird:

```

if (artikel != null)
    System.out.println(artikel.preis());

```

## 23.2 Die Klasse Optional

Seit Java-Version 8 gibt es die Klasse `Optional<T>`, mit der optionale Werte ausgedrückt werden. Eine Instanz dieser Klasse enthält einen Wert vom Typ `T` oder eben keinen.

Das folgende Programm zeigt eine Alternative zum vorherigen Programm, die `Optional` und `Stream`-Methoden verwendet.

```

package optional;

import artikel.Artikel;

import java.util.List;
import java.util.Optional;

public class Test1 {
    public static Optional<Artikel> findById(int id, List<Artikel> artikelListe) {
        return artikelListe
            .stream()
            .filter(a -> a.id() == id)
            .findFirst();
    }

    public static void main(String[] args) {
        List<Artikel> artikelListe = List.of(

```

```
    new Artikel(4711, 10.),
    new Artikel(4712, 20.),
    new Artikel(4713, 30.));

    int id = 4712;
    Optional<Artikel> optionalArtikel = findById(id, artikelListe);
    if (optionalArtikel.isPresent())
        System.out.println(optionalArtikel.get());
    else
        System.out.println("Artikel " + id + " nicht gefunden.");
}
}
```

Die Stream-Methode

```
Optional<T> findFirst()
```

liefert das erste Element, falls ein solches existiert.

Einige Optional-Methoden:

```
boolean isPresent()
```

liefert true, wenn ein Wert vorhanden ist, sonst false.

```
boolean isEmpty()
```

liefert true, wenn ein Wert nicht vorhanden ist, sonst false.

```
T get()
```

gibt den Wert zurück, falls dieser vorhanden ist, ansonsten wird eine Ausnahme vom Typ NoSuchElementException ausgelöst.

```
static <T> Optional<T> of(T value)
```

liefert eine Optional-Instanz für einen vorhandenen Wert value (ungleich null). Falls der Wert null ist wird eine NullPointerException ausgelöst.

```
static <T> Optional<T> ofNullable(T value)
```

liefert eine Optional-Instanz mit vorhandenem Wert value oder aber nicht vorhandenem Wert (wenn value == null).

```
static <T> Optional<T> empty()
```

liefert eine Optional-Instanz, die einen nicht vorhandenen Wert beschreibt.

Das folgende Programm zeigt, dass die Methode `findById` aus Kapitel 23.1 wieder verwendet werden kann, um eine Optional-Instanz zurückzugeben.

```
package optional;

import artikel.Artikel;
import motivation.Test;

import java.util.List;
import java.util.Optional;
```

```
public class Test2 {
    public static Optional<Artikel> findById(int id, List<Artikel> artikelListe) {
        return Optional.ofNullable(Test.findById(id, artikelListe));
    }

    public static void main(String[] args) {
        List<Artikel> artikelListe = List.of(
            new Artikel(4711, 10.),
            new Artikel(4712, 20.),
            new Artikel(4713, 30.));

        int id = 4712;
        Optional<Artikel> optionalArtikel = findById(id, artikelListe);
        if (optionalArtikel.isPresent())
            System.out.println(optionalArtikel.get());
        else
            System.out.println("Artikel " + id + " nicht gefunden.");
    }
}
```

Weitere Optional-Methoden:

`void ifPresent(Consumer<? super T> action)`

führt die Aktion `action` mit dem vorhandenen Wert aus.

`void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`

führt die Aktion `action` mit dem vorhandenen Wert aus oder `emptyAction`, wenn der Wert nicht vorhanden ist.

```
package optional;

import artikel.Artikel;

import java.util.List;
import java.util.Optional;

public class Test3 {
    public static Optional<Artikel> findById(int id, List<Artikel> artikelListe) {
        return artikelListe
            .stream()
            .filter(a -> a.id() == id)
            .findFirst();
    }

    public static void main(String[] args) {
        List<Artikel> artikelListe = List.of(
            new Artikel(4711, 10.),
            new Artikel(4712, 20.),
            new Artikel(4713, 30.));

        int id = 4712;
        Optional<Artikel> optionalArtikel = findById(id, artikelListe);
```

```
    optionalArtikel.ifPresentOrElse(
        System.out::println,
        () -> System.out.println("Artikel " + id + " nicht gefunden."));
    }
}
```

Das folgende Programm nutzt die `Optional`-Methode:

```
Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)
```

Wenn der Wert der `Optional`-Instanz, für die diese Methode aufgerufen wird, vorhanden ist, wird dieser als `Optional` zurückgeliefert, ansonsten das `Optional`, das vom `Supplier` erzeugt wird.

```
package optional;

import artikel.Artikel;

import java.util.List;
import java.util.Optional;

public class Test4 {
    public static Optional<Artikel> findById(int id, List<Artikel> artikelListe) {
        return artikelListe
            .stream()
            .filter(a -> a.id() == id)
            .findFirst();
    }

    public static void main(String[] args) {
        List<Artikel> list1 = List.of(
            new Artikel(4711, 10.),
            new Artikel(4712, 20.),
            new Artikel(4713, 30.));

        List<Artikel> list2 = List.of(
            new Artikel(4714, 40.),
            new Artikel(4715, 50.),
            new Artikel(4716, 60.));

        int id = 4715;
        Optional<Artikel> optionalArtikel = findById(id, list1)
            .or(() -> findById(id, list2));

        optionalArtikel.ifPresentOrElse(
            System.out::println,
            () -> System.out.println("Artikel " + id + " nicht gefunden."));
    }
}
```

Optional sollte *niemals* bei Attributen einer Klasse verwendet werden wie im folgenden Beispiel:

```
private Optional<String> text;

public Optional<String> getText() {
    return text;
}

public void setText(Optional<String> text) {
    this.text = text;
}
```

Bei Nutzung der Methode `setText` muss der String zunächst in eine `Optional`-Instanz verpackt werden, auch wenn man sich sicher ist, dass der String nicht `null` ist.

Eine bessere Lösung ist die folgende:

```
private String text;

public Optional<String> getText() {
    return Optional.ofNullable(text);
}

public void setText(String text) {
    this.text = text;
}
```

`getText` signalisiert dem Aufrufer, dass der String nicht immer vorhanden ist.

## Fazit

- Die konsequente Verwendung von `Optional` verdeutlicht gut, dass Werte auch "leer" sein können.
- Hat eine Methode beispielsweise den Rückgabetyp `String`, bedeutet das, dass der Wert auf jeden Fall vorhanden ist. Ist der Rückgabetyp aber `Optional<String>`, so heißt das, dass der Wert auch nicht vorhanden sein kann.
- Ausnahmen vom Typ `NullPointerException` zur Laufzeit können damit leichter vermieden werden.
- `Optional` sollte nur beim Rückgabewert eingesetzt werden.

## 23.3 Weitere Beispiele

### Minimum und Maximum

Die folgenden `Stream`-Methoden (terminale Operationen) ermitteln das Minimum bzw. Maximum gemäß dem angegebenen `Comparator`:

```
Optional<T> min(Comparator<? super T> comparator)  
Optional<T> max(Comparator<? super T> comparator)
```

Bei leerem Stream wird ein "leeres" `Optional` zurückgegeben.

### Die `Optional`-Methode

```
<U> Optional<U> map(Function<? super T,? extends U> mapper)
```

liefert ein `Optional`-Objekt mit dem Ergebnis von `mapper`, falls der Wert dieses `Optional`-Objekt existiert, ansonsten ein leeres `Optional`.

```
package minmax;  
  
import artikel.Artikel;  
  
import java.util.Comparator;  
import java.util.List;  
import java.util.Optional;  
  
public class Test {  
    public static void main(String[] args) {  
        List<Artikel> list = List.of(  
            new Artikel(4711, 10.),  
            new Artikel(4712, 20.),  
            new Artikel(4713, 30.));  
  
        Optional<Double> min = list.stream()  
            .min(Comparator.comparing(Artikel::preis))  
            .map(Artikel::preis);  
  
        Optional<Double> max = list.stream()  
            .max(Comparator.comparing(Artikel::preis))  
            .map(Artikel::preis);  
  
        min.ifPresent(x -> System.out.println("Minimum: " + x));  
        max.ifPresent(x -> System.out.println("Maximum: " + x));  
    }  
}
```

Ausgabe des Programms:

```
Minimum: 10.0  
Maximum: 30.0
```

## Optionale Supplier

Das folgende Beispiel zeigt, wie die Rückgabe bei einem `Supplier`-Aufruf in ein `Optional` verpackt werden kann, sodass `null` oder die Auslösung einer Ausnahme ein "leeres" `Optional` liefert.

```
package supplier;

import java.util.Optional;
import java.util.function.Supplier;

public class Test {
    public static <T> Optional<T> optional(Supplier<T> supplier) {
        try {
            return Optional.of(supplier.get());
        } catch (Exception e) {
            return Optional.empty();
        }
    }

    public static void main(String[] args) {
        String text = "123";
        Optional<Integer> value = optional(() -> Integer.parseInt(text));
        value.ifPresentOrElse(System.out::println,
            () -> System.out.println("Keine Zahl"));
    }
}
```

## Das Heronsche Näherungsverfahren

Das *Heronsche Näherungsverfahren*<sup>1</sup> ist ein Rechenverfahren zur Berechnung einer Näherung der Quadratwurzel.

Die Quadratwurzel einer positiven Zahl  $a$  lässt sich durch das folgende rekursive Verfahren bestimmen:

$$\begin{aligned}x_0 &= (a + 1) / 2 \\x_n &= 0.5 * (x_{n-1} + a / x_{n-1}) \text{ für } n > 0, a > 0\end{aligned}$$

Dieses Näherungsverfahren wird mit Hilfe von `Stream`-Methoden implementiert. Mit `iterate` wird der unendliche Stream der einzelnen Folgenglieder erzeugt. Mit einer Filter-Bedingung, die die Genauigkeit der Näherung formuliert, wird der Stream begrenzt. Die terminale Operation `findFirst` terminiert die Suchoperation.

```
package heron;

import java.util.Optional;
import java.util.function.Predicate;
import java.util.stream.Stream;
```

---

1 <https://de.wikipedia.org/wiki/Heron-Verfahren>

```
public class Heron {  
    public static Stream<Double> heron(double a) {  
        return Stream.iterate((a + 1) / 2, x -> 0.5 * (x + a / x));  
    }  
  
    public static <T> Optional<T> find(Stream<T> stream, Predicate<T> predicate) {  
        return stream.filter(predicate).findFirst();  
    }  
  
    public static void main(String[] args) {  
        double a = 2;  
        double EPSILON = 1.e-8;  
  
        Optional<Double> value = find(heron(a), r -> Math.abs(r * r - a) < EPSILON);  
        if (value.isPresent()) {  
            double z = value.get();  
            System.out.println("Wurzel : " + Math.sqrt(a));  
            System.out.println("Näherung: " + z);  
        }  
    }  
}
```

Ausgabe des Programms:

```
Wurzel : 1.4142135623730951  
Näherung: 1.4142135623746899
```

## 23.4 Aufgaben

### Aufgabe 1

Der folgende Quellcode realisiert die Suche nach Personen. Bei einem Treffer wird die Methode gefunden, ansonsten die Methode nichtGefunden aufgerufen:

```
package personen;  
  
import java.util.List;  
  
public class Test1 {  
    public record Person(int id, String name) {}  
  
    private final List<Person> personen;  
  
    public Test1() {  
        personen = List.of(new Person(1001, "Meier"),  
                         new Person(1001, "Schmitz"),  
                         new Person(1002, "Frick"),  
                         new Person(1003, "Winzig"),  
                         new Person(1004, "Riesig")  
        );  
    }  
}
```

```
public Person findById(int id) {  
    for (Person p : personen) {  
        if (p.id() == id)  
            return p;  
    }  
    return null;  
}  
  
public void gefunden(Person p) {  
    System.out.println(p);  
}  
  
public void nichtGefunden() {  
    System.out.println("Person nicht gefunden");  
}  
  
public static void main(String[] args) {  
    Test1 t = new Test1();  
    int id = 1003;  
  
    Person p = t.findById(id);  
    if (p != null)  
        t.gefunden(p);  
    else  
        t.nichtGefunden();  
}
```

Stellen Sie das Programm um, indem Sie die Klasse `Optional` sowie Stream-Methoden analog zu den Beispielen in diesem Kapitel einsetzen.

*Lösung: Paket `personen`*



## 24 Dateien und Datenströme

Java bietet eine umfangreiche Bibliothek von Klassen und Interfaces im Paket `java.io` und `java.nio` zur Verarbeitung von Dateien, zum Lesen von der Tastatur, zur Ausgabe auf dem Bildschirm, zum Senden und Empfangen von Nachrichten über Netzwerkverbindungen und vieles mehr.

### Lernziele

In diesem Kapitel lernen Sie

- wie mit Dateien und Verzeichnissen gearbeitet werden kann,
- was der Unterschied zwischen byte- und zeichenorientierten Datenströmen ist,
- welche Methoden zum Schreiben und Lesen von Dateien existieren und
- wie Dateien komprimiert werden können.

### 24.1 Dateien und Verzeichnisse bearbeiten

#### Standardverzeichnisse

Mit der Methode `getProperty` der Klasse `System` können das aktuelle Verzeichnis, das Home-Verzeichnis und das temporäre Verzeichnis ermittelt werden:

```
String current = System.getProperty("user.dir");
String home = System.getProperty("user.home");
String tmp = System.getProperty("java.io.tmpdir");
```

Objekte der Klasse `java.io.File` repräsentieren Dateien und Verzeichnisse.

Konstruktoren sind:

```
File(String path)
File(String dirName, String name)
File(File fileDir, String name)
```

`path` ist ein Pfadname für ein Verzeichnis oder eine Datei. `dirName` ist ein Verzeichnisname, `name` ein Pfadname für ein Unterverzeichnis oder eine Datei. `dirName` und `name` bilden zusammen den Pfadnamen. Im letzten Fall wird das Verzeichnis durch ein `File`-Objekt benannt. Es wird jedoch beim Erzeugen des `File`-Objekts nicht überprüft, ob die Datei bzw. das Verzeichnis existiert.

Beispiel:

```
File file = new File("src/file/FileInfo.java");
```

Hier muss für Windows der Backslash als Pfadtrenner doppelt angegeben werden. Es kann hier aber auch der normale Schrägstrich genutzt werden.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_24](https://doi.org/10.1007/978-3-658-43574-5_24).

### Informationen über Dateien und Verzeichnisse

```
String getName()
    liefert den letzten Namensbestandteil des Pfadnamens.

String getPath()
    liefert den Pfadnamen.

String getAbsolutePath()
    liefert die komplette Pfadangabe.

String getCanonicalPath() throws java.io.IOException
    liefert die komplette Pfadangabe in kanonischer Form, d. h. Angaben wie "."
    (aktueller Verzeichnis) und ".." (übergeordnetes Verzeichnis) werden entfernt
    bzw. aufgelöst.

String getParent()
    liefert den Namen des übergeordneten Verzeichnisses.

boolean exists()
    liefert true, wenn die Datei bzw. das Verzeichnis existiert.

boolean canRead()
    liefert true, wenn ein lesender Zugriff möglich ist.

boolean canWrite()
    liefert true, wenn ein schreibender Zugriff möglich ist.

boolean isFile()
    liefert true, wenn das Objekt eine Datei repräsentiert.

boolean isDirectory()
    liefert true, wenn das Objekt ein Verzeichnis repräsentiert.

boolean isAbsolute()
    liefert true, wenn das Objekt einen kompletten Pfad repräsentiert.

long length()
    liefert die Länge der Datei in Bytes bzw. 0, wenn die Datei nicht existiert.

long lastModified()
    liefert den Zeitpunkt der letzten Änderung der Datei in Millisekunden seit dem
    1.1.1970 00:00:00 Uhr GMT bzw. 0, wenn die Datei nicht existiert.

package file;

import java.io.File;
import java.io.IOException;
import java.util.Date;

public class FileInfo {
    public static void main(String[] args) throws IOException {
        String current = System.getProperty("user.dir");
        String home = System.getProperty("user.home");
        String tmp = System.getProperty("java.io.tmpdir");
```

```
System.out.println(current);
System.out.println(home);
System.out.println(tmp);

File file = new File("src/file/FileInfo.java");
System.out.println("Name: " + file.getName());
System.out.println("Path: " + file.getPath());
System.out.println("AbsolutePath: " + file.getAbsolutePath());
System.out.println("CanonicalPath: " + file.getCanonicalPath());
System.out.println("Parent: " + file.getParent());
System.out.println("exists: " + file.exists());
System.out.println("canRead: " + file.canRead());
System.out.println("canWrite: " + file.canWrite());
System.out.println("isFile: " + file.isFile());
System.out.println("isDirectory: " + file.isDirectory());
System.out.println("isAbsolute: " + file.isAbsolute());
System.out.println("length: " + file.length());
System.out.println("lastModified: " + new Date(file.lastModified()));
}
}
```

Ausgabe des Programms (Beispiel):

```
/home/dietmar/GKJava12/Kap24-I0/kap24
/home/dietmar
/tmp
Name: FileInfo.java
Path: src/file/FileInfo.java
AbsolutePath: /home/dietmar/GKJava12/Kap24-I0/kap24/src/file/FileInfo.java
CanonicalPath: /home/dietmar/GKJava12/Kap24-I0/kap24/src/file/FileInfo.java
Parent: src/file
exists: true
canRead: true
canWrite: true
isFile: true
isDirectory: false
isAbsolute: false
length: 1354
lastModified: Wed Nov 23 14:13:18 CET 2022
```

## Erstellen, Umbenennen, Löschen

`boolean createNewFile() throws java.io.IOException`

erstellt die von diesem Objekt benannte Datei, wenn sie vorher nicht existiert und liefert `true`, andernfalls wird `false` geliefert.

`boolean mkdir()`

liefert `true`, wenn das von diesem Objekt benannte Verzeichnis erstellt werden konnte.

`boolean mkdirs()`

legt im Unterschied zu `mkdir` auch im Pfad evtl. fehlende Verzeichnisse an.

`boolean renameTo(File newName)`

liefert `true`, wenn die Datei in den neuen Namen umbenannt werden konnte.

```
boolean delete()
```

liefert true, wenn das Verzeichnis bzw. die Datei gelöscht werden konnte. Verzeichnisse müssen zum Löschen leer sein.

## Zugriff auf Verzeichnisse

```
File[] listFiles()
```

liefert für ein Verzeichnis ein Array von File-Objekten zu Datei- und Unterverzeichnisnamen.

```
File[] listFiles(FileFilter filter)
```

verhält sich wie obige Methode, nur dass ausschließlich File-Objekte geliefert werden, die dem spezifischen Filter genügen.

java.io.FileFilter ist ein *Funktionsinterface* mit der Methode

```
boolean accept(File f)
```

Die Methode listFiles liefert ein File-Objekt file genau dann, wenn der Aufruf von filter.accept(file) den Wert true liefert.

Das folgende Programm zeigt nur die Namen derjenigen Dateien eines Verzeichnisses an, die mit einem vorgegebenen Suffix enden.

```
package file;

import java.io.File;
import java.io.FileFilter;

public class FilterTest {
    public static void main(String[] args) {
        String home = System.getProperty("user.home");

        File file = new File(home + "/Dokumente");
        if (!file.isDirectory()) {
            System.err.println("Kein Verzeichnis");
            System.exit(1);
        }

        FileFilter filter = p -> p.isFile() && p.getName().endsWith(".html");
        File[] list = file.listFiles(filter);
        if (list != null) {
            for (File f : list) {
                System.out.println(f.getName());
            }
        }
    }
}
```

## Neues API

Mit Java-Version 7 wurde ein neues API zur Verarbeitung von Verzeichnissen und Dateien eingeführt.

## Path

Das Interface `java.nio.file.Path` repräsentiert einen Dateipfad. `Path` umfasst u. a. die Funktionalität der Klasse `java.io.File`.

File- und Path-Objekte können ineinander umgewandelt werden. Die Path-Methode `toFile()` liefert ein File-Objekt, die File-Methode `toPath()` das entsprechende Path-Objekt.

Die Path-Methoden `getFileName()`, `getParent()` und `getRoot()` liefern das letzte Pfadelement, das übergeordnete Verzeichnis bzw. das Root-Verzeichnis (Name des Laufwerks bei Windows).

Mit der Path-Methode `getName` kann man auf die Namen der Elemente, aus die der Pfad aufgebaut ist, zugreifen (siehe das folgende Programm).

## Paths

Path-Objekte werden mit Hilfe der Methode `get` der Klasse `java.nio.file.Paths` erzeugt:

```
static Path get(String first, String... more)
```

. und .. stehen für das aktuelle bzw. für das übergeordnete Verzeichnis.

`normalize` kürzt den Pfad so, dass überflüssige Elemente wie . und .. verschwinden, `toAbsolutePath` liefert den absoluten Pfad, `toRealPath` liefert einen absoluten Pfad ohne die Kurzschreibweisen ".." und "...".

## Files

Zahlreiche Eigenschaften von Dateien und Verzeichnissen können über statische Methoden der Klasse `java.nio.file.Files` ermittelt werden.

Das folgende Programm zeigt, wie Dateipfade aufgebaut und deren Eigenschaften ausgewertet werden können.

```
package nio;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathTest {
    public static void main(String[] args) throws IOException {
        String current = System.getProperty("user.dir");
        Path path = Paths.get(current, "src", "nio", "PathTest.java");
        System.out.println(path);
```

```

System.out.println("fileName: " + path.getFileName());
System.out.println("parent: " + path.getParent());
System.out.println("root: " + path.getRoot());

System.out.println("exists: " + Files.exists(path));
System.out.println("isDirectory: " + Files.isDirectory(path));
System.out.println("isRegularFile: " + Files.isRegularFile(path));
System.out.println("isReadable: " + Files.isReadable(path));
System.out.println("isWritable: " + Files.isWritable(path));
System.out.println("isHidden: " + Files.isHidden(path));
System.out.println("isExecutable: " + Files.isExecutable(path));
System.out.println("size: " + Files.size(path));
System.out.println("lastModifiedTime: " + Files.getLastModifiedTime(path));
System.out.println("owner: " + Files.getOwner(path));

for (int i = 0; i < path.getNameCount(); i++) {
    System.out.print(path.getName(i) + "|");
}
System.out.println();

for (Path element : path) { // Iteration über die Dateipfad-Elemente
    System.out.print(element + "|");
}
System.out.println();

path = Paths.get("./src/../src/nio/PathTest.java");
System.out.println("normalized: " + path.normalize());
System.out.println("absolutePath: " + path.toAbsolutePath());
System.out.println("realPath: " + path.toRealPath());
}
}

```

### Ausgabe des Programms (Beispiel):

```

/home/dietmar/GKJava12/Kap24-I0/kap24/src/nio/PathTest.java
fileName: PathTest.java
parent: /home/dietmar/GKJava12/Kap24-I0/kap24/src/nio
root: /
exists: true
isDirectory: false
isRegularFile: true
isReadable: true
isWritable: true
isHidden: false
isExecutable: false
size: 1641
lastModifiedTime: 2022-11-23T13:39:14.864986842Z
owner: dietmar
home|dietmar|GKJava12|Kap24-I0|kap24|src|nio|PathTest.java|
home|dietmar|GKJava12|Kap24-I0|kap24|src|nio|PathTest.java|
normalized: src/nio/PathTest.java
absolutePath: /home/dietmar/GKJava12/Kap24-I0/kap24/.src/../src/nio/PathTest.java
realPath: /home/dietmar/GKJava12/Kap24-I0/kap24/src/nio/PathTest.java

```

Die `Files`-Methode `newDirectoryStream` liefert ein Objekt vom Typ des Interface `java.nio.file.DirectoryStream`. Hiermit können alle Dateien und Unterverzeichnisse eines vorgegebenen Verzeichnisses ermittelt und auch gefiltert werden.

Die `Files`-Methode `list` liefert ein `Stream`-Objekt. Hier können dann Methoden wie `filter` und `forEach` mit Lambda-Ausdrücken genutzt werden.

Die `Files`-Methoden `walkFileTree` oder `walk` durchlaufen rekursiv alle Unterverzeichnisse eines vorgegebenen Verzeichnisses.

`walkFileTree` verwendet dazu ein `FileVisitor`-Objekt, dessen Methoden beim Durchlaufen aufgerufen werden. Die Klasse `java.nio.file.SimpleFileVisitor` implementiert das Interface `FileVisitor`.

```
package nio;

import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;
import java.util.stream.Stream;

public class DirTest {
    public static void main(String[] args) {
        Path dir = Paths.get(".");

        System.out.println("--- alle (Variante 1) ---");
        try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
            for (Path path : stream) {
                System.out.println(path.getFileName());
            }
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }

        System.out.println("--- alle (Variante 2) ---");
        try (Stream<Path> stream = Files.list(dir)) {
            stream.forEach(p -> System.out.println(p.getFileName()));
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }

        String suffix = ".iml";
        System.out.println("--- gefiltert (Variante 1) ---");
        DirectoryStream.Filter<Path> filter = path -> Files.isRegularFile(path) &&
            path.toString().toLowerCase().endsWith(suffix);

        try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, filter)) {
            for (Path path : stream) {
                System.out.println(path.getFileName());
            }
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

```

System.out.println("--- gefiltert (Variante 2) ---");
try (Stream<Path> stream = Files.list(dir)) {
    stream.filter(Files::isRegularFile)
        .filter(p -> p.toString().toLowerCase().endsWith(suffix))
        .forEach(p -> System.out.println(p.getFileName()));
} catch (IOException e) {
    System.err.println(e.getMessage());
}

System.out.println("--- rekursiver Durchlauf (Variante 1) ---");
SimpleFileVisitor<Path> visitor = new SimpleFileVisitor<>() {
    @Override
    public FileVisitResult preVisitDirectory(
        Path dir, BasicFileAttributes attrs) {
        System.out.println(dir);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }
};

try {
    Files.walkFileTree(dir, visitor);
} catch (IOException e) {
    System.err.println(e.getMessage());
}

System.out.println("--- rekursiver Durchlauf (Variante 2) ---");
try (Stream<Path> stream = Files.walk(dir)) {
    stream.forEach(System.out::println);
} catch (IOException e) {
    System.err.println(e.getMessage());
}
}
}
}

```

Mit `Files`-Methoden können Dateien und Verzeichnisse erzeugt, verschoben, kopiert und gelöscht werden:

- `createDirectories` erzeugt Verzeichnisse mit Unterverzeichnissen.
- `createFile` erzeugt eine neue, leere Datei.
- `createTempFile` erzeugt eine neue, leere Datei in einem systemspezifischen Verzeichnis. Der vom System erzeugte Dateiname kann mit Präfix und Suffix versehen werden.
- `move` benennt eine Datei um oder verschiebt sie, wenn Quelle und Ziel in unterschiedlichen Verzeichnissen liegen.
- `copy` kopiert eine Datei.
- `delete` löscht eine Datei.

Ausnahmen werden bei `createFile`, `move`, `copy` und `delete` ausgelöst, wenn die Zielfile bereits existiert bzw. die Quelldatei nicht vorhanden ist.

Mit `copy` kann ein `InputStream` in eine Datei geschrieben bzw. der Inhalt einer Datei in einen `OutputStream` übertragen werden. Die `URL`-Methode `openStream` lädt eine Web-Ressource herunter und stellt den Inhalt als `InputStream` zur Verfügung. Zur Erzeugung eines `URL` mit Hilfe der `URI`-Methode `create` siehe Kapitel 30.

```
packagenio;  
  
importjava.io.IOException;  
importjava.net.URI;  
importjava.net.URL;  
importjava.nio.file.Files;  
importjava.nio.file.Path;  
importjava.nio.file.Paths;  
  
publicclassOpTest{  
    publicstaticvoidmain(String[]args){  
        Stringop=args[0];  
  
        try{  
            switch(op){  
                case"create_dir"->Files.createDirectories(Paths.get("dir/sub"));  
                case"create_file"->Files.createFile(Paths.get("dir/test.txt"));  
                case"create_temp_file"->{  
                    Pathpath=Files.createTempFile("test",null);  
                    System.out.println(path);  
                }  
                case"move"->Files.move(Paths.get("dir/test.txt"),  
                                         Paths.get("dir/sub/test.txt"));  
                case"copy"->Files.copy(Paths.get("dir/sub/test.txt"),  
                                         Paths.get("dir/test.txt"));  
                case"delete"->Files.delete(Paths.get("dir/sub/test.txt"));  
                case"copy_from_file"->Files.copy(Paths.get(  
                                         "src/nio/OpTest.java"),System.out);  
                case"copy_from_url"->{  
                    URLurl=URI.create("https://www.google.de").toURL();  
                    Files.copy(url.openStream(),Paths.get("dir/aus.txt"));  
                }  
            }  
        }catch(IOExceptione){  
            System.err.println(e.getMessage());  
        }  
    }  
}
```

## 24.2 Zeichensätze und Codierungen

### Zeichensatz

Ein *Zeichensatz* (*character set*) ist eine Menge von Schriftzeichen, z. B. Buchstaben, Zahlen, Sonderzeichen. Er bestimmt, wie Zeichenfolgen in Bytes umgewandelt werden und umgekehrt (z. B. beim Schreiben und Lesen einer Textdatei).

Instanzen von `java.nio.charset.Charset` repräsentieren Zeichensätze. Die Klasse `java.nio.charset.StandardCharsets` definiert Konstanten für Zeichensätze, z. B. `US_ASCII`, `ISO_8859_1`, `UTF_8`.

### Zeichencodierung

Eine *Zeichencodierung* (*character encoding*) ist die eindeutige Zuordnung von Zahlen zu Schriftzeichen eines Zeichensatzes.

### Codierungsschema

Ein *Codierungsschema* (*encoding scheme*) legt fest, wie ein Zeichen binär darzustellen ist.

Beispiele:

- US-ASCII (8-Bit-Zeichensatz, der aber nur 7 Bits verwendet)
- ISO-8859-1 (8-Bit-Zeichensatz mit Umlauten)
- UTF-8 (Codierung für Unicode-Zeichen, Zeichen werden in verschiedener Länge von 1 bis 4 Bytes codiert)
- Cp1252 (Windows-Standardzeichensatz für westeuropäische Länder)

Wenn eine Anwendung in einer Umgebung (z. B. Linux) entwickelt und getestet und dann in einer anderen Umgebung (z. B. Windows) ausgeführt wird, können Probleme auftreten, da der Java-Standardzeichensatz je nach Betriebssystem und Spracheinstellungen unterschiedlich sein kann. Bei Linux: UTF-8, bei Windows: windows-1252.

Durch Angabe des geeigneten Zeichensatzes (genauer: Codierungsschemas) bei vielen Konstruktoren und Methoden der Java-Klassenbibliothek können diese Probleme gelöst werden. Alternativ kann auch die System Property `file.encoding` bei Aufruf des Programms gesetzt werden, z. B. `java -Dfile.encoding=UTF-8 ...`

### UTF-8 als Standardzeichensatz ab Java 18

Ab Java 18 ist der Standardzeichensatz *auf allen Betriebssystemen* immer UTF-8.

Das folgende Programm zeigt, wie die Zeilen einer Textdatei (`eingabe.txt`) in eine String-Liste übertragen werden können und wie umgekehrt Zeichenketten mit Zeilenvorschub in eine Datei geschrieben werden können. Hier wird jeweils der Standardzeichensatz UTF-8 verwendet.

*eingabe.txt:*

Japanese: こんにちは世界

Deutsch: Hallo Welt

```
packagenio;  
  
importjava.io.IOException;  
importjava.nio.charset.Charset;  
importjava.nio.file.Files;  
importjava.nio.file.Path;  
importjava.nio.file.Paths;  
importjava.util.List;  
  
publicclassReadWriteTest{  
    publicstaticvoidmain(String[]args) throws IOException {  
        System.out.println("Default Charset: " + Charset.defaultCharset());  
        displayCharsets();  
  
        Path pIn = Paths.get("eingabe.txt");  
        Path pOut = Paths.get("ausgabe.txt");  
  
        List<String> lines = Files.readAllLines(pIn);  
        for (int i = 0; i < lines.size(); i++) {  
            lines.set(i, i + 1 + "\t" + lines.get(i));  
        }  
  
        Files.write(pOut, lines);  
    }  
  
    privatestaticvoiddisplayCharsets(){  
        for (String charset : Charset.availableCharsets().keySet()) {  
            System.out.println(charset);  
        }  
    }  
}
```

Ein bestimmter Zeichensatz kann mit der `Charset`-Methode `forName` als letzter Parameter bei `readAllLines` und `write` eingestellt werden, z. B.

```
Charset charset = Charset.forName("ISO-8859-1");
```

Neben den hier für Zeichenketten verwendeten `Files`-Methoden `readAllLines` und `write` gibt es noch die Lese- und Schreibmethoden für Binärdateien:

```
staticbyte[]readAllBytes(Pathpath) throws IOException  
staticPathwrite(Pathpath, byte[]bytes) throws IOException
```

Diese Lese- und Schreibmethoden sind allerdings nur für kleinere Dateien geeignet, da die `String`-Liste bzw. das `byte`-Array komplett im Hauptspeicher abgelegt ist.

## 24.3 Datenströme

Die sequentielle Ein- und Ausgabe wird mittels sogenannter Datenströme realisiert.

Ein *Datenstrom (Stream)* kann als eine Folge von Bytes betrachtet werden, die aus Programmsicht aus einer Datenquelle (*Eingabestrom*) oder in eine Datensenke (*Ausgabestrom*) fließen.

Dabei ist es bei diesem abstrakten Konzept zunächst nicht wichtig, von welchem Eingabegerät gelesen bzw. auf welches Ausgabegerät geschrieben wird. Methoden diverser Klassen bieten die nötigen Zugriffsmöglichkeiten.

Datenströme können so geschachtelt werden, dass lese- und schreibtechnische Erweiterungen, wie z. B. das Puffern von Zeichen, möglich sind.

### Standarddatenströme

Die von der Klasse `System` bereitgestellten *Standarddatenströme* `System.in` (vom Typ `java.io.InputStream`), `System.out` und `System.err` (beide vom Typ `java.io.PrintStream`) zur Eingabe von der Tastatur bzw. zur Ausgabe am Bildschirm können beim Aufruf des Programms mit Hilfe der Symbole `<` und `>` bzw. `>>` so umgelenkt werden, dass von einer Datei gelesen bzw. in eine Datei geschrieben wird.

Beispiel:

```
java Programm < ein > aus
```

Hier wird auf Betriebssystemebene die Tastatur durch die Datei `ein` und der Bildschirm durch die Datei `aus` ersetzt. Mittels Methoden der Klasse `System.err` erzeugte Fehlermeldungen erscheinen am Bildschirm. Sie können aber auch durch Angabe von `>>dateiname` in eine Datei umgelenkt werden. `>>` anstelle von `>` schreibt an das Ende einer bestehenden Datei.

Datenströme können nach

- Ein- und Ausgabe,
- nach der Art der Datenquelle bzw. Datensenke (z. B. Datei, Array, String),
- nach der Art der Übertragung (z. B. gepuffert, gefiltert) und
- nach der Art der verwendeten Dateneinheiten

unterschieden werden.

### Byteströme

*Byteströme* verwenden als Dateneinheit das Byte (8 Bit). Ihre Implementierung wird von den abstrakten Klassen `java.io.InputStream` und `java.io.OutputStream` vorgegeben.

Tabelle 24-1 gibt eine Übersicht über die wichtigsten Klassen. Die Namen der abstrakten Klassen sind kursiv gedruckt. Die Vererbungshierarchie wird durch Einrückungen wiedergegeben.

**Tabelle 24-1:** Byteströme

Eingabe	Ausgabe
<i>InputStream</i>	<i>OutputStream</i>
<i>ByteArrayInputStream</i>	<i>ByteArrayOutputStream</i>
<i>FileInputStream</i>	<i>FileOutputStream</i>
<i>FilterInputStream</i>	<i>FilterOutputStream</i>
<i>BufferedInputStream</i>	<i>BufferedOutputStream</i>
<i>DataInputStream</i>	<i>DataOutputStream</i>
<i>PushbackInputStream</i>	
	<i>PrintStream</i>
<i>ObjectInputStream</i>	<i>ObjectOutputStream</i>
<i>PipedInputStream</i>	<i>PipedOutputStream</i>
<i>SequenceInputStream</i>	

**InputStream / OutputStream**

ist Superklasse aller Byte-Eingabeströme bzw. Byte-Ausgabeströme.

**ByteArrayInputStream / ByteArrayOutputStream**

liest aus bzw. schreibt in byte-Arrays.

**FileInputStream / FileOutputStream**

liest aus bzw. schreibt in Dateien.

**FilterInputStream / FilterOutputStream**

ist mit einem anderen Ein- bzw. Ausgabestrom verbunden und wird benutzt, um Daten unmittelbar nach der Eingabe bzw. vor der Ausgabe zu transformieren.

**BufferedInputStream / BufferedOutputStream**

verfügt über interne Puffer für effiziente Schreib- bzw. Leseoperationen.

**DataInputStream / DataOutputStream**

besitzt Methoden zum Lesen bzw. Schreiben von Werten einfacher Datentypen im Binärformat.

**PushbackInputStream**

kann bereits gelesene Daten in den Eingabestrom zurückstellen.

**PrintStream**

gibt Werte verschiedener Datentypen im Textformat aus.

`ObjectInputStream / ObjectOutputStream`

kann komplexe Objekte schreiben bzw. wieder rekonstruieren.

`PipedInputStream / PipedOutputStream`

bieten Methoden, um Daten zwischen zwei unabhängig laufenden Programmen (*Threads*) über sogenannte *Pipes* auszutauschen.

`SequenceInputStream`

kann aus mehreren Eingabestromen sukzessive lesen.

Alle Zugriffsmethoden lösen im Fehlerfall kontrollierte Ausnahmen vom Typ `java.io.IOException` aus. Die Lesemethoden blockieren bis Eingabedaten vorliegen, das Ende des Datenstroms erreicht ist oder eine Ausnahme ausgelöst wird.

### InputStream-Methoden

Grundlegende Methoden der Klasse `InputStream` sind:

`int available()`

liefert die Anzahl Bytes, die ohne Blockieren gelesen werden können.

`abstract int read()`

liest das nächste Byte aus dem Eingabestrom und gibt es als Wert vom Typ `int` im Bereich von 0 bis 255 zurück. Der Wert -1 zeigt das Ende des Eingabestroms an.

`int read(byte[] b)`

liest maximal `b.length` Bytes, speichert sie in das Array `b` und liefert die Anzahl der tatsächlich gelesenen Bytes als Rückgabewert. Der Wert -1 zeigt das Ende des Eingabestroms an.

`int read(byte[] b, int offset, int count)`

liest maximal `count` Bytes, speichert sie beginnend bei Position `offset` in das Array `b` und liefert die Anzahl der tatsächlich gelesenen Bytes als Rückgabewert. Der Wert -1 zeigt das Ende des Eingabestroms an.

`byte[] readAllBytes()`

liest alle Bytes aus dem Eingabestrom.

`long transferTo(OutputStream out)`

kopiert Daten aus dem Eingabestrom in den Ausgabestrom und gibt die Anzahl der kopierten Bytes zurück.

`void close()`

schließt den Eingabestrom.

### OutputStream-Methoden

Grundlegende Methoden der Klasse `OutputStream` sind:

```
abstract void write(int b)
    schreibt die 8 niederwertigen Bits von b in den Ausgabestrom.
```

```
void write(byte[] b)
    schreibt die Bytes aus dem Array b in den Ausgabestrom.
```

```
void write(byte[] b, int offset, int count)
    schreibt beginnend bei Position offset count Bytes aus dem Array b in den
    Ausgabestrom.
```

```
void flush()
    schreibt alle in Puffern zwischengespeicherten Daten sofort in den Ausgabe-
    strom.
```

```
void close()
    schließt den Ausgabestrom. Bei FilterOutputStream-Objekten wird vorher
    flush automatisch aufgerufen.
```

### Zeichenströme

Zeichenströme sind von den abstrakten Klassen `java.io.Reader` und `java.io.Writer` abgeleitet und lesen bzw. schreiben Unicode-Zeichen vom Typ `char`.

**Tabelle 24-2:** Zeichenströme

Eingabe	Ausgabe
<code>Reader</code>	<code>Writer</code>
<code>BufferedReader</code>	<code>BufferedWriter</code>
<code>LineNumberReader</code>	
<code>CharArrayReader</code>	<code>CharArrayWriter</code>
<code>FilterReader</code>	<code>FilterWriter</code>
<code>PushbackReader</code>	
<code>InputStreamReader</code>	<code>OutputStreamWriter</code>
<code>FileReader</code>	<code>FileWriter</code>
<code>PipedReader</code>	<code>PipedWriter</code>
<code>StringReader</code>	<code>StringWriter</code>
	<code>PrintWriter</code>

Tabelle 24-2 gibt eine Übersicht über die wichtigsten Klassen. Die Namen der abstrakten Klassen sind kursiv gedruckt. Die Vererbungshierarchie wird durch Einrückungen wiedergegeben.

**Reader / Writer**

ist Superklasse aller zeichenorientierten Eingabe- bzw. Ausgabeströme.

**BufferedReader / BufferedWriter**

verfügt über interne Puffer für effiziente Lese- bzw. Schreiboperationen.

**LineNumberReader**

hat die Fähigkeit, Zeilen zu zählen.

**CharArrayReader / CharArrayWriter**

liest aus bzw. schreibt in char-Arrays.

**FilterReader / FilterWriter**

ist mit einem anderen Ein- bzw. Ausgabestrom verbunden und wird benutzt, um Daten unmittelbar nach der Eingabe bzw. vor der Ausgabe zu transformieren.

**PushbackReader**

kann bereits gelesene Daten in den Eingabestrom zurückstellen.

**InputStreamReader / OutputStreamWriter**

liest Bytes von einem `InputStream` und wandelt sie in char-Werte bzw. wandelt char-Werte in Bytes und schreibt sie in einen `OutputStream`.

**FileReader / FileWriter**

liest aus einer bzw. schreibt in eine Datei.

**PipedReader / PipedWriter**

bieten Methoden, um Daten zwischen zwei unabhängig laufenden Programmen (*Threads*) über sogenannte *Pipes* auszutauschen.

**StringReader / StringWriter**

liest Zeichen aus einem String bzw. schreibt Zeichen in einen String.

**PrintWriter**

gibt Werte verschiedener Datentypen im Textformat aus.

Alle Zugriffsmethoden lösen im Fehlerfall kontrollierte Ausnahmen vom Typ `java.io.IOException` aus. Die Lesemethoden blockieren, bis Eingabedaten vorliegen, das Ende des Datenstroms erreicht ist oder eine Ausnahme ausgelöst wird.

**Reader-Methoden**

Grundlegende Methoden der Klasse Reader sind:

```
int read()
    liest das nächste Zeichen aus dem Eingabestrom und gibt es als Wert vom Typ
    int im Bereich von 0 bis 65535 zurück. Der Wert -1 zeigt das Ende des
    Eingabestroms an.
```

```
int read(char[] c)
    liest maximal c.length Zeichen, speichert sie in das Array c und liefert die
    Anzahl der tatsächlich gelesenen Zeichen als Rückgabewert. Der Wert -1 zeigt
    das Ende des Eingabestroms an.
```

```
abstract int read(char[] c, int offset, int count)
    liest maximal count Zeichen, speichert sie beginnend bei Position offset in das
    Array c und liefert die Anzahl der tatsächlich gelesenen Zeichen als Rückgabewert.
    Der Wert -1 zeigt das Ende des Eingabestroms an.
```

```
long transferTo(Writer out)
    kopiert Daten aus dem Eingabestrom in den Ausgabestrom und gibt die Anzahl
    der kopierten Zeichen zurück.
```

```
abstract void close()
    schließt den Eingabestrom.
```

## Writer-Methoden

Grundlegende Methoden der Klasse `Writer` sind:

```
void write(int c)
    schreibt die 16 niederwertigen Bits von c in den Ausgabestrom.
```

```
void write(char[] c)
    schreibt die Zeichen aus dem Array c in den Ausgabestrom.
```

```
abstract void write(char[] c, int offset, int count)
    schreibt beginnend bei Position offset count Zeichen aus dem Array c in den
    Ausgabestrom.
```

```
void write(String s)
    schreibt die Zeichen aus s in den Ausgabestrom.
```

```
void write(String s, int offset, int count)
    schreibt beginnend bei Position offset count Zeichen aus s in den Aus-
    gabestrom.
```

```
abstract void flush()
    schreibt in Puffern enthaltene Daten sofort in den Ausgabestrom.
```

```
abstract void close()
    schreibt alle in Puffern zwischengespeicherten Daten heraus und schließt den
    Ausgabestrom.
```

## 24.4 Daten lesen und schreiben

Mit Hilfe der Klassen `FileInputStream` und `FileOutputStream` kann byteorientiert auf Dateien zugegriffen werden.

### `FileInputStream / FileOutputStream`

```
FileInputStream(File file) throws FileNotFoundException
FileInputStream(String filename) throws FileNotFoundException
    erzeugen jeweils einen FileInputStream für die angegebene Datei.

OutputStream(File file) throws FileNotFoundException
OutputStream(String filename) throws FileNotFoundException
OutputStream(String filename, boolean append)
    throws FileNotFoundException
erzeugen jeweils einen OutputStream für die angegebene Datei. Wenn die Datei nicht existiert, wird sie erzeugt. Eine bestehende Datei wird fortgeschrieben, wenn append den Wert true hat, sonst überschrieben.
```

Die Ausnahme `FileNotFoundException` ist eine Subklasse von `IOException`.

Das folgende Programm kopiert eine Datei byteweise.

```
package copy;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyV1 {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("src/copy/CopyV1.java");
            out = new FileOutputStream("CopyV1.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            try {
                if (in != null) {
                    in.close();
                }
            } catch (IOException ignored) {}
        }
        try {
            if (out != null) {
```

```
        out.flush();
        out.close();
    }
} catch (IOException ignored) {
}
}
```

Das Programm ist so codiert (`finally`), dass die beiden Dateien ordnungsgemäß am Ende geschlossen werden, auch wenn eine Ausnahme im `try`-Block ausgelöst wird.

### Automatic Resource Management (ARM)

Ab Java-Version 7 kann die Codierung stark vereinfacht werden. Instanzen der Datenströme werden innerhalb von runden Klammern der `try`-Anweisung erzeugt. Am Ende werden die Dateien dann in jedem Fall (auch wenn eine Ausnahme im `try`-Block ausgelöst wird) implizit geschlossen.

Solche `try`-Anweisungen (*try with resources*) können `catch`- und `finally`-Blöcke wie normale `try`-Anweisungen haben. Diese werden dann ausgeführt, nachdem die Ressourcen geschlossen wurden (siehe Variante `CopyV2`).

### AutoCloseable

Klassen, die das Interface `java.lang.AutoCloseable` implementieren, wie z. B. die Klassen `FileInputStream` und `FileOutputStream`, können auf diese Weise implizit geschlossen werden. Zu den Klassen, die `AutoCloseable` implementieren, gehören u. a. `java.util.Scanner`, `java.net.Socket`, `java.net.ServerSocket` sowie die Klassen, die `java.sql.Connection`, `java.sql.Statement` und `java.sql.ResultSet` implementieren.

```
package copy;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyV2 {
    public static void main(String[] args) throws IOException {
        try (FileInputStream in = new FileInputStream("src/copy/CopyV2.java");
             FileOutputStream out = new FileOutputStream("CopyV2.txt")) {

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }
    }
}
```

Ab Java-Version 9 können die Ressourcen auch außerhalb des `try`-Blocks definiert werden (Programm CopyV3):

```
FileInputStream in = new FileInputStream("src/copy/CopyV3.java");
FileOutputStream out = new FileOutputStream("CopyV3.txt");

try (in; out) {
    ...
}
```

Das nächste Programm (CopyV4) ist die im Allgemeinen schnellere, gepufferte Version von CopyV2.

Die Klassen `BufferedInputStream` und `BufferedOutputStream` verwenden intern eine Pufferung, um Lese- bzw. Schreibzugriffe zu optimieren.

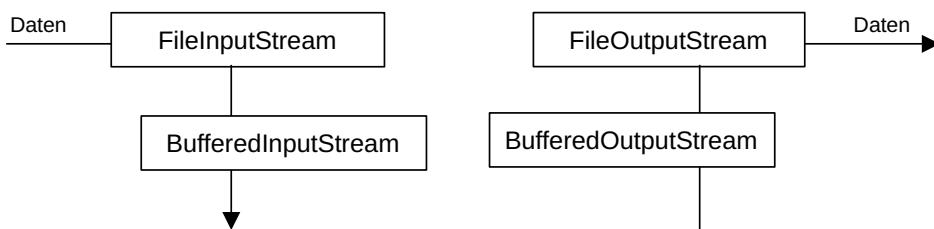
### **BufferedInputStream / BufferedOutputStream**

```
BufferedInputStream(InputStream in)
BufferedInputStream(InputStream in, int size)
```

erzeugen jeweils einen `BufferedInputStream`, der aus dem angegebenen `InputStream` liest. `size` ist die Größe des verwendeten Puffers. Im ersten Fall wird eine Standardgröße benutzt.

```
BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)
```

erzeugen jeweils einen `BufferedOutputStream`, der in den Strom `out` schreibt. `size` ist die Größe des verwendeten Puffers. Im ersten Fall wird eine Standardgröße benutzt.



**Abbildung 24-1:** Verschachtelte Ströme

Eine Instanz vom Typ `BufferedInputStream` wird erzeugt, indem man dem Konstruktor eine Instanz vom Typ `InputStream` übergibt.

Das Beispiel zeigt, wie die Klasse `BufferedInputStream` die Funktionalität der Klasse `InputStream` mit der Fähigkeit der Pufferung "dekoriert". Ähnliches gilt für die Klasse `BufferedOutputStream`.

```
package copy;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyV4 {
    public static void main(String[] args) throws IOException {
        try (BufferedInputStream in = new BufferedInputStream(
            new FileInputStream("src/copy/CopyV4.java"));
            BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream("CopyV4.txt"))) {

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }
    }
}
```

### Daten im Binärformat lesen und schreiben

Werte einfacher Datentypen können plattformunabhängig gelesen und geschrieben werden.

`DataInputStream(InputStream in)`  
erzeugt einen DataInputStream, der aus dem angegebenen `InputStream` liest.

`DataOutputStream(OutputStream out)`  
erzeugt einen DataOutputStream, der in den `OutputStream` `out` schreibt.

### DateOutputStream

Methoden der Klasse `DataOutputStream`:

```
void writeBoolean(boolean x)
void writeChar(int x)
void writeByte(int x)
void writeShort(int x)
void writeInt(int x)
void writeLong(long x)
void writeFloat(float x)
void writeDouble(double x)
```

Diese Methoden schreiben Werte vom einfachen Datentyp im Binärformat. So schreibt z. B. `writeInt` 4 Bytes.

`void writeBytes(String s)`  
schreibt die Zeichenkette `s` als Folge von Bytes. Je Zeichen werden nur die 8 niederwertigen Bits geschrieben.

```
void writeChars(String s)
```

schreibt die Zeichenkette s als Folge von char-Werten. Jedes Zeichen wird als zwei Bytes geschrieben.

```
void writeUTF(String s)
```

schreibt eine Zeichenkette in einem leicht modifizierten UTF-8-Format.

## UTF-8

*UTF-8 (Unicode Transformation Format)* ist eine byteorientierte Codierung von Unicode-Zeichen in variabler Länge (1 - 4 Bytes). ASCII-Zeichen mit Werten aus dem Bereich 0 bis 127 werden als ein Byte mit dem gleichen Wert dargestellt. Diese kompakte Repräsentation von Unicode-Zeichen sorgt für einen sparsamen Verbrauch von Speicherplatz.

Die Ausgabe der Methode `writeUTF` besteht aus zwei Bytes für die Anzahl der folgenden Bytes, gefolgt von den codierten Zeichen.

Alle diese Methoden können die Ausnahme `IOException` auslösen.

## DataInputStream

Methoden der Klasse `DataInputStream`:

```
boolean readBoolean()  
char readChar()  
byte readByte()  
short readShort()  
int readInt()  
long readLong()  
float readFloat()  
double readDouble()
```

Diese Methoden lesen Werte im Binärformat, die von entsprechenden Methoden der Klasse `DataOutputStream` geschrieben wurden.

```
String readUTF()
```

liest Zeichen im *UTF-Format* (UTF-8), wie sie von der Methode `writeUTF` der Klasse `DataOutputStream` geschrieben wurden.

```
int readUnsignedByte()
```

liest ein Byte, erweitert es mit Nullen zum Typ `int` und gibt den Wert im Bereich von 0 bis 255 zurück. Diese Methode ist geeignet, ein Byte zu lesen, das von `writeByte` mit einem Argument im Bereich von 0 bis 255 geschrieben wurde.

```
int readUnsignedShort()
```

liest zwei Bytes und gibt einen `int`-Wert im Bereich von 0 bis 65535 zurück. Diese Methode ist geeignet, zwei Bytes zu lesen, die von `writeShort` mit einem Argument im Bereich von 0 bis 65535 geschrieben wurden.

```
void readFully(byte[] b)
    liest b.length Bytes und speichert sie in b.

void readFully(byte[] b, int offset, int count)
    liest count Bytes und speichert sie in b ab Index offset.
```

Alle diese Methoden können die Ausnahme `IOException` auslösen.

Wenn die gewünschten Bytes nicht gelesen werden können, weil das Ende des Eingabestroms erreicht ist, wird die Ausnahme `EOFException` (Subklasse von `IOException`) ausgelöst.

Das Programm `DataTest` schreibt und liest Daten im Binärformat.

```
package data;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Random;

public class DataTest {
    public static void main(String[] args) throws IOException {
        String name = "zahlen.data";

        try (DataOutputStream out = new DataOutputStream(new FileOutputStream(
            name))) {
            out.writeUTF("Zufallszahlen:");

            Random random = new Random();
            for (int i = 0; i < 100; i++) {
                int z = random.nextInt(100);
                out.writeInt(z);
            }

            System.out.println("Size: " + new File(name).length());
        }

        try (DataInputStream in = new DataInputStream(new FileInputStream(name))) {
            System.out.println(in.readUTF());

            while (true) {
                System.out.print(in.readInt() + " ");
            }
        } catch (EOFException ignored) {
        }
    }
}
```

Die Datei `zahlen.data` enthält 416 Bytes:  $(2 + 14) + 4 * 100$

## Pushback

Mit Hilfe eines `PushbackInputStream` können bereits gelesene Bytes in den Eingabestrom zurückgestellt und anschließend wieder gelesen werden.

Dies ist in manchen Situationen hilfreich, wenn zum Beispiel die Behandlung eines gerade gelesenen Zeichens vom nachfolgenden Zeichen abhängt.

```
PushbackInputStream(InputStream in)
PushbackInputStream(InputStream in, int size)
```

erzeugen jeweils einen `PushbackInputStream`, der aus dem angegebenen `InputStream` liest. `size` ist die Größe des Pushback-Puffers. Beim ersten Konstruktor kann der Puffer genau ein Byte aufnehmen.

Methoden der Klasse `PushbackInputStream`:

```
void unread(int b) throws IOException
```

stellt das Byte `b` in den Eingabestrom zurück. Ein zurückgestelltes Byte steht beim nächsten Lesen wieder zur Verfügung.

```
void unread(byte[] b) throws IOException
```

stellt das Array `b` in den Eingabestrom zurück, indem es an den Anfang des Pushback-Puffers kopiert wird.

```
void unread(byte[] b, int offset, int count) throws IOException
```

stellt `count` Bytes aus dem Array `b` ab der Position `offset` in den Eingabestrom zurück, indem das Teil-Array an den Anfang des Pushback-Puffers kopiert wird.

Das Programm `Kompression` enthält einen einfachen *Komprimierungsalgorithmus*.

Aufeinander folgende gleiche Bytes werden durch drei Bytes ersetzt: `@`, das Wiederholungsbyte, die Anzahl der gleichen Bytes (in einem Byte codiert). Die Komprimierung findet statt, wenn mindestens vier gleiche Bytes aufeinander folgen. `@` selbst darf *nicht* in der ursprünglichen Datei vorkommen.

```
package pushback;

import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PushbackInputStream;

public class Kompression {
    public static void main(String[] args) throws IOException {
        try (PushbackInputStream in = new PushbackInputStream(
            new FileInputStream("daten.txt"));
            BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream("daten_komprimiert.txt"))) {

            int z, b, next;
            while ((b = in.read()) != -1) {
```

```
// zählt die Anzahl gleicher Bytes b
for (z = 1; (next = in.read()) != -1; z++) {
    if (b != next || z == 255)
        break;
}

// Komprimierung nur bei mindestens 4 gleichen Bytes
if (z > 3) {
    out.write('@');
    out.write(b);
    out.write(z);
} else {
    for (int i = 0; i < z; i++)
        out.write(b);
}

// letztes Byte next wird zurückgestellt, da b != next bzw. z == 255
if (next != -1)
    in.unread(next);
}
}
}
```

## 24.5 Texte lesen und schreiben

Byteströme können als Zeichenströme auf der Basis eines Zeichensatzes interpretiert werden.

Hierbei helfen die beiden Klassen `InputStreamReader` und `OutputStreamWriter`.

### `InputStreamReader / OutputStreamWriter`

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String charsetName)
    throws UnsupportedEncodingException
InputStreamReader(InputStream in, Charset cs)
```

erzeugen jeweils einen `InputStreamReader`, der aus einem `InputStream` liest. `charsetName` bezeichnet den Zeichensatz, auf dem die Umwandlung von Bytes in char-Werte basiert. Im ersten Fall wird mit der voreingestellten Standardzeichencodierung (siehe Java-Systemeigenschaft `file.encoding`) gelesen.

```
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, String charsetName)
    throws UnsupportedEncodingException
OutputStreamWriter(OutputStream out, Charset cs)
```

erzeugen jeweils einen `OutputStreamWriter`, der in den `OutputStream` schreibt. `charsetName` bezeichnet den Zeichensatz, auf dem die Umwandlung von char-Werten in Bytes basiert. Im ersten Fall wird die Standardzeichencodierung verwendet.

Bekannte Zeichensätze bzw. Codierungsschemata sind:

US-ASCII, ISO-8859-1, UTF-8, UTF-16.

Das Programm `EncodingTest` nutzt *UTF-8*, um einen String in eine Datei zu schreiben. Schließlich wird der String aus der Datei wieder gelesen, indem der gleiche Zeichensatz verwendet wird. Dieser String wird dann mit dem Originalstring auf Gleichheit geprüft.

```
package text;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.nio.charset.StandardCharsets;

public class EncodingTest {
    public static void main(String[] args) throws IOException {
        String s1 = "Alpha \u03b1, Beta \u03b2, Epsilon \u03b5";

        try (OutputStreamWriter out = new OutputStreamWriter(
                new FileOutputStream("data.txt"), StandardCharsets.UTF_8)) {
            out.write(s1);
        }

        try (InputStreamReader in = new InputStreamReader(
                new FileInputStream("data.txt"), StandardCharsets.UTF_8)) {
            int c;
            StringBuilder sb = new StringBuilder();
            while ((c = in.read()) != -1) {
                sb.append((char) c);
            }

            String s2 = sb.toString();
            System.out.println(s1.equals(s2));
        }
    }
}
```

Wird beim Lesen statt *UTF-8* z. B. *US-ASCII* verwendet, können die Originalstrings nicht mehr rekonstruiert werden. Der Vergleich liefert dann `false`.

Ab Java 18 ist *UTF-8* die Standardeinstellung, sodass auf die Angabe dieses Zeichensatzes verzichtet werden kann.

Für Zeichenströme existieren die zu den Byteströmen analogen Klassen und Konstrukturen:

### **FileReader / FileWriter**

```
FileReader(File file) throws FileNotFoundException
FileReader(String name) throws FileNotFoundException
```

```
FileWriter(File file) throws IOException  
FileWriter(File file, boolean append) throws IOException  
FileWriter(String name) throws IOException  
FileWriter(String name, boolean append) throws IOException
```

FileReader und FileWriter nutzen die voreingestellte Zeichencodierung.

### BufferedReader / BufferedWriter

```
BufferedReader(Reader in)  
BufferedReader(Reader in, int size)
```

Die BufferedReader-Methode

```
String readLine() throws IOException
```

liest eine komplette Textzeile. Der zurückgegebene String enthält nicht das Zeilentrennzeichen. readLine gibt null zurück, wenn das Ende des Datenstroms erreicht ist.

```
BufferedWriter(Writer out)  
BufferedWriter(Writer out, int size)
```

Die BufferedWriter-Methode

```
void newLine() throws IOException
```

schreibt einen Zeilentrenner gemäß der Java-Systemeigenschaft line.separator.

Das Programm Tastatur liest Eingaben von der Tastatur. Die Eingabeschleife kann mit der Eingabe-Taste (Return) beendet werden. Die Tastenkombination Strg+Z oder Strg+D (je nach Betriebssystem) signalisiert ebenfalls das Ende der Eingabe (readLine liefert dann null).

```
package text;  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
  
public class Tastatur {  
    public static void main(String[] args) throws IOException {  
        try (BufferedReader in = new BufferedReader(new InputStreamReader(  
            System.in))) {  
            String line;  
            while (true) {  
                System.out.print("> ");  
  
                line = in.readLine();  
                if (line == null || line.isEmpty())  
                    break;  
            }  
        }  
    }  
}
```

```
        System.out.println(line);
    }
}
}
```

Das folgende Programm gibt gelesene Zeilen einer Textdatei mit ihrer Zeilennummer aus.

```
package text;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Zeilennummern {
    public static void main(String[] args) throws IOException {
        try (BufferedReader in = new BufferedReader(
            new FileReader("src/Zeilennummern.java")));
            BufferedWriter out = new BufferedWriter(
                new FileWriter("Zeilennummern.txt"))) {

            int c = 0;
            String line;
            while ((line = in.readLine()) != null) {
                out.write(++c + ": ");
                out.write(line);
                out.newLine();
            }
        }
    }
}
```

## PrintWriter

```
PrintWriter(String filename)
PrintWriter(File file)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoflush)
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoflush)
```

erzeugen jeweils einen PrintWriter. Hat autoflush den Wert true, so wird der verwendete Puffer immer dann geleert, wenn die Methode println aufgerufen wird. Dies geschieht auch bei Verwendung der ersten beiden Konstruktoren.

Methoden der Klasse `PrintWriter`:

`void print(Typ x)`

schreibt den angegebenen Parameter im Textformat in den Ausgabestrom. *Typ* steht hier für `boolean`, `char`, `int`, `long`, `float`, `double`, `char[]`, `String` oder `Object`. Im letzten Fall wird die String-Darstellung des Objekts ausgegeben, wie sie die Methode `toString()` der Klasse `Object` bzw. die überschreibende Methode liefert.

`void println(Typ x)`

verhält sich wie obige Methode mit dem Unterschied, dass zusätzlich der Zeilentrenner gemäß der Java-Systemeigenschaft `line.separator` geschrieben wird.

`void println()`

gibt nur den Zeilentrenner aus.

## Formatierte Ausgabe

`PrintWriter printf(String format, Object... args)`

schreibt einen formatierten String und liefert die Referenz auf die `PrintWriter`-Instanz zurück. `format` enthält die Formatangaben für jedes Argument der mit dem `Varargs`-Parameter `args` bezeichneten Liste.

Wir stellen hier eine *vereinfachte Format-Syntax* zur Ausgabe der einzelnen Argumente vor, bei der die Reihenfolge der einzelnen Formatangaben mit der Reihenfolge der Argumente übereinstimmen muss.

Format-Syntax zur Ausgabe eines Arguments:

`%[flags][width][.precision]conversion`

`flags` steuert die Ausgabe:

- 0 führende Nullen bei Zahlen
- + Vorzeichen bei positiven Zahlen
- linksbündige Ausgabe

`width` gibt die minimale Anzahl Zeichen an, die ausgegeben werden sollen, `precision` gibt für Zahlen die Anzahl Nachkommastellen an.

`conversion` gibt an, wie das Argument aufbereitet werden soll:

- d ganze Zahl
- f Fließkommazahl
- o Oktalzahl
- x Hexadezimalzahl
- s String
- n Zeilenvorschub
- % das Prozentzeichen selbst

Optional kann an die Methode `printf` auch ein `Locale`-Objekt als erstes Argument übergeben werden.

### PrintStream

Obige Methoden existieren auch für den Bytestrom `PrintStream`.

Ebenso existieren entsprechend die ersten vier Konstruktoren des `PrintWriter` analog für `PrintStream`.

```
package text;

public class FormatTest {
    public static void main(String[] args) {
        String[] artikel = {"Zange", "Hammer", "Bohrmaschine"};
        double[] preise = {3.99, 2.99, 44.99};

        double sum = 0;
        for (int i = 0; i < artikel.length; i++) {
            System.out.printf("%-15s %8.2f%n", artikel[i], preise[i]);
            sum += preise[i];
        }

        System.out.printf("%-15s -----%n", " ");
        System.out.printf("%-15s %8.2f%n", " ", sum);
    }
}
```

Ausgabe des Programms:

Zange	3,99
Hammer	2,99
Bohrmaschine	44,99
<hr style="width: 100px; margin-left: 0;"/>	
	51,97

### Files-Methoden

Die Klasse `java.nio.file.Files` hat die Methoden `writeString` und `readString`, die Strings in eine Datei schreiben bzw. daraus lesen.

Das folgende Programm veranschaulicht diese Möglichkeiten.

```
package text;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class ReadAndWriteStrings {
    public static void main(String[] args) throws IOException {
```

```
Path file = Paths.get("sample.txt");

Files.writeString(file, "Erste Zeile\n");
Files.writeString(file, "Zweite Zeile\n", StandardOpenOption.APPEND);

String content = Files.readString(file);
content.lines().forEach(System.out::println);
}
```

Die Strings werden im *UTF-8-Format* geschrieben und gelesen. `writeString` öffnet die Datei zum Schreiben. Sie wird erzeugt, wenn sie nicht existiert, und sonst überschrieben. Die Option `StandardOpenOption.APPEND` sorgt dafür, dass die Datei fortgeschrieben wird.

`content.lines()` liefert einen Stream aus den durch Zeilentrennzeichen getrennten Strings.

## 24.6 Datenströme filtern

Die Klassen `FilterInputStream` und `FilterOutputStream` bzw. `FilterReader` und `FilterWriter` ermöglichen es, Daten unmittelbar vor dem Schreiben bzw. nach dem Lesen zu transformieren.

Hierzu müssen dann eigene Subklassen dieser Klassen gebildet werden, um in den `write`-Methoden die Ausgabedaten unmittelbar vor dem Schreiben bzw. in den `read`-Methoden die Eingabedaten unmittelbar nach dem Lesen zu manipulieren.

Das folgende Programm demonstriert die Vorgehensweise.

Das Programm nutzt eine Subklasse der Klasse `FilterWriter`, um Umlaute und 'ß' in die Zeichen ae, oe, ue bzw. ss zu wandeln.

In der von `FilterWriter` abgeleiteten Klasse `UmlautWriter` werden die fünf `write`-Methoden überschrieben. Dabei werden die Ausgabezeichen vor der Übergabe an die Superklassenmethode manipuliert (*Filterfunktion*). Die an den `UmlautWriter` gerichteten Schreibaufrufe werden über einen `BufferedWriter` an einen `FileWriter` weitergeleitet.

```
package filter;

import java.io.FilterWriter;
import java.io.IOException;
import java.io.Writer;

public class UmlautWriter extends FilterWriter {
    public UmlautWriter(Writer out) {
        super(out);
    }
}
```

```

public void write(int c) throws IOException {
    switch ((char) c) {
        case 'ä' -> super.write("ae");
        case 'ö' -> super.write("oe");
        case 'ü' -> super.write("ue");
        case 'Ä' -> super.write("Ae");
        case 'Ö' -> super.write("Oe");
        case 'Ü' -> super.write("Ue");
        case 'ß' -> super.write("ss");
        default -> super.write(c);
    }
}

public void write(char[] c, int offset, int count) throws IOException {
    for (int i = 0; i < count; i++)
        write(c[offset + i]);
}

public void write(char[] c) throws IOException {
    write(c, 0, c.length);
}

public void write(String s, int offset, int count) throws IOException {
    for (int i = 0; i < count; i++)
        write(s.charAt(offset + i));
}

public void write(String s) throws IOException {
    write(s, 0, s.length());
}
}

package filter;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Umlaute {
    public static void main(String[] args) throws IOException {
        try (BufferedReader in = new BufferedReader(
            new FileReader("Text-mit-Umlauten.txt")));
            UmlautWriter out = new UmlautWriter(new BufferedWriter(
                new FileWriter("ausgabe.txt")))) {

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }
    }
}

```

### *Text-mit-Umlauten.txt:*

Oh hätte ich doch geschwiegen, so wäre ich ein Philosoph geblieben!  
 Oh wie gut, dass niemand weiß, dass ich Rumpelstielzchen heiße.

Das Programm erzeugt die Ausgabe in *ausgabe.txt*:

Oh haette ich doch geschwiegen, so waere ich ein Philosoph geblieben!  
Oh wie gut, dass niemand weiss, dass ich Rumpelstielzchen heiss.

## 24.7 Wahlfreier Dateizugriff

Objekte der Klasse `RandomAccessFile` ermöglichen den Zugriff auf sogenannte Random-Access-Dateien.

Eine *Random-Access-Datei* ist eine Datei mit wahlfreiem Zugriff (Direktzugriff), die entweder nur zum Lesen oder zum Lesen und Schreiben geöffnet werden kann. Sie verhält sich wie ein großes Array von Bytes. Ein *Dateizeiger* (*Filepointer*) markiert die Stelle, an der das nächste Zeichen gelesen oder geschrieben wird.

```
RandomAccessFile(String name, String mode) throws FileNotFoundException  
RandomAccessFile(File file, String mode) throws FileNotFoundException  
erzeugen jeweils ein RandomAccessFile-Objekt für die angegebene Datei.  
mode gibt die Art des Zugriffs an. "r" steht für den Lesezugriff, "rw" für den  
Lese- und Schreibzugriff. Eine Datei wird neu angelegt, wenn sie beim Öffnen  
im Modus "rw" nicht existiert.
```

Alle Zugriffs Routinen lösen im Fehlerfall Ausnahmen vom Typ `IOException` aus.

Einige Methoden:

```
long getFilePointer()  
liefert die aktuelle Position des Dateizeigers. Das erste Byte der Datei hat die  
Position 0.  
  
void seek(long pos)  
setzt die Position des Dateizeigers auf pos.  
  
int skipBytes(int n)  
versucht n Bytes zu überspringen und liefert die Anzahl der übersprungenen  
Bytes.  
  
long length()  
liefert die Größe der Datei in Bytes.  
  
void setLength(long newLength)  
setzt die Größe der Datei auf newLength Bytes. Ist die aktuelle Größe der Datei  
größer als newLength, so wird die Datei abgeschnitten, ist sie kleiner als  
newLength, so wird sie mit von der Implementierung gewählten Byte-Werten auf  
die neue Länge vergrößert.  
  
void close()  
schließt die Datei.
```

Die Klasse `RandomAccessFile` enthält die gleichen Methoden zum Lesen und Schreiben wie die Klassen `DataInputStream` und `DataOutputStream`.

Das folgende Programm erstellt eine Artikeldatei, indem neu aufzunehmende Artikel an das Ende der Datei geschrieben werden. Auf Artikel kann mit Hilfe der Artikelnummer zugegriffen werden. Außerdem kann der Lagerbestand eines Artikels erhöht bzw. vermindert werden.

```
package random_access;

public record Artikel(int nr, double preis, int bestand) {
}

package random_access;

import java.io.EOFException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Optional;

public class ArtikelManager {
    private final RandomAccessFile file;

    public ArtikelManager(String name) throws IOException {
        file = new RandomAccessFile(name, "rw");
    }

    public void close() throws IOException {
        file.close();
    }

    public Optional<Artikel> getArtikel(int nr) throws IOException {
        boolean found = false;
        int artNr;

        file.seek(0L);
        try {
            while (!found) {
                artNr = file.readInt();
                if (artNr == nr) {
                    found = true;
                } else {
                    // Preis (8 Bytes) und Bestand (4 Bytes) überspringen
                    file.skipBytes(12);
                }
            }
        } catch (EOFException e) {
            return Optional.empty();
        }

        double preis = file.readDouble();
        int bestand = file.readInt();
        return Optional.of(new Artikel(nr, preis, bestand));
    }
}
```

```
public void list() throws IOException {
    file.seek(0L);
    try {
        while (true) {
            int artNr = file.readInt();
            double preis = file.readDouble();
            int bestand = file.readInt();
            System.out.printf("%4d %8.2f %8d%n", artNr, preis, bestand);
        }
    } catch (EOFException ignored) {
    }
}

public boolean addArtikel(Artikel a) throws IOException {
    if (getArtikel(a.nr()).isEmpty()) {
        file.seek(file.length()); // Zeiger auf Ende setzen
        file.writeInt(a.nr());
        file.writeDouble(a.preis());
        file.writeInt(a.bestand());
        return true;
    } else
        return false;
}

public boolean addBestand(int nr, int zugang) throws IOException {
    Optional<Artikel> artikel = getArtikel(nr);
    if (artikel.isEmpty())
        return false;
    else {
        // Zeiger steht hinter Bestand,
        // Zeiger nun auf das erste Byte von Bestand setzen
        file.seek(file.getFilePointer() - 4L);
        file.writeInt(artikel.get().bestand() + zugang);
        return true;
    }
}

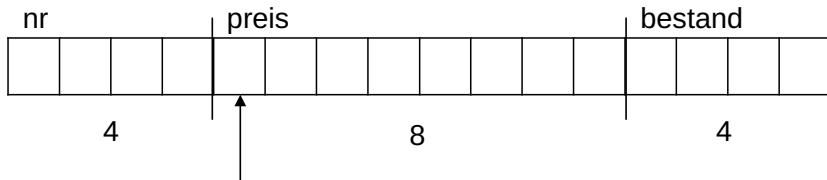
public static void main(String[] args) throws IOException {
    ArtikelManager manager = new ArtikelManager("artikel.dat");
    manager.addArtikel(new Artikel(4711, 140.99, 1000));
    manager.addArtikel(new Artikel(5011, 100., 450));
    manager.addArtikel(new Artikel(1112, 47.5, 1000));

    manager.list();
    manager.getArtikel(5011).ifPresentOrElse(System.out::println,
        () -> System.out.println("Artikel nicht gefunden."));
    manager.addBestand(5011, -100);
    manager.list();
    manager.close();
}
}
```

Ausgabe des Programms:

4711	140,99	1000
5011	100,00	450
1112	47,50	1000

```
Artikel[nr=5011, preis=100.0, bestand=450]
4711  140,99    1000
5011  100,00    350
1112  47,50    1000
```



Nachdem die Artikelnummer in `getArtikel` gefunden wurde, steht der Dateizeiger hier.

**Abbildung 24-2:** Datensatzstruktur

## 24.8 Datenkomprimierung

`java.util.zip.GZIPOutputStream` und `java.util.zip.GZIPInputStream` komprimieren bzw. expandieren einen Bytestrom.

Dabei wird das Kompressionsverfahren *gzip* (GNU zip) angewandt.<sup>1</sup>

Zur Demonstration der Komprimierung und Dekomprimierung benötigen wir eine genügend große Datei. `GenerateData` erzeugt Testdaten mit 1.000.000 zufälligen Zahlen zwischen 0 und 100.

```
package zip;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Random;

public class GenerateData {
    public static void main(String[] args) throws IOException {
        int n = 1_000_000;

        try (PrintWriter writer = new PrintWriter(new FileWriter("data"))) {
            Random r = new Random();
            for (int i = 0; i < n; i++) {
                writer.print(r.nextInt(100) + " ");
            }
        }
    }
}
```

---

<sup>1</sup> <https://de.wikipedia.org/wiki/Gzip>

## Komprimierung

```
package zip;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.zip.GZIPOutputStream;

public class Gzip {
    public static void main(String[] args) throws IOException {
        try (InputStream in = new FileInputStream("data");
             OutputStream out = new GZIPOutputStream(new FileOutputStream(
                     "data.gz"))) {

            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = in.read(buffer)) != -1) {
                out.write(buffer, 0, c);
            }

            out.flush();
        }
    }
}
```

Der *Kompressionsfaktor* gibt das Verhältnis der komprimierten Datenmenge zur ursprünglichen Datenmenge an. Der Kompressionsfaktor ist also der Wert, mit dem die unkomprimierte Datenmenge zu multiplizieren ist, um die komprimierte Datenmenge zu erhalten.

Die Originaldatei *data* enthält z. B. 2899648 Bytes, die komprimierte Datei *data.gz* 1048772 Bytes. Der Kompressionsfaktor beträgt also ca. 0,36.

## Dekomprimierung

```
package zip;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.zip.GZIPInputStream;

public class Gunzip {
    public static void main(String[] args) throws IOException {
        try (InputStream in = new GZIPInputStream(new FileInputStream("data.gz"));
             OutputStream out = new FileOutputStream("data2")) {

            byte[] buffer = new byte[8 * 1024];
            int c;
```

```
        while ((c = in.read(buffer)) != -1) {
            out.write(buffer, 0, c);
        }

        out.flush();
    }
}
```

Datenkomprimierung ist besonders interessant für die Übertragung von größerem Datenvolumen über das Netz:

Komprimieren beim Sender, Expandieren beim Empfänger.

### **ZIP-Dateien erstellen und entpacken**

Das *ZIP-Dateiformat* ist ein Format für komprimierte Dateien, wobei mehrere Dateien und Verzeichnisse zusammengefasst werden können.<sup>2</sup>

`java.util.zip.ZipOutputStream` ist ein `FilterOutputStream`, der eine Datei im ZIP-Dateiformat erstellen kann.

`java.util.zip.ZipEntry` repräsentiert einen Eintrag in der ZIP-Datei.

Das folgende Programm `Zip` erstellt eine Zip-Datei, wobei neben dem Ausgabedateinamen die zu komprimierenden Verzeichnisse als Aufrufparameter mitgegeben werden.

```
package zip;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;

public class Zip {
    public static void main(String[] args) throws IOException {
        try (ZipOutputStream out = new ZipOutputStream(new FileOutputStream(
                args[0]))) {
            for (int i = 1; i < args.length; i++) {
                zip(out, new File(args[i]));
            }
        }
    }
}
```

---

2 <https://de.wikipedia.org/wiki/ZIP-Dateiformat>

```
private static void zip(ZipOutputStream out, File file) throws IOException {
    if (file.isDirectory()) {
        String[] dirList = file.list();
        for (String name : dirList) {
            zip(out, new File(file.getPath(), name));
        }
    } else {
        System.out.println(file.getPath());
        try (InputStream in = new FileInputStream(file)) {
            ZipEntry entry = new ZipEntry(file.getPath());
            out.putNextEntry(entry);
            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = in.read(buffer)) != -1) {
                out.write(buffer, 0, c);
            }
        }
    }
}
```

Die Methode `zip` löst die Unterverzeichnisse bis auf Dateiebene rekursiv auf.

Aufrufbeispiel:

```
java -cp out/production/kap24 zip.Zip test.zip src/zip
```

`java.util.zip.ZipFile` wird genutzt, um die Einträge einer ZIP-Datei zu lesen.

```
package zip;

import java.io.IOException;
import java.util.zip.ZipFile;

public class ZipInfo {
    public static void main(String[] args) throws IOException {
        try (ZipFile zipFile = new ZipFile(args[0])) {
            zipFile.stream().forEach(entry ->
                System.out.printf(
                    "%s%n  size: %6d  compressed size: %6d  %tF %tT%n",
                    entry.getName(),
                    entry.getSize(),
                    entry.getCompressedSize(),
                    entry.getTime(),
                    entry.getTime())
            );
        }
    }
}
```

Aufrufbeispiel:

```
java -cp out/production/kap24 zip.ZipInfo test.zip
```

Ausgabe des Programms (Beispiel):

```
src/zip/Gunzip.java
  size:    594  compressed size:    289  2023-06-23 15:57:00
src/zip/Zip.java
  size:   1025  compressed size:   426  2023-06-23 15:57:00
src/zip/GenerateData.java
  size:    439  compressed size:   251  2023-06-23 15:57:00
src/zip/Unzip.java
  size:    950  compressed size:   404  2023-06-23 15:57:00
src/zip/ZipInfo.java
  size:    494  compressed size:   270  2023-06-23 15:57:00
src/zip/Gzip.java
  size:    593  compressed size:   288  2023-06-23 15:57:00
```

java.util.zip.ZipInputStream ist ein FilterInputStream zum Lesen von ZIP-Dateien.

Das Programm Unzip entpackt die ZIP-Datei in einem vorgegebenen Verzeichnis. Evtl. Unterverzeichnisse werden dabei erstellt.

```
package zip;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

public class Unzip {
    public static void main(String[] args) throws IOException {
        try (ZipInputStream in = new ZipInputStream(
            new FileInputStream(args[0]))) {
            ZipEntry entry;
            while ((entry = in.getNextEntry()) != null) {
                unzip(in, entry, args[1]);
            }
        }
    }

    private static void unzip(ZipInputStream in, ZipEntry entry, String destDir)
        throws IOException {
        if (entry.isDirectory())
            return;

        System.out.println(entry.getName());
        File file = new File(destDir, entry.getName());
        new File(file.getParent()).mkdirs();

        try (OutputStream out = new FileOutputStream(file)) {
            byte[] buffer = new byte[8 * 1024];
            int c;
```

```
        while ((c = in.read(buffer)) != -1) {
            out.write(buffer, 0, c);
        }
    }
}
```

Aufrufbeispiel:

```
java -cp out/production/kap24 zip.Unzip test.zip tmp
```

## 24.9 Aufgaben

### Aufgabe 1

Schreiben Sie ein Programm, das rekursiv die Unterverzeichnisse und die Dateien eines vorgegebenen Verzeichnisses auflistet.

*Lösung: Paket list\_dir*

### Aufgabe 2

Ausgehend von einem Startverzeichnis sollen alle Dateien mit dem Suffix `.java` in diesem Verzeichnis und in allen Unterverzeichnissen ausgegeben werden. Dabei soll nur der zum Startverzeichnis relative Pfad angezeigt werden. Verwenden Sie hierfür die `Path`-Methode:

```
Path relativize(Path other)
```

Zusätzlich sollen die Anzahl gefundener Dateien sowie die Gesamtgröße in Bytes ermittelt werden.

*Lösung: Paket search*

### Aufgabe 3

Nutzen Sie die Methoden `readAllBytes` und `write` für Binärdateien, um eine Datei zu kopieren.

*Lösung: Paket copy*

### Aufgabe 4

Eine Textdatei soll am Bildschirm ausgegeben werden. Nach jeweils 10 ausgegebenen Zeilen soll das Programm anhalten, bis die Eingabe-Taste gedrückt wird, und dann weiter ausgegeben.

*Lösung: Paket paging*

**Aufgabe 5**

Schreiben Sie ein Programm, das die Anzahl Zeichen (ohne `\n` und `\r`) und Zeilenwechsel einer Textdatei ermittelt. Die Textdatei kann über Eingabe-Umlenkung mit `System.in.read()` eingelesen werden.

*Lösung: Paket count*

**Aufgabe 6**

Schreiben Sie ein Programm, das eine Datei nach einem vorgegebenen Wort durchsucht und alle Zeilen, in denen das Wort gefunden wird, mit der Zeilenummer davor ausgibt.

*Lösung: Paket find*

**Aufgabe 7**

Schreiben Sie ein Programm, das eine Textdatei, die in einem bestimmten Zeichensatz codiert ist, in eine Textdatei mit einem anderen Zeichensatz konvertiert.

*Lösung: Paket charset*

**Aufgabe 8**

Fügen Sie der Klasse `Artikel` mit den Attributen `nr`, `preis` und `bestand` sowie den get/set-Methoden und der `toString`-Methode eine Methode hinzu, die die Werte der Instanzvariablen dieses Objekts (`this`) in einen `DataOutputStream` schreibt. Erstellen Sie einen Konstruktor, der die Instanzvariablenwerte für das neue Objekt aus einem `DataInputStream` liest.

*Lösung: Paket load\_store*

**Aufgabe 9**

Schreiben Sie ein Programm, das die mit dem Programm aus Kapitel 24.4 komprimierte Datei dekomprimiert.

*Lösung: Paket decompress*

**Aufgabe 10**

Wie kann das Programm aus Kapitel 24.4 so erweitert werden, dass das Sonderzeichen `@` in der Datei als normales Zeichen vorkommen darf?

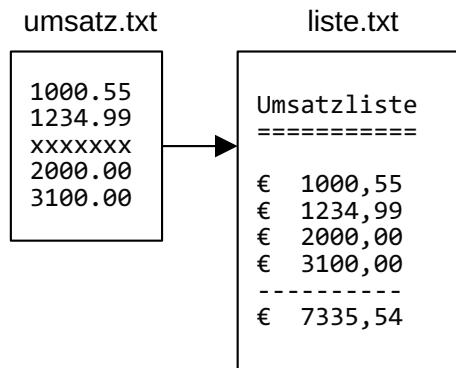
Schreiben Sie ein Komprimierungs- und Dekomprimierungsprogramm.

Tipp: Kommt `@` als normales Zeichen in der ursprünglichen Datei vor, so wird es verdoppelt.

*Lösung: Paket compression*

**Aufgabe 11**

Schreiben Sie ein Programm, das die folgende Eingabe in eine Ausgabeliste transformiert:



Fehlerhafte Zahlen sollen beim Einlesen ignoriert werden.

*Lösung: Paket umsatz*

**Aufgabe 12**

Schreiben Sie ein Programm, das die Zeichen einer Textdatei beim Lesen über einen Filter sofort in Großbuchstaben umwandelt.

*Lösung: Paket filter*

**Aufgabe 13**

Schreiben Sie zwei von `FilterOutputStream` bzw. `FilterInputStream` abgeleitete Klassen `EncryptOutputStream` und `DecryptInputStream`, die eine Datei verschlüsseln bzw. entschlüsseln. Hierzu sollen die Bytes der Klartextdatei bzw. der chiffrierten Datei mit Hilfe des exklusiven ODER-Operators (`^`) mit den Bytes eines vorgegebenen Schlüssels byteweise verknüpft werden. Ist die Schlüssellänge kleiner als die Länge der Datei, so soll der Schlüssel sukzessive wiederholt werden.

Beispiel zur Verschlüsselung mit einem Schlüssel der Länge 4:

Bytes der Datei:	B1	B2	B3	B4	B5	B6	B7	B8	...
	^	^	^	^	^	^	^	^	^
Schlüsselbytes:	k1	k2	k3	k4	k1	k2	k3	k4	...

*Lösung: Paket crypt*

**Aufgabe 14**

Schreiben Sie ein Programm, das den Inhalt einer Datei im Hexadezimalcode ausgibt. Jeweils 16 Bytes sollen in einer Zeile ausgegeben werden:

links im Hexadezimalcode, rechts als lesbare Zeichen (nicht druckbare Zeichen sind durch einen Punkt zu ersetzen).

Beispiel:

```
70 61 63 6b 61 67 65 20 68 65 78 3b 0d 0a 0d 0a  package hex;....  
69 6d 70 6f 72 74 20 6a 61 76 61 2e 69 6f 2e 42  import java.io.B  
75 66 66 65 72 65 64 49 6e 70 75 74 53 74 72 65  ufferedInputStre  
...
```

Tipp: Nutzen Sie die Integer-Methode `toHexString`.

Lösung: Paket hex

### Aufgabe 15

Entwickeln Sie eine Klasse, die das Interface `AutoCloseable` implementiert und testen Sie das *Automatic Resource Management* mit *try with resources*.

Lösung: Paket arm

### Aufgabe 16

Dateien können byteweise und blockweise (`int read(byte[] b)`) gelesen und geschrieben (`void write(byte[] b)`) werden. Schreiben Sie zwei Programme, die eine große Datei byteweise bzw. blockweise kopieren und ermitteln Sie dabei die Laufzeit in Millisekunden sowie den Durchsatz in Bytes/Sek. Testen Sie auch verschiedene Array-Größen.

Lösung: Paket runtime\_test

### Aufgabe 17

Schreiben Sie ein Programm, das bei jedem Aufruf einen als Aufrufparameter mitgegebenen `double`-Wert zusammen mit der aktuellen Systemzeit in Millisekunden (`long`-Wert) in eine `RandomAccess`-Datei einträgt. Ein weiteres Programm soll die `n` letzten Einträge am Bildschirm anzeigen. Die Millisekunden-Angabe soll für die Ausgabe in Datum und Uhrzeit umgewandelt werden.

Lösung: Paket tail

### Aufgabe 18

Eine Datei enthält englische Wörter mit ggf. mehreren Übersetzungen.

Beispiel:

```
key#Schlüssel#Taste  
slice#Scheibe#Schnitte#Stück  
value#Wert  
object#Objekt  
update#Aktualisierung
```

Das Programm VokabelTrainer lädt zu Beginn diese Daten in eine Map vom Typ `Map<String, List<String>>`. Schlüssel der Map ist die zu übersetzende Vokabel, Wert ist die Liste der möglichen Übersetzungen.

Die Steuerung des Programms erfolgt auf Kommandozeilenebene. Per Zufall wird ein Wort gewählt, das zu übersetzen ist. Bei richtiger Übersetzung wird die Vokabel aus der Map gelöscht, ansonsten wird wieder eine Vokabel abgefragt. Die Schleife läuft so lange, bis alle Vokabeln gelernt wurden, die Map also leer ist.

*Lösung: Paket `vokabeln`*



# 25 Serialisierung

*Serialisieren* ermöglicht das dauerhafte Speichern von kompletten Objekten mit allen Variablenwerten über die Laufzeit der Anwendung hinaus. Die gespeicherten Daten können dann später genutzt werden, um Objekte zur Laufzeit der Anwendung wiederherzustellen.

## Lernziele

In diesem Kapitel lernen Sie

- wie Objekte dauerhaft in Dateien gespeichert werden können,
- wie diese aus den gespeicherten Daten wieder rekonstruiert werden können,
- warum die Serialisierung von Objekten mit Vorsicht verwendet werden soll und
- wie das JSON-Format zur Serialisierung verwendet werden kann.

## 25.1 Serialisieren und Deserialisieren

Die beiden folgenden Programme schreiben bzw. lesen Objekte im Binärformat. Damit können Objekte zwischen Programmaufrufen in Dateien aufbewahrt werden (*Persistenz*). Der Zustand eines Objekts (d. h. die Werte der Instanzvariablen) wird in Bytes umgewandelt (*Serialisierung*).

Aus diesen Daten kann das Objekt wieder rekonstruiert werden (*Deserialisierung*).

### ObjectInputStream / ObjectOutputStream

`ObjectInputStream(InputStream in) throws IOException`  
erzeugt einen ObjectInputStream, der aus dem angegebenen InputStream liest.

`ObjectOutputStream(OutputStream out) throws IOException`  
erzeugt einen ObjectOutputStream, der in den angegebenen OutputStream schreibt.

#### Die ObjectOutputStream-Methode

`void writeObject(Object obj) throws IOException`

schreibt das Objekt obj in den Ausgabestrom. Die Werte aller Attribute, die nicht als static bzw. transient definiert sind, werden geschrieben. Dies gilt auch für ggf. in diesem Objekt referenzierte andere Objekte.

#### Die objectInputStream-Methode

`Object readObject() throws IOException, ClassNotFoundException`

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_25](https://doi.org/10.1007/978-3-658-43574-5_25).

liest ein Objekt aus dem Eingabestrom, das von der entsprechenden Methode der Klasse `ObjectOutputStream` geschrieben wurde.

In `ObjectInputStream` und `ObjectOutputStream` sind auch die Methoden aus `DataInputStream` bzw. `DataOutputStream` implementiert.

## Serializable

Die Klasse des zu schreibenden Objekts muss *serialisierbar* sein, d. h. sie (oder eine ihrer Basisklassen) muss das Interface `java.io.Serializable` implementieren. Dieses Interface besitzt keine Methoden, es dient lediglich der "Markierung".

Zudem muss die Klasse Zugriff auf den parameterlosen Konstruktor der ersten nicht serialisierbaren Superklasse haben. Ebenso müssen diese Voraussetzungen für die referenzierten Objekte erfüllt sein.

## Versionsnummer serialVersionUID

Die Klasse, deren Objekt serialisiert wurde, muss kompatibel sein zu der Klasse, für die das Objekt später wieder deserialisiert werden soll.

Würde man nach der Serialisierung beispielsweise das Attribut `adresse` der Klasse `Kunde` im folgenden Programm entfernen, so könnte das ursprüngliche Objekt nicht mehr rekonstruiert werden. Das Hinzufügen eines weiteren Attributs bereitet hingegen keine Probleme.

Bei der Serialisierung erzeugt das Laufzeitsystem eine *Versionsnummer* auf Basis verschiedener Aspekte der serialisierbaren Klasse.

Diese Versionsnummer wird genutzt, um bei der Deserialisierung feststellen zu können, ob die hierzu verwendete Klasse noch kompatibel zur ursprünglich benutzten Klasse ist.

Die Versionsnummer kann auch explizit mit einem eigenen Wert (hier z. B. 1) angegeben werden:

```
private static final long serialVersionUID = 1L;
```

Dies ist sinnvoll, wenn später Erweiterungen der Klasse stattfinden, die keinen Einfluss auf die Rekonstruierbarkeit eines Objekts haben (wie z. B. das oben geschilderte Hinzufügen eines neuen Attributs). In diesem Fall lässt man dann die alte Versionsnummer bestehen.

Im folgenden Beispiel implementiert die Klasse `Kunde` das Interface `Serializable`. Ihre Superklasse `Object` ist nicht serialisierbar, hat aber einen parameterlosen `public`-Konstruktor. Damit erfüllt `Kunde` die oben beschriebenen Voraussetzungen. Die Klasse `ArrayList` implementiert ebenfalls `Serializable`.

```
package serialization;

import java.io.Serializable;

public class Kunde implements Serializable {
    private static final long serialVersionUID = 1L;
    private final String name;
    private final String adresse;

    public Kunde(String name, String adresse) {
        this.name = name;
        this.adresse = adresse;
    }

    public String getName() {
        return name;
    }

    public String getAdresse() {
        return adresse;
    }
}

package serialization;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class SerializeTest {
    public static void main(String[] args) throws Exception {
        Kunde k1 = new Kunde("Meier, Hugo", "Hauptstr. 12, 40880 Ratingen");
        Kunde k2 = new Kunde("Schmitz, Otto", "Dorfstr. 5, 40880 Ratingen");

        List<Kunde> kunden = new ArrayList<>();
        kunden.add(k1);
        kunden.add(k2);
        kunden.add(k2);

        try (ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("kunden.ser"))) {

            out.writeObject(kunden);
            out.flush();
        }
    }
}

package serialization;

import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.ArrayList;
import java.util.List;

public class DeserializeTest {
```

```

public static void main(String[] args) throws Exception {
    try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(
        "kunden.ser"))) {

        List<Kunde> list = (ArrayList<Kunde>) in.readObject();
        for (Kunde k : list) {
            System.out.println(k.getName() + " " + k.getAdresse());
        }

        System.out.println("list.get(1) == list.get(2): "
            + (list.get(1) == list.get(2)));
    }
}
}

```

Das `List`-Objekt `kunden` speichert das `Kunde`-Objekt `k1` und zweimal *dasselbe* `Kunde`-Objekt `k2`.

Das Ergebnis der Deserialisierung ist ein *Objektgraph*, der zum Eingabograph äquivalent ist, d. h. auch im Ergebnis referenzieren die Listeneinträge mit der Nummer 1 und 2 *dasselbe* Objekt. Ein Objekt wird nur einmal serialisiert, auch wenn es mehrfach referenziert wird.

## 25.2 Sonderbehandlung bei Serialisierung

Die Klasse `Document` (siehe den folgenden Quellcode) enthält eine Klassenvariable, die den aktuellen Zählerstand enthält. Mit jedem Aufruf des Konstruktors wird der Zählerstand um eins erhöht und dieser Wert in der Instanzvariablen `id` gespeichert.

Durch Deserialisierung rekonstruierte `Document`-Instanzen enthalten die jeweiligen eindeutigen Ids. Ein dann im selben Programm neu erzeugtes Objekt enthält aber wiederum die Id mit der Nummer 1.

Der Grund hierfür ist:

Klassenvariablen werden bei der Serialisierung ignoriert.

In solchen Fällen hilft eine Sonderbehandlung durch Implementierung der folgenden Methoden in der betreffenden Klasse:

```

private void writeObject(ObjectOutputStream out) throws IOException
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException

```

Diese Methoden werden dann bei der Serialisierung bzw. Deserialisierung aufgerufen.

Die Standard zur Serialisierung und Deserialisierung kann weiterhin mittels der `ObjectOutputStream`-Methode

```
defaultWriteObject
```

und der `ObjectInputStream`-Methode

`defaultReadObject`

genutzt werden.

Das Beispiel zeigt, dass zunächst die Standard-Serialisierung aufgerufen und anschließend der Klassenvariablenwert geschrieben bzw. gelesen wird.

```
package extras;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Document implements Serializable {
    private static final long serialVersionUID = 1L;
    private static int nextId;
    private final int id;
    private final String content;

    public Document(String content) {
        id = ++nextId;
        this.content = content;
    }

    @Override
    public String toString() {
        return "Document{" +
            "id=" + id +
            ", content='" + content + '\'' +
            '}';
    }

    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        out.writeInt(nextId);
    }

    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException {
        in.defaultReadObject();
        nextId = in.readInt();
    }
}

package extras;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializeTest {
    public static void main(String[] args) throws IOException {
```

```

        try (ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("documents.ser"))) {
            out.writeObject(new Document("AAA"));
            out.writeObject(new Document("BBB"));
            out.flush();
        }
    }
}

package extras;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializeTest {
    public static void main(String[] args) throws IOException,
        ClassNotFoundException {
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(
            "documents.ser"))) {
            System.out.println(in.readObject());
            System.out.println(in.readObject());
        }
        System.out.println(new Document("CCC"));
    }
}

```

Ausgabe des Programms:

```

Document [id=1, content=AAA]
Document [id=2, content=BBB]
Document [id=3, content=CCC]

```

### Vorsicht: Sicherheitsproblem

Bei der Deserialisierung wird eine Byte-Folge in ein Objekt gewandelt. Es wird kein expliziter Konstruktor der Klasse verwendet und demnach werden im Konstruktor implementierte Prüfungen nicht durchgeführt.

Wird nun die Byte-Folge extern manipuliert, können so Objekte entstehen, die den geforderten Bedingungen nicht genügen.

Deshalb sollten keine Daten deserialisiert werden, die nicht vertrauenswürdig sind.

Es gibt eine Reihe alternativer, plattformübergreifender Mechanismen, um Objekte in strukturierte und auch lesbare Daten zu transformieren und umgekehrt, z. B. das textbasierte JSON-Format.<sup>1</sup>

---

<sup>1</sup> [https://de.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](https://de.wikipedia.org/wiki/JavaScript_Object_Notation)

## 25.3 Das Datenformat JSON

JSON (JavaScript Object Notation) ist ein Datenformat in Textform, das zur Übertragung und Speicherung strukturierter Daten eingesetzt wird.

JSON kennt die folgenden Elementtypen:

Nullwert (`null`), Boolescher Wert (`true` / `false`), Zahl, Zeichenkette, Array (`[...]`) und Objekt (`{...}`).

Ein Objekt enthält eine durch Kommas getrennte ungeordnete List von Schlüssel/Wert-Paaren. Der Schlüssel ist eine Zeichenkette, der Wert ein Element. Schlüssel und Wert sind durch einen Doppelpunkt getrennt.

Beispiel (Tageskarte eines Restaurants):

```
{
    "date": "27.01.2024",
    "items": [
        {
            "description": "Steak",
            "price": 12.99
        },
        {
            "description": "Salat",
            "price": 5.99
        }
    ]
}
```

JSON ist von Programmiersprachen unabhängig. Für Java gibt es mehrere Implementierungen für die Transformation zwischen Java-Objekten und JSON-Format (Serialisierung/Deserialisierung).

Wir verwenden in den folgenden Beispielen die *Gson-Bibliothek* von Google.<sup>2</sup> Sie befindet sich im Begleitmaterial zu diesem Kapitel und muss zur Compilierung und Ausführung dem Klassenpfad zugefügt werden.

Zur Demonstration verwenden wir die Klassen `Menu` und `MenuItem`.

`Menu` stellt die Tageskarte dar. Sie enthält das Tagesdatum (`LocalDate`) und eine Liste von `MenuItem`-Objekten. `MenuItem` enthält die Beschreibung und den Preis einer Speise.

```
package json;

public class MenuItem {
    private final String description;
    private final double price;
```

---

<sup>2</sup> <https://github.com/google/gson>

```
public MenuItem(String description, double price) {
    this.description = description;
    this.price = price;
}

@Override
public String toString() {
    return "MenuItem{" +
        "description='" + description + '\'' +
        ", price=" + price +
        '}';
}

package json;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class Menu {
    private final LocalDate date;
    private final List<MenuItem> items;

    public Menu(LocalDate date) {
        this.date = date;
        items = new ArrayList<>();
    }

    public void addItem(MenuItem item) {
        items.add(item);
    }

    @Override
    public String toString() {
        return "Menu{" +
            "date=" + date +
            ", items=" + items +
            '}';
    }
}
```

Das Programm `Serialize` erzeugt eine Tageskarte und transformiert den Objektgraph in einen String im JSON-Format.

```
package json;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

import java.time.LocalDate;

public class Serialize {
    public static void main(String[] args) {
```

```
Menu menu = new Menu(LocalDate.of(2024, 1, 27));
menu.addItem(new MenuItem("Steak", 12.99));
menu.addItem(new MenuItem("Salat", 5.99));

Gson gson = new GsonBuilder()
    .registerTypeAdapter(LocalDate.class, new LocalDateAdapater())
    .setPrettyPrinting().create();

String jsonString = gson.toJson(menu);
System.out.println(jsonString);
}
```

Da `LocalDate` kein einfacher Datentyp ist, benötigt `Gson` Hilfe, um ein `LocalDate`-Objekt in eine Zeichenkette mit gewünschtem Datumsformat (bzw. umgekehrt) zu transformieren. Zu diesem Zweck wird mit `registerTypeAdapter` ein Adapter registriert. Mit `setPrettyPrinting` wird der String mit Einrückungen aufbereitet. Die Ausgabe entspricht dem Beispiel zu Beginn dieses Unterkapitels.

Wird diese zusätzliche Konfiguration nicht benötigt, kann auch einfach

```
Gson gson = new Gson();
```

verwendet werden.

```
package json;

import com.google.gson.*;
import java.lang.reflect.Type;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class LocalDateAdapater implements JsonSerializer<LocalDate>,
    JsonDeserializer<LocalDate> {

    @Override
    public JsonElement serialize(
        LocalDate localDate, Type type,
        JsonSerializationContext jsonSerializationContext) {

        return new JsonPrimitive(localDate.format(
            DateTimeFormatter.ofPattern("dd.MM.yyyy")));
    }

    @Override
    public LocalDate deserialize(
        JsonElement jsonElement, Type type,
        JsonDeserializationContext jsonDeserializationContext)
        throws JsonParseException {

        return LocalDate.parse(jsonElement.getAsString(),
            DateTimeFormatter.ofPattern("dd.MM.yyyy"));
    }
}
```

Das Programm `Deserialize` transformiert ein Zeichenkette im JSON-Format in ein Menu-Objekt. Dazu muss der Zieltyp als Klassenliteral angegeben werden.

```
package json;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

import java.time.LocalDate;

public class Deserialize {
    public static void main(String[] args) {
        Gson gson = new GsonBuilder()
            .registerTypeAdapter(LocalDate.class, new LocalDateAdapater())
            .create();

        String jsonString = """
            {
                "date": "27.01.2024",
                "items": [
                    {
                        "description": "Steak",
                        "price": 12.99
                    },
                    {
                        "description": "Salat",
                        "price": 5.99
                    }
                ]
            }
            """;
        Menu menu = gson.fromJson(jsonString, Menu.class);
        System.out.println(menu);
    }
}
```

### Ausgabe des Programms:

```
Menu{date=2024-01-27, items=[MenuItem{description='Steak', price=12.99}, MenuItem{description='Salat', price=5.99}]} 
```

Die nächsten beiden Programme zeigen, wie eine Liste von Tageskarten transformiert werden kann.

Das Programm `SerializeList` enthält zwei Lösungsmöglichkeiten:

- `toJson` erzeugt einen String, der mit `Files.writeString` in eine Datei geschrieben wird,
- `toJson` schreibt selbst mit Hilfe eines `Writer` in eine Datei.

```
package json;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.time.LocalDate;
import java.util.List;

public class SerializeList {
    public static void main(String[] args) throws IOException {
        Menu menu1 = new Menu(LocalDate.of(2024, 1, 27));
        menu1.addItem(new MenuItem("Steak", 12.99));
        menu1.addItem(new MenuItem("Salat", 5.99));

        Menu menu2 = new Menu(LocalDate.of(2024, 1, 28));
        menu2.addItem(new MenuItem("Spaghetti", 7.99));
        menu2.addItem(new MenuItem("Steak", 12.99));
        menu2.addItem(new MenuItem("Salat", 5.99));

        List<Menu> menus = List.of(menu1, menu2);

        Gson gson = new GsonBuilder()
            .registerTypeAdapter(LocalDate.class, new LocalDateAdapater())
            .setPrettyPrinting().create();

        serialize1(menus, gson, "menus.json");
        // serialize2(menus, gson, "menus.json");
    }

    private static void serialize1(List<Menu> menus, Gson gson, String filename)
        throws IOException {
        String jsonString = gson.toJson(menus);
        Files.writeString(Paths.get(filename), jsonString);
    }

    private static void serialize2(List<Menu> menus, Gson gson, String filename)
        throws IOException {
        try (Writer writer = new FileWriter(filename)) {
            gson.toJson(menus, writer);
        }
    }
}
```

*menus.json* enthält:

```
[  
  {  
    "date": "27.01.2024",  
    "items": [  
      {  
        "description": "Steak",  
        "price": 12.99  
      },  
      {  
        "description": "Salat",  
        "price": 5.99  
      }  
    ]  
  }]
```

```
{
    "description": "Salat",
    "price": 5.99
}
],
},
{
    "date": "28.01.2024",
    "items": [
        {
            "description": "Spaghetti",
            "price": 7.99
        },
        {
            "description": "Steak",
            "price": 12.99
        },
        {
            "description": "Salat",
            "price": 5.99
        }
    ]
}
```

Das Programm `DeserializeList` enthält mehrere Lösungsmöglichkeiten.

Zum einen unterscheiden sich die Varianten in der Art der Eingabe: `fromJson` liest aus einem String oder mit Hilfe eines Reader direkt aus der Datei.

Zum andern ist das Zielobjekt vom Typ `Menu[]` oder vom Typ `List<Menu>`.

- Im ersten Fall wird `fromJson` mit dem Klassenliteral `Menu[].class` aufgerufen.
- Im zweiten Fall wird die Typinformation `Menu` in `List<Menu>` mit Hilfe von `TypeToken` zur Laufzeit zur Verfügung gestellt.

Da der Konstruktor `TypeToken()` `protected` ist, muss eine Subklasse dieser Klasse erstellt werden, die dann das Abrufen der Typinformationen zur Laufzeit ermöglicht:

```
Type type = new TypeToken<List<Menu>>() {}.getType();

package json;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.reflect.TypeToken;

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;
import java.lang.reflect.Type;
import java.nio.file.Files;
```

```
import java.nio.file.Paths;
import java.time.LocalDate;
import java.util.List;

public class DeserializeList {
    public static void main(String[] args) throws IOException {
        Gson gson = new GsonBuilder()
            .registerTypeAdapter(LocalDate.class, new LocalDateAdapater())
            .create();

        Menu[] menus = deserialize1("menus.json", gson);
        // Menu[] menus = deserialize2("menus.json", gson);
        // List<Menu> menus = deserialize3("menus.json", gson);
        // List<Menu> menus = deserialize4("menus.json", gson);

        for (Menu menu : menus) {
            System.out.println(menu);
        }
    }

    // Array als Root-Objekt
    private static Menu[] deserialize1(String filename, Gson gson)
        throws IOException {

        String jsonString = Files.readString(Paths.get(filename));
        return gson.fromJson(jsonString, Menu[].class);
    }

    private static Menu[] deserialize2(String filename, Gson gson)
        throws IOException {

        try (Reader reader = new FileReader(filename)) {
            return gson.fromJson(reader, Menu[].class);
        }
    }

    // List als Root-Objekt
    private static List<Menu> deserialize3(String filename, Gson gson)
        throws IOException {

        String jsonString = Files.readString(Paths.get(filename));
        Type type = new TypeToken<List<Menu>>() {}.getType();
        return gson.fromJson(jsonString, type);
    }

    private static List<Menu> deserialize4(String filename, Gson gson)
        throws IOException {

        try (Reader reader = new FileReader(filename)) {
            Type type = new TypeToken<List<Menu>>() {}.getType();
            return gson.fromJson(reader, type);
        }
    }
}
```

## 25.4 Aufgaben

### Aufgabe 1

Fügen Sie der Klasse `Artikel` mit den Attributen `nr`, `preis` und `bestand` sowie den `get/set`-Methoden und der `toString`-Methode die beiden folgenden Methoden hinzu:

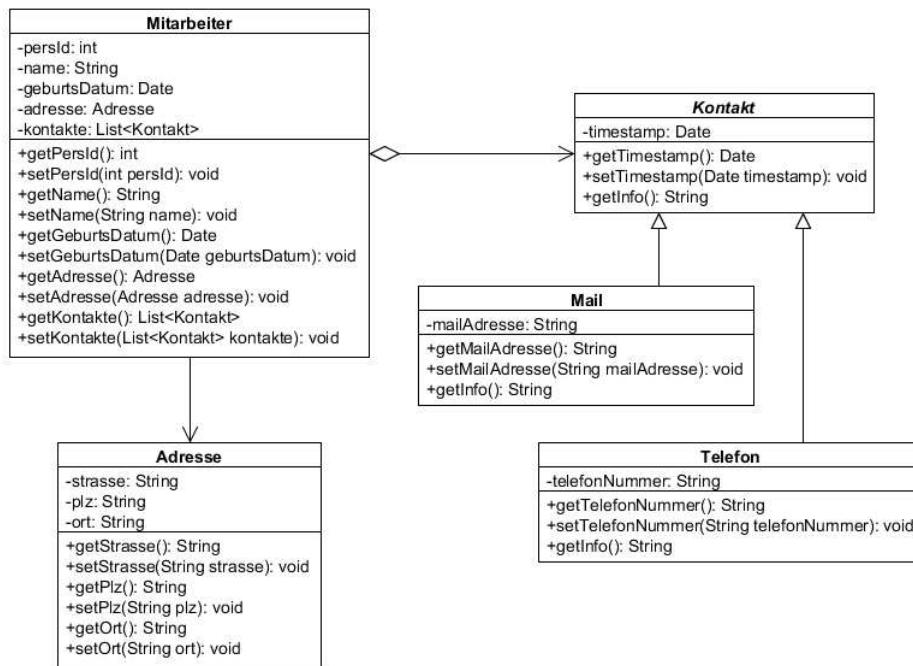
```
public void store(String name) throws IOException
public static Artikel load(String name) throws IOException,
    ClassNotFoundException
```

`store` soll das Objekt `this` serialisieren und in die Datei `name` schreiben. `load` soll das Objekt durch Deserialisierung wieder rekonstruieren.

Lösung: Paket `load_store`

### Aufgabe 2

Entwickeln Sie gemäß Vorgabe die Klassen des nachfolgenden Klassendiagramms.



Ein Mitarbeiter hat eine Adresse und ggf. mehrere Kontakte (Mailadressen, Telefonnummern). Die Klasse `Kontakt` ist abstrakt und besitzt die abstrakte Methode `getInfo`, die in den Subklassen `Mail` und `Telefon` implementiert ist.

Schreiben Sie ein Programm, das mehrere Objekte vom Typ `Mitarbeiter` serialisiert und ein weiteres Programm, das diese `Mitarbeiter`-Objekte durch Deserialisierung rekonstruiert.

*Lösung: Paket serialize*

### Aufgabe 3

Schreiben Sie ein Programm, das eine bestimmte Anzahl einzelner Objekte einer von Ihnen selbst bestimmten Klasse serialisiert. Schreiben Sie ein weiteres Programm, das diese Objekte wieder durch Deserialisierung rekonstruiert. Um das Ende der Folge von Objekten zu erkennen, gibt es verschiedene Möglichkeiten:

1. Nutzung der Ausnahme `EOFException`, die das Ende des Datenstroms signalisiert.
2. Markierung des Endes durch Einfügen des Wertes `null` nach dem letzten Objekt bei der Serialisierung und beim Einlesen Erkennen des Endes am Wert `null`.
3. Schreiben einer Zahl, die die Anzahl der folgenden Objekte angibt, als ersten Wert bei der Serialisierung. Bei der Deserialisierung wird dann die Leseschleife so oft durchlaufen, wie der Zählerwert angibt.

Testen Sie diese verschiedenen Lösungsalternativen.

*Lösung: Paket eof*

### Aufgabe 4

Implementieren Sie eine Methode, die eine tiefe Kopie (Klon) eines Objekts durch Duplizierung des Objektgraphen mittels Serialisierung/Deserialisierung liefert:

```
public static <T extends Serializable> T clone(T obj)
```

Verwenden Sie `ByteArrayOutputStream` bzw. `ByteArrayInputStream`.

*Lösung: Paket deep\_copy*

### Aufgabe 5

Die Klasse `java.beans.XMLEncoder` bietet eine Alternative zur Serialisierung mittels `ObjectOutputStream`. Objekte werden extern in Form einer XML-Struktur dargestellt.

Konstruktor:

```
XMLEncoder(OutputStream out)
```

```
void writeObject(Object obj)
```

erzeugt die XML-Darstellung des Objekts `obj`.

```
void close()
```

leert den Puffer und schließt die Ausgabedatei.

Umgekehrt können Objekte mit Hilfe der Klasse `java.beans.XMLDecoder` aus der XML-Darstellung rekonstruiert werden.

Konstruktor:

```
XMLDecoder(InputStream in)
```

```
Object readObject()
```

liest das nächste Objekt aus dem Eingabestrom. `ArrayIndexOutOfBoundsException` wird ausgelöst, wenn der Eingabestrom keine Objekte mehr enthält.

```
void close()
```

schließt die Eingabedatei.

Klassen, deren Objekte auf diese Weise serialisiert werden sollen, müssen den Standardkonstruktor besitzen und get/set-Methoden für alle ihre Attribute haben.

Testen Sie diese Methoden für Objekte einer Klasse Ihrer Wahl.

*Lösung: Paket xml*

## Aufgabe 6

Eine Map enthält zum `Integer`-Schlüssel Artikelnummer ein `Artikel`-Objekt als Wert. Erzeugen Sie mehrere Map-Einträge. Serialisieren Sie die Map in eine Datei im JSON-Format und schreiben Sie ein Programm, das die Map als Java-Objekt aus dieser Datei rekonstruiert.

Nutzen Sie

```
Type type = new TypeToken<Map<Integer, Artikel>>() {}.getType();
```

ähnlich wie in Kapitel 25.3.

*Lösung: Paket json*



## 26 Nebenläufige Programmierung

*Nebenläufigkeit* ist die Eigenschaft eines Programms, zwei oder mehrere Aktivitäten so ausführen, dass alle voranschreiten.

Hat ein Rechner mehrere CPUs bzw. Rechnerkerne, können die Aktivitäten *parallel* (gleichzeitig) ausgeführt werden (*parallele Ausführung*).

In Java wird Nebenläufigkeit mit sogenannten *Threads* erzielt.

### Lernziele

In diesem Kapitel lernen Sie

- wie Threads erzeugt, gestartet und beendet werden können,
- welche Zustände Threads einnehmen können (*Lebenszyklus*),
- welche Probleme durch *konkurrierende Zugriffe* mehrerer Threads auf dieselben Daten auftreten können,
- wie solche Zugriffe durch *Synchronisierung* kontrolliert werden können,
- wie Threads koordiniert an einer gemeinsamen Aufgabe arbeiten können,
- wie der Einsatz spezieller Klassen und Methoden die Programmierung *asynchroner Abläufe* vereinfachen kann und
- wie Prozesse des Betriebssystems kontrolliert werden können (*Process-API*).

### 26.1 Threads erzeugen, starten und beenden

Moderne Betriebssysteme können mehrere Programme quasi gleichzeitig (*Multi-tasking*) oder tatsächlich gleichzeitig (bei Mehrkernprozessoren und Mehrprozessorsystemen) ausführen.

Die sequentielle Ausführung der Anweisungen eines Programms durch den Prozessor stellt einen *Prozess* dar, für den ein eigener Speicherbereich reserviert ist und der vom Betriebssystem verwaltet, gestartet und angehalten wird.

Ein *Thread* (Ausführungsader, Handlungsstrang) ist ein einzelner in sich geschlossener Steuerfluss innerhalb eines Prozesses.

Threads in Java werden auf Betriebssystem-Threads abgebildet, welche dann vom Scheduler des Betriebssystems einzelnen Rechnerkernen zugeordnet werden.

Zur Ausführung eines Java-Programms startet die JVM (*Java Virtual Machine*) den sogenannten `main`-Thread. Dieser führt die `main`-Methode aus. Mehrere neue Threads können nun vom Programm selbst gestartet werden (*Multithreading*).

Diese Threads laufen dann alle *quasi parallel* ab, besitzen jeweils einen eigenen Zustand mit Befehlszähler, Stack usw., arbeiten aber im Gegensatz zu Prozessen auf demselben Speicherbereich im Arbeitsspeicher.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_26](https://doi.org/10.1007/978-3-658-43574-5_26).

Wenn ein Programm mehr Threads erzeugt als Prozessoren bzw. Rechnerkerne vorhanden sind, können nicht alle Threads zur selben Zeit laufen. In diesem Fall werden die Aktivitäten auf die knappen CPU-Ressourcen zeitlich verteilt. Die Gleichzeitigkeit wird dann dadurch simuliert, dass die Aktivitäten abwechselnd in schneller Folge Rechenzeit zugewiesen bekommen.

Multithreading verbessert die Bedienbarkeit von grafischen Dialoganwendungen, insbesondere, wenn sie mit Animationen verbunden sind. Sie ermöglichen die Ausführung zeitintensiver Operationen im Hintergrund.

Im Rahmen von Client/Server-Anwendungen müssen Serverprogramme Anfragen verschiedener Clients gleichzeitig bearbeiten können. Zur Abarbeitung dieser Anfragen können Threads vom Serverprogramm gestartet werden.

Threads werden durch Objekte der Klasse `java.lang.Thread` repräsentiert.

Die Klasse `Thread` implementiert das Interface `java.lang.Runnable`, das die Methode

```
void run()
```

vereinbart.

Diese Methode bestimmt den als Thread auszuführenden Code. Die Standardimplementierung von `run` in der Klasse `Thread` tut gar nichts (leerer Anweisungsblock).

### Threads erzeugen und starten

Um einen Thread zu erzeugen, gibt es grundsätzlich zwei Möglichkeiten:

- Man leitet von der Klasse `Thread` ab und überschreibt die `run`-Methode oder
- man erzeugt eine Klasse, die das Interface `Runnable` implementiert. Mit einem Objekt dieser Klasse als Konstruktor-Argument erzeugt man dann ein `Thread`-Objekt.

Diese zweite Möglichkeit muss immer dann genutzt werden, wenn die Klasse selbst bereits Subklasse einer anderen Klasse ist.

In beiden Fällen muss die Methode `run` implementiert werden. Sie enthält den Programmcode des Threads.

Die `Thread`-Methode

```
void start()
```

sorgt für das Starten des Threads.

Die weiteren Anweisungen des Aufrufers der Methode `start` laufen nun quasi gleichzeitig (nebenläufig) zu den Anweisungen des gestarteten Threads.

Wenn der Thread mit der Ausführung der `run`-Methode fertig ist, terminiert er. `start` darf nur einmal für einen Thread aufgerufen werden. Bei zu wiederholender Ausführung muss eine neue Thread-Instanz erzeugt werden.

Konstruktoren der Klasse `Thread` sind:

```
Thread()  
Thread(String name)
```

`name` ersetzt den Standardnamen des Threads durch einen eigenen Namen.

```
Thread(Runnable runObj)  
Thread(Runnable runObj, String name)
```

`runObj` ist eine Referenz auf das Objekt, dessen `run`-Methode benutzt werden soll. `name` ersetzt den Standardnamen des Threads durch einen eigenen Namen.

Die folgenden Programme zeigen die beiden Möglichkeiten, Threads zu erzeugen. Die `Thread`-Methode `sleep` legt hier den aktuellen Thread für eine Sekunde schlafen. Die verwendeten Methoden `sleep`, `getName` und `currentThread` sind weiter unten erklärt.

```
package erzeugen;  
  
public class Test1 extends Thread {  
    @Override  
    public void run() {  
        String name = getName();  
  
        for (int i = 0; i < 3; i++) {  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException ignored) {}  
  
            System.out.println(name + ": " + i);  
        }  
  
        System.out.println(name + ": Ich bin fertig!");  
    }  
  
    public static void main(String[] args) {  
        Test1 t1 = new Test1();  
        Test1 t2 = new Test1();  
        t1.start();  
        t2.start();  
        System.out.println("Habe zwei Threads gestartet.");  
    }  
}
```

```

package erzeugen;

public class Test2 implements Runnable {
    @Override
    public void run() {
        String name = Thread.currentThread().getName();

        for (int i = 0; i < 3; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ignored) {
            }

            System.out.println(name + ": " + i);
        }

        System.out.println(name + ": Ich bin fertig!");
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new Test2());
        Thread t2 = new Thread(new Test2());
        t1.start();
        t2.start();
        System.out.println("Habe zwei Threads gestartet.");
    }
}

```

### Ausgabe der Programme (Beispiel):

```

Habe zwei Threads gestartet.
Thread-0: 0
Thread-1: 0
Thread-0: 1
Thread-1: 1
Thread-1: 2
Thread-0: 2
Thread-0: Ich bin fertig!
Thread-1: Ich bin fertig!

```

Die Methode `println` von `System.out` ist so implementiert, dass sich die einzelnen Ausgaben bei mehreren Threads nicht gegenseitig überschreiben. Die Reihenfolge kann aber durchaus bei jedem Programmlauf eine andere sein.

Ein Programm endet erst dann, wenn alle Threads beendet sind.

Ein *Daemon Thread* ist ein Thread, der das Programmende nicht verhindert. Das Programm endet, sobald die einzigen Threads, die noch laufen, Daemon-Threads sind.

### Die `Thread`-Methode

```
final void setDaemon(boolean on)
```

markiert den Thread als Daemon-Thread, falls `on` den Wert `true` hat.

Zur Unterscheidung nennt man einen "normalen" Thread auch *User Thread* oder *Vordergrund-Thread* und einen Daemon Thread auch *Hintergrund-Thread*.

```
package erzeugen;

public class DaemonTest extends Thread {
    @Override
    public void run() {
        System.out.println("Beginn run");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException ignored) {
        }
        System.out.println("Ende run");
    }

    public static void main(String[] args) throws InterruptedException {
        DaemonTest p = new DaemonTest();
        p.setDaemon(true);
        p.start();
        Thread.sleep(2000);
        System.out.println("Ende main");
    }
}
```

Ausgabe des Programms mit `p.setDaemon(true)`:

```
Beginn run
Ende main
```

Ausgabe des Programms mit `p.setDaemon(false)`:

```
Beginn run
Ende main
Ende run
```

## Einige Thread-Methoden

`String getName()`

liefert den Namen des Threads.

`void setName(String name)`

gibt dem Thread den neuen Namen `name`.

`boolean isAlive()`

liefert `true`, wenn der Thread gestartet wurde und noch nicht beendet ist, sonst `false`.

`static Thread currentThread()`

liefert eine Referenz auf den aktuellen Thread, der die Methode ausführt, in deren Rumpf der Aufruf von `currentThread` steht.

```
void interrupt()
```

sendet ein *Unterbrechungssignal* an den Thread, für den die Methode aufgerufen wurde.

Dieser Thread befindet sich dann im Status "unterbrochen". Ist der Thread blockiert (z. B. durch den Aufruf von `sleep`, `join` oder `wait`), so wird eine `java.lang.InterruptedException` ausgelöst und der Status "unterbrochen" gelöscht.

```
boolean isInterrupted()
```

liefert `true`, falls der Thread den Status "unterbrochen" hat, sonst `false`.

```
static boolean interrupted()
```

prüft, ob der aktuelle Thread unterbrochen wurde und setzt den Status "unterbrochen" auf `false` zurück.

```
static void sleep(long millis) throws InterruptedException
```

hält die Ausführung des aktuellen Threads für `millis` Millisekunden an. Er verbraucht in dieser Phase keine Rechenzeit. Ist der Thread beim Aufruf der Methode im Status "unterbrochen" oder erhält er während der Wartezeit diesen Status, so wird die Ausnahme `java.lang.InterruptedException` ausgelöst und der Status gelöscht.

Sind mehrere Threads vorhanden, so sollten diejenigen mit höherer Priorität vor denen mit niedrigerer Priorität ausgeführt werden.

```
void setPriority(int p)
```

setzt die Priorität des Threads.

`Thread.NORM_PRIORITY` (= 5) ist der Normalwert. `p` muss zwischen den Thread-Konstanten `MIN_PRIORITY` (= 1) und `MAX_PRIORITY` (= 10) liegen.

```
int getPriority()
```

liefert die Priorität des Threads.

```
void join() throws InterruptedException
```

wartet, bis der Thread, für den `join` aufgerufen wurde, beendet ist.

Bei Unterbrechung während der Wartezeit wird eine `InterruptedException` ausgelöst und der Status "unterbrochen" wird gelöscht.

```
void join(long millis) throws InterruptedException
```

wartet maximal `millis` Millisekunden auf das Beenden des Threads.

```
static void yield()
```

lässt den aktuellen Thread kurzzeitig pausieren, um anderen Threads die Gelegenheit zur Ausführung zu geben. Allerdings gibt es für das Umschalten auf einen anderen Thread keine Garantie.

### Threads beenden

Wie kann man Threads zwangsweise beenden?

Das Programm `Test1` zeigt, wie ein Thread, der im Sekundenrhythmus die aktuelle Uhrzeit anzeigt, mit `interrupt` beendet werden kann.

```
package beenden;

import java.io.IOException;

public class Test1 implements Runnable {
    public void run() {
        while (true) {
            System.out.println(new java.util.Date());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                break;
            }
        }
    }

    public static void main(String[] args) throws IOException {
        Thread t = new Thread(new Test1());
        t.start();
        System.in.read(); // blockiert bis RETURN
        t.interrupt();
    }
}
```

Ein Variante:

```
package beenden;

import java.io.IOException;

public class Test2 implements Runnable {
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println(new java.util.Date());
                Thread.sleep(1000);
            }
        } catch (InterruptedException ignored) {
        }
    }

    public static void main(String[] args) throws IOException {
        Thread t = new Thread(new Test2());
        t.start();
        System.in.read(); // blockiert bis RETURN
        t.interrupt();
    }
}
```

Test3 zeigt eine weitere Möglichkeit zur Beendigung der while-Schleife innerhalb der run-Methode.

```
package beenden;

import java.io.IOException;

public class Test3 implements Runnable {
    private volatile boolean isStopped;

    public void start() {
        new Thread(this).start();
    }

    public void stop() {
        isStopped = true;
    }

    public void run() {
        while (!isStopped) {
            System.out.println(new java.util.Date());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ignored) {
            }
        }
    }

    public static void main(String[] args) throws IOException {
        Test3 test = new Test3();
        test.start();
        System.in.read();
        test.stop();
    }
}
```

Die Methode `start` erzeugt einen Thread und startet diesen. Die Methode `stop` setzt die Variable `isStopped` auf `true`.

Die `while`-Schleife läuft so lange, wie `isStopped` den Wert `false` hat. Ist diese Bedingung nicht mehr erfüllt, wird die Schleife verlassen. Hierbei ist wichtig, dass die Variable `isStopped` `volatile` ist.

### **volatile**

Jeder Thread kann für die Variablen, mit denen er arbeitet, seine eigene Kopie erstellen. Wenn nun bei einer gemeinsamen Variablen ein Thread den Wert in seiner Kopie ändert, haben andere Threads in ihrer Kopie immer noch den alten Wert. Das ist in Systemen mit mehreren Prozessoren bzw. Rechnerkernen durchaus möglich.

Der `main`-Thread im obigen Programm verändert über die Methode `stop` die Instanzvariable `isStopped`. Der gestartete Thread liest `isStopped`, um die Schleifenbedingung zu prüfen.

Wird von zwei oder mehreren Threads auf *ein und dieselbe* Variable zugegriffen, wobei mindestens ein Thread den Wert verändert (wie in diesem Beispiel), dann sollte diese Variable mit dem Modifizierer `volatile` versehen werden. Der Compiler verzichtet dann auf gewisse Code-Optimierungen und stellt sicher, dass das Lesen einer `volatile`-Variablen immer den zuletzt geschriebenen Wert zurückgibt.

Die Kennzeichnung eines Attributs mit `volatile` ist also wichtig, wenn immer der aktuelle Wert benutzt werden soll. Bei einer Referenzvariablen wird jedoch nicht garantiert, dass der "Inhalt" des referenzierten Objekts aktuell ist.

### Auf das Ende eines Threads warten

Mit der Thread-Methode `join` kann ein Thread auf das Ende eines anderen Threads warten.

```
package beenden;

public class JoinTest extends Thread {
    @Override
    public void run() {
        System.out.println(getName() + ": Es geht los!");

        try {
            Thread.sleep(5000);
        } catch (InterruptedException ignored) {

        }

        System.out.println(getName() + ": Erledigt!");
    }

    public static void main(String[] args) throws InterruptedException {
        JoinTest t = new JoinTest();
        t.start();
        System.out.println("Auf das Ende warten ...");
        t.join();
        System.out.println("Endlich!");
    }
}
```

Ausgabe des Programms:

```
Auf das Ende warten ...
Thread-0: Es geht los!
Thread-0: Erledigt!
Endlich!
```

## 26.2 Lebenszyklus eines Threads

Abbildung 26-1 zeigt die verschiedenen Zustände, die ein Thread einnehmen kann.

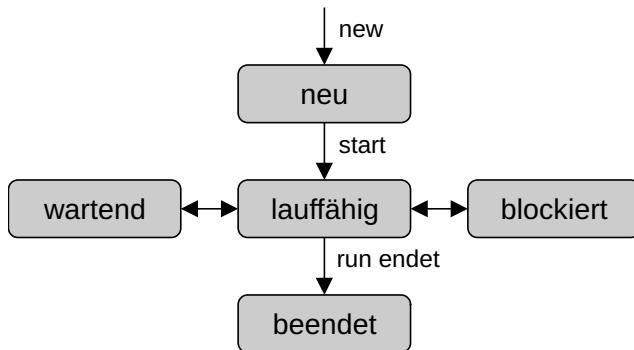


Abbildung 26-1: Thread-Zustände

Ein *lauffähiger* Thread kann aktiv oder inaktiv sein, je nachdem, ob er auf Ressourcen wie Prozessorzuteilung u. Ä. wartet oder nicht.

Ein Thread kann in den Zustand *wartend* z. B. durch Aufrufe der Methoden `sleep`, `join` und `wait` gelangen.

Ein Thread ist *blockiert*, wenn er aufgrund einer Sperre auf den Eintritt in einen synchronisierten Block (Methode) warten muss.

## 26.3 Synchronisation

Beim Multithreading können Probleme durch den gleichzeitigen Zugriff auf gemeinsame Objekte und Variablen auftreten (*Wettlaufbedingung, race condition*).

Es darf in der Regel nicht vorkommen, dass ein Thread bereits aus einem Objekt liest, während ein anderer Thread noch Daten desselben Objekts ändert.

Um falsche Ergebnisse zu vermeiden, müssen die Zugriffe bei Änderung gemeinsamer Daten kontrolliert werden.

Hierzu stehen Sperrmechanismen zur Verfügung. Jedes Java-Objekt besitzt eine implizite Sperre, auf die man Zugriff durch Verwendung des Schlüsselworts `synchronized` erhält.

Kritische Programmteile, die zu einer Zeit nur von einem Thread durchlaufen werden dürfen, müssen bei der Programmentwicklung erkannt und dann geschützt werden.

Java bietet hierzu zwei Möglichkeiten:

- Schützen einer Methode oder
- Schützen eines Blocks innerhalb einer Methode.

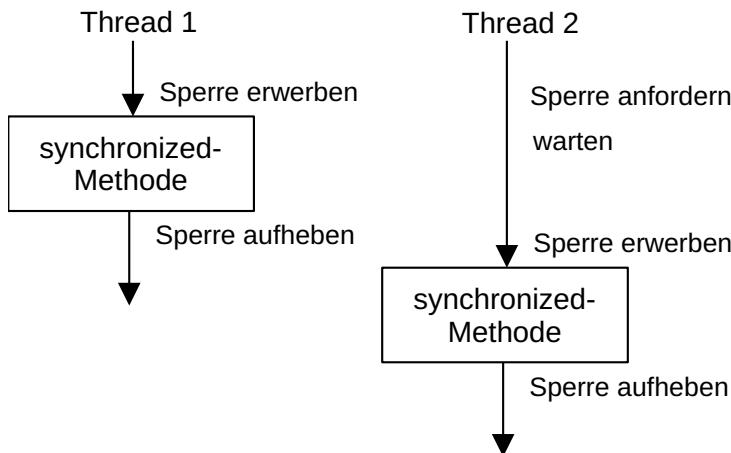
Wir beschäftigen uns zunächst mit der ersten Möglichkeit.

### **synchronized für eine Methode**

Durch den Modifizierer `synchronized` kann die Ausführung von *Instanzmethoden*, die *für dasselbe Objekt* aufgerufen werden, so kontrolliert werden, dass *nur eine* der so gekennzeichneten Methoden zu einer Zeit von einem Thread ausgeführt werden kann.

Hier muss ein Thread also erst die Sperre dieses Objekts erwerben, bevor er die Methode ausführen darf. Nach Ausführung der Methode wird die Sperre aufgehoben. Ist die Sperre bereits belegt, wird der Thread blockiert. Blockierte Threads werden intern in eine *Warteliste* eingetragen. Ist die Sperre wieder frei, kann sie von einem blockierten Thread erworben werden.

Ein Thread, der die Sperre erworben hat, kann sie erneut anfordern (Aufruf einer weiteren `synchronized`-Methode für dasselbe Objekt), ohne warten zu müssen. Man bezeichnet diese Eigenschaft einer Sperre als *reentrant* (wiederbetretbar).



**Abbildung 26-2:** Sperrkonzept

Sperrobjekt bei `synchronized`-Klassenmethoden (`static`-Methoden) ist das `Class`-Objekt der jeweiligen Klasse (`Klassename.class`), sodass zu jedem Zeitpunkt *nur immer eine* synchronisierte Klassenmethode dieser Klasse ausgeführt werden kann.

## Wichtige Hinweise

- `synchronized` hat für das Überladen (Overloading) oder Überschreiben (Overriding) von Methoden keine Bedeutung.
- Zwei Threads können dieselbe `synchronized`-Methode gleichzeitig ausführen, wenn sie *für verschiedene Objekte* aufgerufen wurde.
- Eine `synchronized`-Methode und eine Methode, die nicht mit `synchronized` gekennzeichnet ist, können zur selben Zeit für dasselbe Objekt ausgeführt werden.

## Zugriff mehrerer Threads auf gemeinsame Daten

Wird von zwei oder mehreren Threads auf *ein und dieselbe* Variable zugegriffen, wobei mindestens ein Thread den Wert verändert, dann sollten alle Zugriffe auf diese Variable in `synchronized`-Methoden oder -Blöcken erfolgen.

Synchronisation stellt sicher, dass die Variable vor dem Betreten der synchronisierten Methode bzw. des synchronisierten Blocks aus dem zentralen Speicher geladen und beim Verlassen wieder zurückgeschrieben wird, sodass sie in jedem Thread korrekte Werte enthält (analoges Verhalten wie beim Zugriff auf `volatile`-Attribute).

Das folgende Programm demonstriert die Wirkung von `synchronized`-Methoden.

Die Klasse `Work` enthält die synchronisierte Instanzmethode `doWorkA` und die synchronisierte Klassenmethode `doWorkB`. Die `main`-Methode startet zunächst zwei Threads unmittelbar hintereinander, die `doWorkA` für dasselbe Objekt aufrufen. Nach Beendigung der beiden Threads wird nochmals ein Thread wie vorher gestartet, anschließend wird die Klassenmethode `doWorkB` zweimal aufgerufen.

```
package sync;

public class Work {
    public synchronized void doWorkA(String name) {
        System.out.println(name + ": Beginn A");
        try {
            Thread.sleep(4000);
        } catch (InterruptedException ignored) {
        }
        System.out.println(name + ": Ende A");
    }

    public synchronized static void doWorkB(String name) {
        System.out.println(name + ": Beginn B");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ignored) {
        }
        System.out.println(name + ": Ende B");
    }
}
```

```
package sync;

public class SyncTest extends Thread {
    private final Work w;

    public SyncTest(Work w) {
        this.w = w;
    }

    @Override
    public void run() {
        w.doWorkA(getName());
    }

    public static void main(String[] args) throws InterruptedException {
        Work w = new Work();
        SyncTest t1 = new SyncTest(w);
        SyncTest t2 = new SyncTest(w);
        SyncTest t3 = new SyncTest(w);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        t3.start();
        Work.doWorkB(Thread.currentThread().getName());
        Work.doWorkB(Thread.currentThread().getName());
    }
}
```

Ausgabe des Programms (Beispiel):

```
Thread-0: Beginn A
Thread-0: Ende A
Thread-1: Beginn A
Thread-1: Ende A
Thread-2: Beginn A
main: Beginn B
main: Ende B
main: Beginn B
Thread-2: Ende A
main: Ende B
```

Das Beispiel zeigt auch, dass eine synchronisierte Instanzmethode und eine synchronisierte Klassenmethode "gleichzeitig" laufen können.

### **synchronized für einen Block**

Um kritische Codebereiche zu schützen können auch Anweisungsblöcke mit dem Schlüsselwort **synchronized** eingeleitet werden:

```
synchronized (obj) { ... }
```

Hier muss die Sperre für das Objekt `obj` erworben werden, um die Anweisungen im Block ausführen zu können. Beim Austritt aus dem Block wird die Sperre für das Objekt aufgehoben.

Der `synchronized`-Block bietet im Vergleich zur `synchronized`-Methode zwei Vorteile:

- Es kann ein Codeabschnitt synchronisiert werden, der nur einen Teil des Methodenrumpfs ausmacht. Damit wird die exklusive Sperre für eine im Vergleich zur Ausführungszeitdauer der gesamten Methode kürzere Zeitdauer beansprucht.
- Als Sperrobjecte können neben `this` auch andere Objekte gewählt werden.

Die `synchronized`-Methode

```
public void synchronized xyz() { ... }
```

ist äquivalent zu:

```
public void xyz() {
    synchronized(this) { ... }
}
```

Wenn in einer Klasse zwei Methoden auf jeweils unterschiedliche Datenbereiche synchronisiert zugreifen sollen, die Methoden also *unabhängig voneinander* sind und beide zur selben Zeit ohne Konflikt aufgerufen werden können, bieten sich `synchronized`-Blöcke mit *verschiedenen* Sperrobjecten an:

```
private final Object lock1 = new Object();
private final Object lock2 = new Object();

public void m1() {
    synchronized (lock1) {
        // Zugriff auf Datenbereich A
    }
}

public void m2() {
    synchronized (lock2) {
        // Zugriff auf Datenbereich B
    }
}
```

Wären beide Methoden `synchronized`, hätten diese dieselbe Sperre `this` und *nur eine* Methode könnte demzufolge *zu einer Zeit* ausgeführt werden.

Referenzen auf Sperrobjecte sollten mit `final` vor versehentlicher Änderung geschützt werden.

## Deadlock

Das folgende Programm zeigt eine *Deadlock*-Situation:

Zwei Threads warten auf die Aufhebung der Sperre durch den jeweils anderen Thread und bleiben so für immer blockiert.

```
package sync;

public class DeadlockTest {
    private final Object a = new Object();
    private final Object b = new Object();

    public void f(String name) {
        System.out.println(name + " will a sperren");
        synchronized (a) {
            System.out.println(name + " a gesperrt");
            try {
                Thread.sleep(500);
            } catch (InterruptedException ignored) {
            }
            System.out.println(name + " will b sperren");
            synchronized (b) {
                System.out.println(name + " b gesperrt");
            }
        }
    }

    public void g(String name) {
        System.out.println(name + " will b sperren");
        synchronized (b) {
            System.out.println(name + " b gesperrt");
            System.out.println(name + " will a sperren");
            synchronized (a) {
                System.out.println(name + " a gesperrt");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        final DeadlockTest test = new DeadlockTest();
        Thread t1 = new Thread(() -> test.f("Thread 1"));
        Thread t2 = new Thread(() -> test.g("Thread 2"));

        t1.start();
        Thread.sleep(200);
        t2.start();
    }
}
```

Thread t1 sperrt zunächst Objekt a und versucht dann, Objekt b zu sperren, während Thread t2 zuerst Objekt b sperrt und anschließend versucht, Objekt a zu sperren. Thread.sleep(500) wurde eingefügt, damit t2 das Objekt b sperren kann, bevor dies t1 gelingt.

Ausgabe des Programms:

```
Thread 1 will a sperren
Thread 1 a gesperrt
Thread 2 will b sperren
Thread 2 b gesperrt
Thread 2 will a sperren
Thread 1 will b sperren
```

Das Programm bleibt hängen und muss "gewaltsam" abgebrochen werden.

## 26.4 Fallbeispiel Summenberechnung

Die Programme dieses Kapitels zeigen anhand einer einfachen Summenberechnung, wie eine Anwendung parallelisiert werden kann, um die Laufzeit des Programms zu reduzieren. Allerdings kommt es darauf an, abhängig von der Aufgabenstellung richtig im Sinne der Anforderung zu synchronisieren.<sup>1</sup>

In `SimpleSum` berechnet der `main`-Thread alleine die Summe.

```
package sum;

public class SimpleSum {
    private long sum;

    public static void main(String[] args) {
        new SimpleSum().sum();
    }

    public void sum() {
        long startTime = System.currentTimeMillis();
        for (long n = 0; n < 1_000_000_000; n++) {
            sum += n;
        }
        long time = System.currentTimeMillis() - startTime;
        System.out.println("Sum: " + sum);
        System.out.println("Time: " + time);
    }
}
```

Ausgabe des Programms:

```
Sum: 499999999500000000
Time: 1498
```

Die konkrete Laufzeit hängt natürlich von der Leistungsfähigkeit des verwendeten Prozessors ab.

---

<sup>1</sup> Dieses Beispiel wurde angeregt durch den Fachartikel "Datensynchronisation zwischen Threads" von Christian Robert im Java Spektrum 4/2013. Die ermittelten Zeiten sind natürlich sehr stark abhängig von der Prozessorarchitektur des Laufzeitsystems.

In `ThreadedSum` teilen sich vier Threads die Aufgabe der Berechnung. Sie addieren jeweils Zahlen aus den Bereichen 0 ... 249999999, 250000000 ... 499999999, 500000000 ... 749999999 und 750000000 ... 999999999.

```
package sum;

public class ThreadedSum {
    private long sum;
    private final Thread[] threads = new Thread[4];

    public static void main(String[] args) {
        new ThreadedSum().sum();
    }

    public void sum() {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 4; i++) {
            final long start = i * (1_000_000_000 / 4);
            final long end = (i + 1) * (1_000_000_000 / 4);
            threads[i] = new Thread(() -> {
                for (long n = start; n < end; n++) {
                    sum += n;
                }
            });
            threads[i].start();
        }

        for (int i = 0; i < 4; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException ignored) {
            }
        }
    }

    long time = System.currentTimeMillis() - startTime;
    System.out.println("Sum: " + sum);
    System.out.println("Time: " + time);
}
}
```

Ausgabe des Programms:

```
Sum: 93772553944641919
Time: 681
```

Die Laufzeit ist deutlich geringer, allerdings ist das Ergebnis falsch. Jeder Thread arbeitet mit einer eigenen Kopie von `sum`, die zu undefinierten Zeitpunkten auch zurückgeschrieben werden kann. Damit kann keine korrekte Summenberechnung stattfinden.

In `SynchronizedSum` erfolgt jeder Berechnungsschritt synchronisiert. Das Ergebnis ist korrekt, die Laufzeit hat aber erheblich zugenommen.

```

package sum;

public class SynchronizedSum {
    private long sum;
    private final Thread[] threads = new Thread[4];

    public static void main(String[] args) {
        new SynchronizedSum().sum();
    }

    private void sum() {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 4; i++) {
            final long start = i * (1_000_000_000 / 4);
            final long end = (i + 1) * (1_000_000_000 / 4);
            threads[i] = new Thread(() -> {
                for (long n = start; n < end; n++) {
                    synchronized (SynchronizedSum.this) {
                        sum += n;
                    }
                }
            });
            threads[i].start();
        }

        for (int i = 0; i < 4; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException ignored) {
            }
        }
    }

    long time = System.currentTimeMillis() - startTime;
    System.out.println("Sum: " + sum);
    System.out.println("Time: " + time);
}
}

```

### Ausgabe des Programms:

```

Sum: 499999999500000000
Time: 71855

```

In `ConcurrentSum` nutzt jeder Thread eine eigene Ergebnisvariable. Die berechnete Teilsumme wird jeweils am Ende der Thread-Ausführung synchronisiert auf-addiert.

```

package sum;

public class ConcurrentSum {
    private long sum;
    private final Thread[] threads = new Thread[4];

    public static void main(String[] args) {
        new ConcurrentSum().sum();
    }
}

```

```
private void sum() {
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < 4; i++) {
        final long start = i * (1_000_000_000 / 4);
        final long end = (i + 1) * (1_000_000_000 / 4);
        threads[i] = new Thread(() -> {
            long threadSum = 0;
            for (long n = start; n < end; n++) {
                threadSum += n;
            }
            synchronized (ConcurrentSum.this) {
                sum += threadSum;
            }
        });
        threads[i].start();
    }

    for (int i = 0; i < 4; i++) {
        try {
            threads[i].join();
        } catch (InterruptedException ignored) {
        }
    }

    long time = System.currentTimeMillis() - startTime;
    System.out.println("Sum: " + sum);
    System.out.println("Time: " + time);
}
}
```

Ausgabe des Programms:

Sum: 499999999500000000

Time: 170

Die Laufzeit ist erheblich geringer als im ersten Beispiel (`simpleSum`) und das Ergebnis stimmt.

## 26.5 Thread-Sicherheit

Methoden sind *Thread-sicher*, wenn sie für die parallele Nutzung durch mehrere Threads geeignet sind.

Die gleichzeitige Änderung von Attributwerten eines Objekts durch solche Methoden führt nicht zu einem inkonsistenten Zustand des Objekts. Ändert ein Thread das Objekt, "sehen" andere Threads dieses Objekt immer in einem gültigen Zustand.

Im folgenden Programmbeispiel zahlen zwei Threads jeweils 10.000 mal 1 Euro auf ein Bankkonto ein. Am Ende sollten 20.000 Euro eingezahlt worden sein. Das ist aber nicht der Fall.

```
package safe.bad;

public class Account {
    private int balance;

    public void deposit(int amount) {
        balance += amount;
    }

    public static void main(String[] args) throws InterruptedException {
        Account konto = new Account();

        Runnable runnable = () -> {
            for (int i = 0; i < 10000; i++) {
                konto.deposit(1);
            }
        };
        Thread t1 = new Thread(runnable);
        Thread t2 = new Thread(runnable);
        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Eingezahlt wurden insgesamt " + konto.balance
                           + " Euro.");
    }
}
```

Ausgabe (Beispiel):

```
Eingezahlt wurden insgesamt 10634 Euro.
```

Die Ursache für das falsche Ergebnis liegt offensichtlich in der Anweisung

```
balance += amount;
```

der Methode deposit.

Es kann zu folgender Bearbeitungsreihenfolge kommen:

1. Thread 1 liest den aktuellen Wert von balance, z. B. 100.
2. Thread 2 liest den aktuellen Wert von balance: 100.
3. Thread 1 addiert 1 zum gelesenen Wert und speichert 101.
4. Thread 2 addiert 1 zum gelesenen Wert und speichert 101.

Die erste Änderung wird überschrieben und ist damit verloren.

Die Methode deposit ist also nicht Thread-sicher.

Synchronisation löst hier das Problem:

```
public synchronized void deposit(int amount) {  
    balance += amount;  
}
```

Ausgabe:

Eingezahlt wurden insgesamt 20000 Euro.

### Thread-sicheres Singleton

Das in Kapitel 12 vorgestellte Entwurfsmuster *Singleton* stellt sicher, dass von einer Klasse genau eine Instanz existiert.

Eine Thread-sichere Implementierung mit verzögerter Initialisierung zeigt das folgende Beispiel:

```
package singleton;  
  
public class MySingleton1 {  
    private static MySingleton1 instance;  
    ...  
  
    private MySingleton1() {  
        ...  
    }  
  
    public static synchronized MySingleton1 getInstance() {  
        if (instance == null) {  
            instance = new MySingleton1();  
        }  
        return instance;  
    }  
  
    // weitere Methoden  
}
```

Die Zugriffszeit kann verbessert werden, indem nur bei der Erzeugung der Instanz synchronisiert wird:

```
package singleton;  
  
public class MySingleton2 {  
    private static volatile MySingleton2 instance;  
    ...  
  
    private MySingleton2() {  
        ...  
    }
```

```

public static MySingleton2 getInstance() {
    if (instance == null) {
        synchronized (MySingleton2.class) {
            if (instance == null) {
                instance = new MySingleton2();
            }
        }
    }
    return instance;
}

// weitere Methoden
}

```

Die Abfrage `instance == null` muss hier zweimal erfolgen, da nach der ersten Abfrage ein konkurrierender Thread bereits die Instanz erzeugt haben könnte. Das Attribut `instance` muss `volatile` sein, weil auch außerhalb des `synchronized`-Blocks auf `instance` zugegriffen wird.

Die in Kapitel 12 vorgestellte `enum`-Variante eines Singltons ist ebenfalls Thread-sicher.

### Thread-sichere Container

Die `public`-Methoden der Klassen `Vector` und `Hashtable` sind Thread-sicher.

Die Methoden der Klassen des *Collections Frameworks* sind nicht Thread-sicher.

Klassenmethoden von `java.util.Collections` liefern synchronisierte Objekte:

```

static <T> List<T> synchronizedList(List<T> list)
static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
static <T> Set<T> synchronizedSet(Set<T> s)

```

Deren Methoden sind Thread-sicher.

Das Durchlaufen eines Containers muss allerdings geschützt werden, da nebenläufige Veränderungen während einer Iteration möglich sind:

```

List<String> syncList = Collections.synchronizedList(new ArrayList<>());
synchronized (syncList) {
    for (int i = 0; i < syncList.size(); i++) {
        // tu etwas
    }
}

```

Das Paket `java.util.concurrent` enthält Thread-sichere Container, wie z. B. `CopyOnWriteArrayList` und `ConcurrentHashMap`.

Die List-Implementierung `CopyOnWriteArrayList` ist für lesende Zugriffe optimiert. Schreibzugriffe finden auf einer neuen Kopie des zugrunde liegenden Arrays statt. Das alte Array wird nach dem Schreiben durch das neue ersetzt. Andere Threads können stets ohne Konflikt lesend zugreifen. Beim Durchlaufen des Containers mit einem Iterator (z. B. *foreach-Schleife*) sind nur lesende Zugriffe möglich.

Die Map-Implementierung `ConcurrentHashMap` ermöglicht paralleles Lesen und Schreiben. Der Hashbereich ist in Segmente aufgeteilt, die jeweils separat durch Sperren geschützt sind. Schreibzugriffe auf verschiedene Bereiche können so nebenläufig stattfinden.

Das folgende Programm zeigt den Einsatz einer `ConcurrentHashMap`, um für jeden Thread einen eigenen User-spezifischen Kontext zu speichern.

```
package collections;

import java.util.HashMap;
import java.util.Map;

public class UserRepository {
    private final Map<Integer, String> map = new HashMap<>();

    public UserRepository() {
        map.put(1, "Hugo");
        map.put(2, "Emil");
    }

    public String getUserId(int id) {
        return map.get(id);
    }
}

package collections;

public class Context {
    private final String name;

    public Context(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Context{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

```

package collections;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class SharedMap implements Runnable {
    private static final Map<Integer, Context> ctxMap = new ConcurrentHashMap<>();
    private static final UserRepository userRepository = new UserRepository();
    private final int id;

    public SharedMap(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        String user = userRepository.getUserById(id);
        ctxMap.put(id, new Context(user));
        System.out.println("Context für " + id + " ist " + ctxMap.get(id));
    }

    public static void main(String[] args) throws InterruptedException {
        SharedMap firstUser = new SharedMap(1);
        SharedMap secondUser = new SharedMap(2);

        new Thread(firstUser).start();
        new Thread(secondUser).start();
    }
}

```

## Thread-lokale Daten

Die Klasse `ThreadLocal` besitzt die Methoden `set` und `get`, um ein Objekt zu speichern bzw. zu lesen. Das Besondere hierbei ist, dass jeder Thread mit `set` sein eigenes Objekt speichern kann und auch nur dieses wieder mit `get` erhält.

So kann das obige Beispiel nun wie folgt implementiert werden:

```

package collections;

public class ContextPerThread implements Runnable {
    private static final ThreadLocal<Context> ctx = new ThreadLocal<>();
    private static final UserRepository userRepository = new UserRepository();
    private final int id;

    public ContextPerThread(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        String user = userRepository.getUserById(id);
        ctx.set(new Context(user));
        System.out.println("Context für " + id + " ist " + ctx.get());
    }
}

```

```
public static void main(String[] args) throws InterruptedException {  
    ContextPerThread firstUser = new ContextPerThread(1);  
    ContextPerThread secondUser = new ContextPerThread(2);  
  
    new Thread(firstUser).start();  
    new Thread(secondUser).start();  
}  
}
```

## 26.6 Kommunikation zwischen Threads

Das Programm Lager (siehe weiter unten) demonstriert ein sogenanntes *Producer/Consumer*-Problem. Ein Thread arbeitet als *Produzent* und erzeugt in zufälligen Abständen Zufallszahlen. Ein anderer Thread, der *Konsument*, "verbraucht" diese Zahlen in zufälligen Abständen.

Die Zahlen werden in einem gemeinsamen "Datenlager" zwischengespeichert. Dort kann nur eine Zahl zu einer Zeit gespeichert sein.

Die unabhängig voneinander laufenden Threads müssen sich synchronisieren. Der Konsument muss warten, bis eine neue Zahl gespeichert wurde, und der Produzent muss warten, bis die vorher erzeugte und abgelegte Zahl gelesen wurde.

### **wait, notify**

Die erforderliche Synchronisation erfolgt über die Methoden `wait` und `notify` bzw. `notifyAll` der Klasse `Object`.

Diese Methoden dürfen *nur innerhalb* von `synchronized`-Methoden oder -Blöcken auftreten und werden *für das gesperrte Objekt* aufgerufen.

#### `void wait() throws InterruptedException`

hebt die Sperre für dieses Objekt auf und der aufrufende Thread wartet so lange, bis er durch den Aufruf der Methode `notify` oder `notifyAll` durch einen anderen Thread, der in den Besitz der Sperre für dieses Objekt gelangt ist, aufgeweckt wird.

`wait` wird erst abgeschlossen, wenn der aufgeweckte Thread wiederum dieses Objekt für sich sperren konnte.

Bei Unterbrechung des Threads wird eine `InterruptedException` ausgelöst und der Status "unterbrochen" wird gelöscht.

#### `void notify()`

weckt einen von evtl. mehreren Threads auf, der für dasselbe Objekt, für das diese Methode aufgerufen wurde, `wait` aufgerufen hat.

#### `void notifyAll()`

weckt alle wartenden Threads auf, die sich nun um eine Sperre bewerben.

Bevor das oben erwähnte Producer/Consumer-Problem behandelt wird, soll mit dem folgenden Programm der Einsatz von `wait` und `notify` anhand eines einfachen Beispiels gezeigt werden.

Beide Threads arbeiten mit demselben Sperrobject `lock` und rufen hierfür `wait` bzw. `notify` auf. Der Thread `Waiter` wartet solange bis die Variable `wait` den Wert `false` hat. Nachdem er durch `notify` aufgeweckt wurde und die Sperre für sich wieder erlangen konnte (also frühestens nachdem der Thread `Notifier` den `synchronized`-Block verlassen hat), überprüft er, ob `wait` noch `false` ist, und setzt dann seine Arbeit fort.

```
package wait_notify;

import java.io.IOException;

public class WaitNotifyDemo {
    private static final Object lock = new Object();
    private static boolean wait = true;

    public static void main(String[] args) {
        Thread waiter = new Thread(() -> {
            synchronized (lock) {
                while (wait) {
                    System.out.println("Waiter: wait == " + wait);
                    System.out.println("Waiter wartet");
                    try {
                        lock.wait();
                    } catch (InterruptedException ignored) {
                    }
                }
                System.out.println("Waiter: wait == " + wait);
            }
        });
        Thread notifier = new Thread(() -> {
            System.out.println("Notifier: Weiter mit RETURN");
            try {
                System.in.read();
            } catch (IOException ignored) {
            }
        });
    }
}
```

```

        synchronized (lock) {
            wait = false;
            lock.notify();
            System.out.println("Notifier: notify aufgerufen");
        }
    });

waiter.start();
notifier.start();
}
}

```

Ausgabe des Programms:

Notifier: Weiter mit RETURN

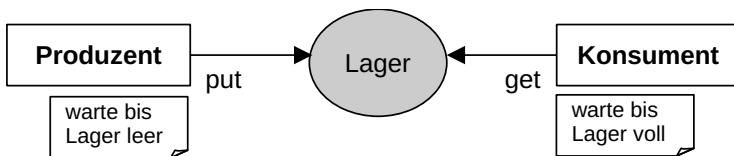
Waiter: wait == true

Waiter wartet

Notifier: notify aufgerufen

Waiter: wait == false

### Producer/Consumer



**Abbildung 26-3:** Kommunikation zwischen Threads

Die Methoden `get` und `put` der Klasse `Lager` enthalten ein typisches Muster für die Anwendung von `wait` und `notify`: Der Thread wartet, bis eine bestimmte Bedingung erfüllt ist. Die Bedingungsprüfung erfolgt in einer Schleife.

```

package wait_notify;

public class Lager {
    private int value;
    private boolean full;

    public synchronized void put(int value) throws InterruptedException {
        while (full) { // solange das Lager voll ist
            wait();
        }
    }
}

```

```
// Lager ist leer und wird gefüllt
this.value = value;
full = true;
notify();
}

public synchronized int get() throws InterruptedException {
    while (!full) { // solange das Lager leer ist
        wait();
    }

    // Lager ist voll und wird geleert
    full = false;
    notify();
    return value;
}

public static void main(String[] args) {
    Lager lager = new Lager();
    Produzent p = new Produzent(lager);
    Konsument k = new Konsument(lager);
    p.start();
    k.start();
}
}
```

```
package wait_notify;

import java.util.Random;

public class Produzent extends Thread {
    private static final int MAX = 5;
    private final Lager lager;

    public Produzent(Lager lager) {
        this.lager = lager;
    }

    @Override
    public void run() {
        try {
            Random random = new Random();
            for (int i = 0; i < MAX; i++) {
                System.out.println(i + " wird produziert");
                Thread.sleep(1000 + random.nextInt(4000));
                lager.put(i);
                System.out.println(i + " auf Lager");
            }
            lager.put(-1); // Produktion gestoppt
        } catch (InterruptedException ignored) {
        }
    }
}
```

```
package wait_notify;

import java.util.Random;

public class Konsument extends Thread {
    private final Lager lager;

    public Konsument(Lager lager) {
        this.lager = lager;
    }

    public void run() {
        try {
            Random random = new Random();
            int i = 0;
            while (true) {
                int value = lager.get();
                if (value == -1)
                    break;
                System.out.println("\t" + value + " aus Lager entfernt");
                Thread.sleep(1000 + random.nextInt(1000));
                System.out.println("\t" + value + " konsumiert");
            }
        } catch (InterruptedException ignored) {
        }
    }
}
```

Ausgabebeispiel:

```
0 wird produziert
0 auf Lager
1 wird produziert
    0 aus Lager entfernt
    0 konsumiert
1 auf Lager
    1 aus Lager entfernt
2 wird produziert
    1 konsumiert
2 auf Lager
    2 aus Lager entfernt
3 wird produziert
    2 konsumiert
3 auf Lager
    3 aus Lager entfernt
4 wird produziert
    3 konsumiert
4 auf Lager
    4 aus Lager entfernt
    4 konsumiert
```

### wait immer in einer while-Schleife aufrufen

wait sollte immer in einer while-Schleife aufgerufen werden:

```
while (Bedingung)
    wait();
```

Es könnte Situationen geben, in denen nach dem Aufwecken die Wartebedingung noch weiterhin erfüllt ist. `if` anstelle von `while` würde dann ggf. zu einem logischen Fehler führen.

Es könnte in Szenarien mit mehreren Threads nach dem Aufwecken des Threads ein dritter Thread aktiv werden und die Wartebedingung wiederherstellen, bevor der aufgeweckte Thread die Sperre wieder erwerben konnte (siehe Beschreibung zu `wait`).

### notify vs. notifyAll

Das obige Beispiel kann auf mehrere Produzenten und Konsumenten ausgedehnt werden. Dann sollte aber `notifyAll` statt `notify` verwendet werden.

Würde beispielsweise der Produzent-Thread `notify` aufrufen, so würde nur ein einziger Thread aufgeweckt, und das könnte dann unglücklicherweise ein anderer Produzent-Thread sein, also ein "falscher Thread", der wieder warten muss, da das Lager noch voll ist. Alle Threads wären nun blockiert.

Es existieren verschiedene Implementierungen des Interface

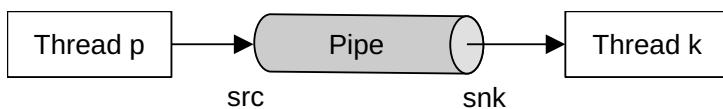
`java.util.concurrent.BlockingQueue<T>`

zur Lösung derartiger Probleme.<sup>2</sup>

### Pipes

`PipedInputStream` und `PipedOutputStream` bieten Methoden, um Daten zwischen zwei Threads über eine sogenannte *Pipe* auszutauschen.

Eine Pipe verbindet einen `PipedOutputStream` `src` mit einem `PipedInputStream` `snk`. Ein Thread schreibt Daten in `src` (Quelle), die von einem anderen Thread aus `snk` (Senke) gelesen werden.



**Abbildung 26-4:** Eine Pipe

Schreib- und Lesevorgänge sind über einen internen Puffer entkoppelt.

Ein `PipedOutputStream` muss mit einem `PipedInputStream` verbunden werden.

---

<sup>2</sup> Siehe auch Aufgabe 14 dieses Kapitels.

Dies geschieht entweder mit der `PipedOutputStream`-Methode

```
void connect(PipedInputStream snk)
```

oder mit der `PipedInputStream`-Methode

```
void connect(PipedOutputStream src)
```

Das folgende Programm demonstriert den *Pipe-Mechanismus*. Ein Produzent erzeugt Zahlen, die der Konsument sukzessive aufaddiert.

```
package pipe;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class Pipe {
    private final DataOutputStream out;
    private final DataInputStream in;

    public Pipe() throws IOException {
        PipedOutputStream src = new PipedOutputStream();
        PipedInputStream snk = new PipedInputStream();
        src.connect(snk);
        out = new DataOutputStream(src);
        in = new DataInputStream(snk);
    }

    public void put(int value) throws IOException {
        out.writeInt(value);
    }

    public int get() throws IOException {
        return in.readInt();
    }

    public void close() throws IOException {
        out.close();
    }

    public static void main(String[] args) throws IOException {
        Pipe pipe = new Pipe();
        Produzent p = new Produzent(pipe);
        Konsument k = new Konsument(pipe);
        p.start();
        k.start();
    }
}

package pipe;

import java.io.IOException;
import java.util.Random;
```

```
public class Produzent extends Thread {  
    private final Pipe pipe;  
  
    public Produzent(Pipe p) {  
        this.pipe = p;  
    }  
  
    @Override  
    public void run() {  
        Random random = new Random();  
        try {  
            for (int i = 0; i < 5; i++) {  
                System.out.println("Produzent: " + i);  
                pipe.put(i);  
                Thread.sleep(1000 + random.nextInt(2000));  
            }  
            pipe.close();  
        } catch (InterruptedException ignored) {}  
        catch (IOException e) {  
            System.err.println(e);  
        }  
    }  
}  
  
package pipe;  
  
import java.io.EOFException;  
import java.io.IOException;  
  
public class Konsument extends Thread {  
    private final Pipe pipe;  
  
    public Konsument(Pipe p) {  
        this.pipe = p;  
    }  
  
    @Override  
    public void run() {  
        try {  
            int summe = 0;  
            while (true) {  
                int value = pipe.get();  
                summe += value;  
                System.out.println("\tKonsument: " + summe);  
                Thread.sleep(3000);  
            }  
        } catch (InterruptedException | EOFException ignored) {}  
        catch (IOException e) {  
            System.err.println(e.getMessage());  
        }  
    }  
}
```

### Threads anhalten und fortsetzen

Die Methoden `wait` und `notify` können auch benutzt werden, um die Ausführung eines Threads vorübergehend anzuhalten und wieder fortzusetzen.

```
package suspend;

import java.io.IOException;
import java.time.LocalTime;

public class StopAndGo extends Thread {
    private boolean stop;

    @Override
    public void run() {
        try {
            while (true) {
                synchronized (this) {
                    while (stop)
                        wait();
                }
                System.out.println(LocalTime.now());
                Thread.sleep(3000);
            }
        } catch (InterruptedException ignored) {
        }
    }

    public synchronized void doSuspend() {
        stop = true;
    }

    public synchronized void doResume() {
        stop = false;
        notify();
    }

    public static void main(String[] args) throws IOException {
        StopAndGo stopAndGo = new StopAndGo();
        stopAndGo.start();

        boolean run = true;
        while (run) {
            int b = System.in.read();
            switch (b) {
                case 's' -> stopAndGo.doSuspend(); // stoppen
                case 'e' -> { // beenden
                    stopAndGo.interrupt();
                    run = false;
                }
                case 'w' -> stopAndGo.doResume(); // weiter
            }
        }
    }
}
```

## Auf Ereignisse warten

Oft kommt es vor, dass Threads auf ein bestimmtes Ereignis warten müssen, bevor sie ihre Arbeit weiter ausführen können.

Die Klasse `java.util.concurrent.CountDownLatch` repräsentiert einen Zähler, der mit einer positiven ganzen Zahl initialisiert werden muss. Die Methode `countDown` erniedrigt den Zähler um eins. Mit dem Aufruf der Methode `await` können Threads darauf warten bis der Zähler 0 erreicht hat. Eine `CountDownLatch`-Instanz kann nur einmal verwendet werden.

Das folgende Programm zeigt den Umgang mit `CountDownLatch`.

```
package countdown;

import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo {
    public static void main(String[] args) throws InterruptedException {
        final int n = 3;
        CountDownLatch start = new CountDownLatch(1);
        CountDownLatch done = new CountDownLatch(n);

        for (int i = 0; i < n; i++) {
            new Thread(new Worker("Worker " + i, start, done)).start();
        }

        System.out.println("BEGIN");
        start.countDown();
        done.await();
        System.out.println("END");
    }
}

class Worker implements Runnable {
    private final String name;
    private final CountDownLatch start;
    private final CountDownLatch done;

    Worker(String name, CountDownLatch start, CountDownLatch done) {
        this.name = name;
        this.start = start;
        this.done = done;
    }

    @Override
    public void run() {
        try {
            start.await();
            doWork();
            done.countDown();
        } catch (InterruptedException ignored) {
        }
    }
}
```

```
void doWork() throws InterruptedException {
    Thread.sleep((1 + (int) (Math.random() * 5)) * 1000);
    System.out.println(name + " beendet");
}
}
```

Ein Worker-Thread wartet mit `start.await()` auf den Zählerstand 0, um beginnen zu können. Ist er mit seiner Arbeit fertig, erniedrigt er den Zähler `done` um eins.

Der `main`-Thread gibt mit `startCountdown()` den Startschuss und wartet mit `done.await()` auf die Beendigung aller Worker-Threads.

Ausgabe des Programms (Beispiel):

```
BEGIN
Worker 2 beendet
Worker 0 beendet
Worker 1 beendet
END
```

Im Unterschied zur `Thread`-Methode `join` benötigt der `main`-Thread keine Referenz auf die jeweiligen Threads.

## 26.7 Shutdown-Threads

Programme können aus verschiedenen Gründen während der Ausführung abbrechen, z. B. dadurch, dass eine Ausnahme nicht abgefangen wird oder dass der Benutzer das Programm durch *Strg+C* terminiert.

Das Programm `ShutdownTest1` gibt ein Beispiel.

Das Programm bricht mit einer `RuntimeException` ab. Die Log-Datei kann nicht mehr geschlossen werden, sodass der Puffer nicht herausgeschrieben werden kann. Die Datei ist leer.

```
package shutdown;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class ShutdownTest1 {
    private final BufferedWriter log;

    public ShutdownTest1() throws IOException {
        log = new BufferedWriter(new FileWriter("log.txt"));
    }

    public void process() throws IOException {
        log.write("Das ist ein Test.");
        log.newLine();
    }
}
```

```
try {
    Thread.sleep(60000);
} catch (InterruptedException ignored) {
}
}

public void close() throws IOException {
    log.close();
}

public static void main(String[] args) throws IOException {
    ShutdownTest1 test = new ShutdownTest1();
    test.process();
    test.close();
}
}
```

Eine Lösung des Problems zeigt die folgende Programmvariante. Hier wird die Datei in jedem Fall geschlossen, gleichgültig auf welche Weise das Programm terminiert.

```
package shutdown;

import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;

public class ShutdownTest2 {
    private final BufferedWriter log;

    public ShutdownTest2() throws IOException {
        log = new BufferedWriter(new FileWriter("log.txt"));

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            try {
                close();
            } catch (IOException ignored) {
            }
        }));
    }

    public void process() throws IOException {
        log.write("Das ist ein Test.");
        log.newLine();
        try {
            Thread.sleep(60000);
        } catch (InterruptedException ignored) {
        }
    }

    public void close() throws IOException {
        log.close();
    }
}
```

```
public static void main(String[] args) throws IOException {
    ShutdownTest2 test = new ShutdownTest2();
    test.process();
}
```

Hier wird ein Thread-Objekt erzeugt, dessen `run`-Methode die `close`-Methode aufruft. Dieses Objekt wird der `java.lang.Runtime`-Methode `addShutdownHook` übergeben:

```
void addShutdownHook(Thread hook)
```

registriert das Thread-Objekt `hook`. Dieser Thread wird gestartet, wenn die Java Virtual Machine (JVM) die Terminierung des Programms (*Shutdown*) einleitet.

Eine Instanz der Klasse `Runtime` repräsentiert das gestartete Programm und die Umgebung, in der es läuft. Die Klassenmethode `Runtime.getRuntime()` liefert eine Referenz auf das aktuelle `Runtime`-Objekt.

Hinweis:

In seltenen Fällen kann der JVM-Prozess so abgebrochen werden, dass der Shutdown-Hook nicht mehr ausgeführt werden kann, z. B. wenn man den Prozess unter Linux mit

```
kill -SIGKILL pid
```

abbricht.

## 26.8 Thread-Pools und Executors

Mehrere Klassen und Interfaces des Pakets `java.util.concurrent` bieten Methoden, um Aufgaben zur nebenläufigen Ausführung entgegenzunehmen, auf deren Beendigung zu warten oder sie abzubrechen.

Ein *Thread-Pool* verwaltet mehrere Threads. Eine Aufgabe wird zur Ausführung einem Thread des Pools zugewiesen. Nach Abschluss der Aufgabe wird der Thread ohne zu terminieren zur Wiederverwendung in den Pool gestellt.

Die Klasse `Executors` kann z. B. folgende Thread-Pools erzeugen:

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int n)
```

Mit `newCachedThreadPool()` wird ein Pool erzeugt, der nach Bedarf neue Threads erstellt und dem Pool hinzugefügt. Threads, die 60 Sekunden lang nicht verwendet wurden, werden beendet und aus dem Cache entfernt.

Mit `newFixedThreadPool(int n)` wird ein Pool mit einer festen Anzahl von Threads erzeugt. Zu jedem Zeitpunkt sind höchstens `n` Threads aktiv. Wenn zusätzliche

Aufgaben übermittelt werden, während alle Threads aktiv sind, warten diese in einer Warteschlange, bis ein Thread verfügbar ist.

Das Funktionsinterface `Executor` (Super-Interface von `ExecutorService`) stellt die folgende Methode zur Ausführung einer Aufgabe bereit:

```
void execute(Runnable command)
```

Mit der Methode

```
void shutdown()
```

des Interface `ExecutorService` wird der Thread-Pool kontrolliert beendet. Zuvor übermittelte Aufgaben werden noch ausgeführt, neue Aufgaben werden jedoch abgewiesen.

Ein Beispiel:

```
package threadpool;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecuteRunnable {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 3; i++) {
            final int n = i;
            service.execute(() -> {
                String name = Thread.currentThread().getName();
                System.out.println(n + ": " + name);
            });
        }

        service.shutdown();
    }
}
```

Weitere `ExecutorService`-Methoden:

```
Future<?> submit(Runnable task)
```

übermittelt eine Aufgabe zur Ausführung und gibt ein `Future`-Objekt zurück, das diese Aufgabe darstellt. Die `get`-Methode von `Future` gibt nach erfolgreichem Abschluss `null` zurück.

```
Future<T> submit(Callable<T> task)
```

übermittelt eine Aufgabe zur Ausführung, die einen Wert zurück gibt. Das `Future`-Objekt stellt das ausstehende Ergebnis der Aufgabe dar. Die `get`-Methode von `Future` gibt das Ergebnis der Aufgabe nach erfolgreichem Abschluss zurück.

Das Funktionsinterface `Callable` deklariert die Methode:

`V call() throws Exception`

Das Interface `Future<V>` deklariert die folgenden Methoden:

`V get() throws InterruptedException, ExecutionException`

wartet ggf. auf die Beendigung der Aufgabe und ruft dann das Ergebnis ab.

`boolean isDone()`

gibt true zurück, wenn die Aufgabe abgeschlossen ist.

`boolean cancel(boolean mayInterruptIfRunning)`

versucht, die Ausführung der Aufgabe abzubrechen. Ist die Aufgabe noch nicht gestartet, wird sie nicht ausgeführt. Wenn die Aufgabe bereits gestartet wurde, bestimmt `mayInterruptIfRunning`, ob der Thread, der diese Aufgabe ausführt, unterbrochen wird.

`boolean isCancelled()`

gibt true zurück, wenn diese Aufgabe abgebrochen wurde.

Beispiel **SubmitRunnable1**: `submit` mit Aufgabe vom Typ `Runnable`

`future.get()` wartet auf den Abschluss der Aufgabe.

```
Future<?> future = service.submit(() -> {
    int n = 3;
    while (n > 0) {
        System.out.println(LocalTime.now());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ignored) {
        }
        n--;
    }
});
try {
    future.get();
} catch (InterruptedException | ExecutionException ignored) {
}
System.out.println(future.isDone());
```

Beispiel **SubmitRunnable2**: `submit` mit Aufgabe vom Typ `Runnable`

Die begonnene Aufgabe wird abgebrochen.

```
Future<?> future = service.submit(() -> {
    int n = 3;
    while (n > 0) {
        System.out.println(LocalTime.now());
```

```

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            break;
        }
        n--;
    }
});
try {
    Thread.sleep(1000);
} catch (InterruptedException ignored) {
}
future.cancel(true);
System.out.println(future.isCancelled());

```

Beispiel **SubmitRunnable3**: submit mit Aufgabe vom Typ Runnable

Nicht behandelte Ausnahmen werden bei Verwendung von submit abgefangen. Erst beim Zugriff mit get auf das Future-Objekt wird die Ausnahme ausgelöst.

```

Future<?> future = service.submit(() -> System.out.println(1 / 0));
try {
    future.get();
} catch (InterruptedException | ExecutionException e) {
    System.out.println(e.getMessage());
}

```

Beispiel **SubmitCallable**: submit mit Aufgabe vom Typ Callable

future.get() liefert das Ergebnis.

```

Future<Integer> future = service.submit(() -> 42);
try {
    int result = future.get();
    System.out.println(result);
} catch (InterruptedException | ExecutionException ignored) {
}

```

## 26.9 Asynchrone Verarbeitung mit CompletableFuture

Mit Hilfe der Klasse

```
java.util.concurrent.CompletableFuture
```

können auf einfache Art und Weise *asynchrone Abläufe* als Verkettung mehrerer Aufgaben formuliert und ausgeführt werden.

Die Klasse `CompletableFuture` repräsentiert das Ergebnis einer asynchronen Berechnung. Aufgaben werden durch Threads eines Thread-Pools, dem *CommonPool*, ausgeführt. Komplexe Berechnungen bzw. Verarbeitungen können als eine Folge

von Aufgaben beschrieben werden, die durch bestimmte Methodenaufrufe miteinander verknüpft werden können.

Der Start für eine asynchrone Verarbeitungskette erfolgt in unseren Beispielen immer mit der statischen Methode

```
static <T> CompletableFuture<T> supplyAsync(Supplier<T> supplier)
```

Der übergebene Supplier berechnet ein Ergebnis vom Typ T.

CompletableFuture implementiert die Interfaces Future und CompletionStage.

CompletionStage stellt eine Reihe von Methoden zur Verkettung von Aufgaben bereit:

```
<U> CompletableFuture<U> thenApplyAsync(  
    Function<? super T,>? extends U> fn)
```

führt eine Funktion aus, wobei das Argument der Funktion das Ergebnis der vorhergehenden Berechnung ist.

```
CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)  
verarbeitet abschließend das Ergebnis der vorhergehenden Berechnung.
```

```
CompletableFuture<V> thenCombineAsync(CompletionStage<? extends U> other,  
    BiFunction<? super T,>? super U,>? extends V> fn)  
verbindet das Ergebnis zweier vorhergehender Berechnungen mit Hilfe der  
BiFunction fn.
```

Noch zwei nützliche CompletableFuture-Methoden:

```
static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)  
gibt ein neues CompletableFuture zurück, das abgeschlossen ist, wenn alle  
übergebenen CompletableFutures abgeschlossen sind.
```

```
T join()  
wartet auf das Ende der Verarbeitung und liefert das Ergebnis.
```

Die Aufgaben in den folgenden Beispielen sind bewusst einfach gehalten, um die Art der Verkettung deutlich herauszustellen.

## Lineare Kette

```
package async;  
  
import java.util.concurrent.CompletableFuture;  
  
public class CompletableFutureDemo1 {  
    public static void main(String[] args) {  
        System.out.println("Variante 1");  
        CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> 1);  
        CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x + 2);  
        CompletableFuture<Integer> f3 = f2.thenApplyAsync(x -> x + 3);  
    }  
}
```

```

CompletableFuture<Void> f4 = f3.thenAcceptAsync(System.out::println);
f4.join();

System.out.println("Variante 2");
CompletableFuture.supplyAsync(() -> 1)
    .thenApplyAsync(x -> x + 2)
    .thenApplyAsync(x -> x + 3)
    .thenAcceptAsync(System.out::println)
    .join();
System.out.println("Ende");
}
}

```

Ausgabe:

```

Variante 1
6
Variante 2
6
Ende

```

Hier wird jeweils das Ergebnis der vorhergehenden Berechnung als Argument für die nachfolgende Berechnung übergeben. Das Endergebnis wird ausgegeben.

Die zweite Variante ist deutlich besser zu lesen (*Fluent Programming Style*).

## Verzweigen und Zusammenführen

```

package async;

import java.util.concurrent.CompletableFuture;

public class CompletableFutureDemo2 {
    public static void main(String[] args) {
        CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> 1);
        CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x + 2);
        CompletableFuture<Integer> f3 = f1.thenApplyAsync(x -> x + 3);
        CompletableFuture<Integer> f4 = f2.thenCombine(f3, (a, b) -> a * b);
        System.out.println(f4.join());
    }
}

```

Ausgabe:

```

12

```

Das Ergebnis der ersten Berechnung ist Eingabe für die zweite und dritte Berechnung. Zum Schluss werden die beiden Ergebnisse mit einer `BiFunction` verknüpft.

### Auf den Abschluss aller Aufgaben warten

Mit `allOf` wird auf alle übergebenen Aufgaben gewartet, bis sie abgeschlossen sind.

```
package async;

import java.util.concurrent.CompletableFuture;

public class CompletableFutureDemo3 {
    public static void main(String[] args) {
        CompletableFuture<Void> f1 = CompletableFuture.supplyAsync(() -> 1)
            .thenAcceptAsync(System.out::println);
        CompletableFuture<Void> f2 = CompletableFuture.supplyAsync(() -> 2)
            .thenAcceptAsync(System.out::println);
        CompletableFuture.allOf(f1, f2).join();
        System.out.println("Ende");
    }
}
```

Ausgabe:

```
1
2
Ende
```

## 26.10 Das Process-API

Mit dem mit Java 9 eingeführten neuen *Process-API* können Prozesse betriebsystemunabhängig kontrolliert werden.

Die folgenden Beispiele demonstrieren den Umgang mit diesem API.

Die `ProcessHandle`-Methode

```
static Stream<ProcessHandle> allProcesses()
```

gibt einen *Snapshot* aller Prozesse zurück, die für den aktuellen Prozess sichtbar sind.

Die Klasse `Util` implementiert die Methode `findProcessByName`, die den Prozess zu einem vorgegebenen Kommando liefert.

```
package process;

import java.util.Optional;

public class Util {
    public static Optional<ProcessHandle> findProcessByName(String name) {
        return ProcessHandle.allProcesses()
```

```

        .filter(h -> {
            Optional<String> opt = h.info().command();
            return opt.isPresent() && opt.get().endsWith(name);
        })
        .findFirst();
    }
}

```

`ProcessHandle.Info info()`  
liefert Informationen zum Prozess.

Die `ProcessHandle.Info`-Methode

`Optional<String> command()`  
liefert den Pfadnamen des ausführbaren Kommandos zum Prozess.

Weitere `ProcessHandle`-Methoden:

`boolean destroy()`  
terminiert den Prozess.

`long pid()`  
liefert die Prozess-Id.

`CompletableFuture<ProcessHandle> onExit()`  
gibt ein `CompletableFuture`-Objekt zurück, das verwendet werden kann, um auf die Terminierung des Prozesses zu warten oder eine Aktion als Reaktion auf das Ende auszuführen.

Das Programm `Test1` beendet einen laufenden Prozess. Dazu muss vor Ausführung dieses Programms, der Editor `xed` (Linux) oder `notepad.exe` (Windows) gestartet sein.

```

package process;

import java.util.Optional;

public class Test1 {
    public static void main(String[] args) {
        String name = "xed";
        Optional<ProcessHandle> opt = Util.findProcessByName(name);
        if (opt.isPresent()) {
            ProcessHandle handle = opt.get();
            handle.destroy();
            handle.onExit()
                .thenAccept(h -> System.out.println("Prozess " + h.pid()
                    + " wurde beendet."))
                .join();
        } else {
            System.out.println("Prozess existiert nicht.");
        }
    }
}

```

Programm Test2 zeigt, wie auf das Beenden eines Prozesses gewartet und dann reagiert werden kann.

```
package process;

import java.util.Optional;

public class Test2 {
    public static void main(String[] args) throws Exception {
        String name = "xed";
        Optional<ProcessHandle> opt = Util.findProcessByName(name);
        if (opt.isPresent()) {
            ProcessHandle handle = opt.get();
            System.out.println("Warte auf Beendigung von " + name + " ...");
            handle.onExit().join();
            System.out.println("Prozess " + handle.pid() + " wurde beendet.");
        } else {
            System.out.println("Prozess existiert nicht.");
        }
    }
}
```

## 26.11 Virtuelle Threads

Mit Java 21 wurden leichtgewichtige *virtuelle Threads* eingeführt. Ihr Einsatz kann den Durchsatz bei sehr vielen parallelen, I/O-intensiven Aktivitäten (Tasks) in einem Programm (z. B. bei der Bearbeitung von Server-Anfragen) erheblich verbessern, wenn keine allzu umfangreichen Berechnungen in einer Task erfolgen.

Virtuelle Threads verhalten sich aus Sicht des Java-Codes wie ganz normale Threads, werden aber im Gegensatz zu den herkömmlichen Java-Threads nicht 1:1 auf *Betriebssystem-Threads* abgebildet.

Viele virtuelle Threads teilen sich einen Betriebssystem-Thread als *Träger-Thread*. Sobald ein virtueller Thread warten muss oder blockiert wird, wird dieser vom Träger-Thread genommen und der Träger-Thread führt einen anderen, zuvor blockierten oder neuen, virtuellen Thread aus.

Programme, die das bisherige "klassische" Thread-API nutzen (wie auch hier in diesem Kapitel) lassen sich mit wenigen Änderungen auf virtuelle Threads umstellen.

Ein virtueller Thread kann mit der folgenden Thread-Methode gestartet werden:

```
static Thread startVirtualThread(Runnable task)
```

Alternativ kann mit der Executors-Methode

```
static ExecutorService newVirtualThreadPerTaskExecutor()
```

ein Executor erzeugt werden, der mit vielen virtuellen Threads arbeitet.

Wir wollen im Folgenden den Durchsatz bei Verwendung von "klassischen" *Java-Threads* und *virtuellen Threads* vergleichen.

Hierzu nutzen wir die folgende Task:

```
package virtual_threads;

import java.util.concurrent.Callable;

public class Task implements Callable<Integer> {
    private final int number;

    public Task(int number) {
        this.number = number;
    }

    @Override
    public Integer call() throws Exception {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            return 0;
        }
        return number;
    }
}
```

Hier schläft die Task eine Sekunde und gibt dann eine Zahl zurück.

In der *ersten Variante* wird die Anzahl der Threads auf 100 begrenzt (Thread-Pool). Für jede Task wird ein Betriebssystem-Thread verwendet.

```
package virtual_threads;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Test1 {
    public static void main(String[] args)
            throws ExecutionException, InterruptedException {

        ExecutorService executor = Executors.newFixedThreadPool(100);
        long start = System.currentTimeMillis();

        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 10_000; i++) {
            futures.add(executor.submit(new Task(i)));
        }

        long sum = 0;
```

```
        for (Future<Integer> future : futures) {
            sum += future.get();
        }

        long end = System.currentTimeMillis();
        System.out.println("sum = " + sum + ", time = " + (end - start) + " ms");
        executor.shutdown();
    }
}
```

Ausgabe des Programms (Beispiel):

```
sum = 49995000, time = 100026 ms
```

Durchsatz: 100 Tasks pro Sekunde

In der *zweiten Variante* wird für jede Task ein neuer virtueller Thread erzeugt.

```
package virtual_threads;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Test2 {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
        long start = System.currentTimeMillis();

        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 1_000; i++) {
            futures.add(executor.submit(new Task(i)));
        }

        long sum = 0;
        for (Future<Integer> future : futures) {
            sum += future.get();
        }

        long end = System.currentTimeMillis();
        System.out.println("sum = " + sum + ", time = " + (end - start) + " ms");
        executor.shutdown();
    }
}
```

Ausgabe des Programms (Beispiel):

```
sum = 49995000, time = 1184 ms
```

---

Durchsatz: 10.000 Tasks pro Sekunde

Man sieht deutlich, dass der Einsatz virtueller Threads den Durchsatz erheblich verbessern kann.

## 26.12 Aufgaben

### Aufgabe 1

Erstellen Sie ein Programm mit zwei Threads, von denen einer das Wort "Hip" und der andere das Wort "HOP" in einer Endlosschleife ausgibt. Realisieren Sie zwei Programmvarianten:

- Ableitung von der Klasse Thread
- Implementierung des Interface Runnable.

*Lösung: Paket hiphop*

### Aufgabe 2

Schreiben Sie ein Programm, das den Namen und die Priorität des Threads ausgibt, der die Methode main ausführt.

*Lösung: Paket info*

### Aufgabe 3

Entwickeln Sie eine Klasse, die das Interface Runnable implementiert sowie die Methoden main, start und stop. start erzeugt einen neuen Thread und startet ihn, stop beendet ihn. Für jeden neu erzeugten Thread ist auch ein neuer Name zur Unterscheidung zu wählen. Der Thread soll seinen Namen in Abständen von zwei Sekunden ausgeben. In main können die Methoden start und stop, gesteuert über Tastatureingabe, aufgerufen werden.

*Lösung: Paket start\_stop*

### Aufgabe 4

Ein Thread soll beendet werden. Das folgende Programm wird jedoch nie beendet:

```
private static boolean stop;

public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(() -> {
        while (!stop) {
            // tue etwas
        }
    });
    t.start();
```

```
    Thread.sleep(1000);
    stop = true;
}
```

Korrigieren Sie das Programm auf zwei Weisen:

- Synchronisation der Zugriffe auf die Variable `stop`
- Nutzung des Modifizierers `volatile`

*Lösung: Paket stop*

### Aufgabe 5

Zwei parallel laufende Threads sollen Nachrichten in dieselbe Protokolldatei schreiben. Entwickeln Sie die Klasse `LogFile`, die die Protokolldatei verwaltet.

Die Klasse soll die Methode

```
public synchronized void writeLine(String msg)
```

enthalten, die eine Zeile bestehend aus Systemzeit und `msg` schreibt. Erstellen Sie ein Testprogramm, das die Protokolldatei erzeugt, zwei Threads startet, die in diese Datei mehrere Nachrichten schreiben, und das die Datei schließt, nachdem die Ausführung der beiden Threads beendet ist.

*Lösung: Paket logging*

### Aufgabe 6

Schreiben Sie ein Programm, in dem mehrere Threads zur gleichen Zeit einen zufälligen Betrag auf dasselbe Konto einzahlen. Die Additionsmethode soll aus den einzelnen Threads mehrfach aufgerufen werden. Geben Sie am Ende die Anzahl der insgesamt durchgeführten Additionen und den aktuellen Kontostand aus. Schreiben Sie das Programm so, dass keine Additionen verloren gehen.

*Lösung: Paket bank*

### Aufgabe 7

In einem Thread soll ein Zähler mit Hilfe einer Schleife 100 Millionen mal um 1 erhöht werden. Starten Sie in der `main`-Methode zwei solcher Threads hintereinander, die beide auf die gleiche Zählervariable, deren Anfangswert 0 enthält, zugreifen und geben Sie am Ende, nachdem die beiden Threads ihre Arbeit beendet haben, den Endstand des Zählers aus.

Warum wird nicht die erwartete Zahl 200.000.000 ausgegeben? Offensichtlich gehen Erhöhungen des Zählers verloren.

Wie kann das Programm so geändert werden, dass stets der Endstand 200.000.000 beträgt?

*Lösung: Paket zaehler*

### Aufgabe 8

Implementieren Sie mit Hilfe eines Arrays einen *Ringpuffer*, in den Messpunkte geschrieben werden können. Ein Messpunkt-Objekt verwaltet eine Zeitpunktangabe und einen numerischen Wert.

Der Ringpuffer hat eine feste Länge. Ist der Puffer voll, so soll der jeweils älteste Eintrag überschrieben werden. Ein Index gibt die Position im Array an, an der die nächste Schreiboperation stattfindet. Nach jedem Schreiben wird der Index um 1 erhöht und auf 0 gesetzt, wenn die obere Grenze des Arrays überschritten wurde.

Schreibende und lesende Zugriffe sollen bei wechselseitigem Ausschluss in jeweils eigenen Threads stattfinden. Die Threads sollen mehrere schreibende bzw. lesende Zugriffe hintereinander mit zeitlicher Verzögerung in einer Schleife ausführen.

Es sollen immer alle gespeicherten Messpunkte im Puffer in der Reihenfolge der Einträge gelesen werden, der älteste Eintrag zuerst.

*Lösung: Paket ring*

### Aufgabe 9

Zwei Threads sollen wechselseitig Geldbeträge überweisen. Dazu werden zwei Konten angelegt.

Thread 1 überweist in einer Schleife mit 10000 Durchgängen jeweils einen Betrag von Konto 1 auf Konto 2.

Thread 2 überweist von Konto 2 auf Konto 1 ebenfalls in einer Schleife mit 10000 Durchgängen.

Gegeben sei die folgende Klasse:

```
public class Konto {
    private int saldo;

    public Konto(int saldo) {
        this.saldo = saldo;
    }

    public synchronized int getSaldo() {
        return saldo;
    }

    public synchronized void zahleEin(int betrag) {
        saldo += betrag;
    }

    public synchronized void transfer(int betrag, Konto konto) {
        saldo -= betrag;
        konto.zahleEin(betrag);
    }
}
```

Die "kritischen" Methoden sind alle synchronisiert. Bei der Ausführung kommt es aber zu einem *Deadlock*.

Analysieren Sie den Code. Beachten Sie, dass in der `synchronized`-Methode `transfer` die `synchronized`-Methode `zahleEin` aufgerufen wird.

Wie kann man die Situation auflösen?

*Lösung: Paket deadlock*

### Aufgabe 10

Mehrere Threads sollen jeweils einen gemeinsamen Zähler und einen Threadlokalen Zähler um eins hochzählen. Nutzen Sie die Klasse `ThreadLocal`.

Ausgabebeispiel:

```
Thread-0: global = 1, local = 1
Thread-2: global = 2, local = 1
Thread-1: global = 3, local = 1
Thread-0: global = 4, local = 2
Thread-2: global = 5, local = 2
Thread-1: global = 6, local = 2
Thread-0: global = 7, local = 3
Thread-2: global = 8, local = 3
Thread-1: global = 9, local = 3
```

*Lösung: Paket local*

### Aufgabe 11

Ein `MessageSender` soll Nachrichten an alle registrierten `MessageReceiver` senden. `MessageReceiver` implementiert das Interface `MessageListener`:

```
public interface MessageListener {
    void displayMessage(String message);
}
```

`MessageSender` enthält eine Datenstruktur für die Verwaltung von *Listener* und bietet die Methoden:

```
void register(MessageListener listener)
void unregister(MessageListener listener)
void sendMessage(String message)
```

Nutzen Sie als Datenstruktur `CopyOnWriteArrayList`. Testen Sie alle Methoden mit mehreren *Listener*.

*Lösung: Paket listener*

### Aufgabe 12

Schreiben Sie ein Programm, das das folgende *Producer/Consumer-Problem* löst:

Ein Thread (Produzent) erzeugt Zufallszahlen vom Typ `int` zwischen 0 und 60 und speichert sie in einem `Vector`-Objekt.

Ein zweiter Thread (Konsument) verbraucht diese Zahlen, indem er sie in einem Balkendiagramm (Ausgabe von so vielen Zeichen \* wie die Zahl angibt) darstellt. Ist die Zahl konsumiert, muss sie vom Konsumenten aus dem Vektor entfernt werden. Ist der Vektor leer, muss der Konsument so lange warten, bis der Produzent wieder eine Zahl gespeichert hat.

Produzent und Konsument sollen nach jedem Schritt eine kleine Pause einlegen. Nutzen Sie zur Synchronisation `synchronized`-Blöcke sowie die Methoden `wait` und `notify`.

*Lösung: Paket producer\_consumer*

### Aufgabe 13

Erstellen Sie eine Variante zur Lösung von Aufgabe 12, die anstelle eines Vektors den Pipe-Mechanismus nutzt.

*Lösung: Paket pipe*

### Aufgabe 14

Erstellen Sie eine Variante zur Lösung von Aufgabe 12, die anstelle eines Vektors eine Instanz der Klasse

```
java.util.concurrent.LinkedBlockingQueue<T>
```

nutzt.

Diese Klasse implementiert das Interface

```
java.util.concurrent.BlockingQueue<T>
```

und implementiert insbesondere die beiden Methoden

```
void put(T t) throws InterruptedException
```

und

```
T take() throws InterruptedException
```

`put` fügt ein Element an das Ende der Schlange ein. Ggf. wartet `put` so lange, bis Speicherplatz zur Verfügung steht. `take` entfernt ein Element vom Anfang der Schlange. Ggf. wartet `take` so lange, bis das Element zur Verfügung steht.

Der Konstruktor `LinkedBlockingQueue(int capacity)` erzeugt eine Instanz mit der angegebenen Aufnahmekapazität.

*Lösung: Paket queue*

### Aufgabe 15

Ein Patient besucht eine Arztpraxis mit genau einem Behandlungsraum. Er erhält eine Nummer und kann den Behandlungsraum erst dann betreten, wenn der Raum frei ist und seine Nummer aufgerufen wurde. Nach einiger Zeit verlässt der Patient

den Behandlungsraum. Es kann nur ein Patient zu einer Zeit im Raum behandelt werden.

Erstellen Sie die Klassen `Patient`, `Behandlungsraum` und ein Testprogramm, das die obige Situation simuliert. Es sollen mehrere `Patient`-Threads erzeugt und gestartet werden.

Die Klasse `Behandlungsraum` verwaltet die aktuell aufgerufene Nummer, die nächste auszugebende Nummer und den Status "besetzt". Die Klasse enthält die folgenden Methoden:

`int registriere()`

liefert die nächste Wartenummer. Die Registrierung dauert 3 bis 8 Sekunden.

`void betrete(int nummer)`

wartet, falls der Raum besetzt ist oder die eigene Nummer nicht mit der aufgerufenen Nummer übereinstimmt. Anschließend wird der Raum als besetzt gekennzeichnet und die aufgerufene Nummer um 1 erhöht.

`void verlasse()`

gibt den Raum frei und weckt alle wartenden Threads.

Nutzen Sie geeignete Synchronisationsmechanismen.

*Lösung: Paket arzt*

### Aufgabe 16

Simulieren Sie das Verhalten bei knappen Ressourcen, um die sich mehrere Threads bewerben. Ein Thread wartet auf die Verfügbarkeit einer Ressource (Methode `acquire`), nutzt diese eine bestimmte Zeit lang und gibt sie dann wieder frei (Methode `release`). `acquire` nutzt die Methode `wait`, `release` nutzt die Methode `notify`.

Testen Sie das Programm mit einer unterschiedlichen Zahl von Ressourcen bei konstanter Anzahl Threads.

*Lösung: Paket resources*

### Aufgabe 17

Das *Philosophenproblem*:

Fünf Philosophen sitzen an einem runden Tisch und jeder hat einen Teller mit Spaghetti vor sich. Zum Essen von Spaghetti benötigt jeder Philosoph zwei Gabeln. Allerdings sind im Haushalt nur fünf Gabeln vorhanden, die nun zwischen den Tellern liegen. Die Philosophen können also nicht gleichzeitig speisen.

Die Philosophen sitzen am Tisch und denken über philosophische Probleme nach. Wenn einer hungrig wird, greift er zuerst die Gabel links von seinem Teller, dann die auf der rechten Seite und beginnt zu essen. Wenn er satt ist, legt er die Gabeln wieder zurück und beginnt wieder zu denken. Sollte eine Gabel nicht an ihrem

Platz liegen, wenn der Philosoph sie aufnehmen möchte, so wartet er, bis die Gabel wieder verfügbar ist.

Solange nur einzelne Philosophen hungrig sind, funktioniert dieses Verfahren wunderbar. Es kann aber passieren, dass sich alle fünf Philosophen gleichzeitig entschließen, zu essen. Sie ergreifen also alle gleichzeitig ihre linke Gabel und nehmen damit dem jeweils links von ihnen sitzenden Kollegen seine rechte Gabel weg. Nun warten alle fünf darauf, dass die rechte Gabel wieder auftaucht. Das passiert aber nicht, da keiner der fünf seine linke Gabel zurücklegt. Die Philosophen verhungern.<sup>3</sup>

Implementieren Sie Programm, das dieses Verhalten simuliert.

*Lösung: Paket dining\_philosophs*

### Aufgabe 18

Stellen Sie das Programm `ConcurrentSum` aus Kapitel 26.4 auf `ExecutorService` mit fixem Thread-Pool um.

*Lösung: Paket pool*

### Aufgabe 19

Die Klasse `Service` implementiert die folgenden Methoden:

```
public static User getUser(int userId)
public static Profile getProfile(User user)
public static AccessRight getAccessRight(Profile profile)
```

Die Bedienung zur Ermittlung der Zugangsberechtigung eines Users erfolgt in folgenden Schritten:

- Ermittlung des Users mittels User-Id,
- Anforderung des Benutzerprofils,
- Abfrage der Zugriffsrechte.

Codieren Sie mit Hilfe von `CompletableFuture` einen asynchronen Ablauf unter Verwendung von `supplyAsync`, `thenApplyAsync` und `thenAcceptAsync`.

Die Klassen `User`, `Profile` und `AccessRight` sind minimal zu implementieren.<sup>4</sup>

*Lösung: Paket async*

---

3 Siehe <https://de.wikipedia.org/wiki/Philosophenproblem>

4 Beispiel angelehnt an: Hettel, J.; Tran, M. T.: Nebenläufige Programmierung mit Java. dpunkt.verlag 2016

**Aufgabe 20**

Entwickeln Sie ein Programm, dass alle Prozesse, die ein Kommando haben, mit Prozess-Id und Kommando ausgibt. Dabei soll der eigene Prozess in der Ausgabe besonders markiert werden.

Die ProcessHandle-Methode

```
static ProcessHandle current()
```

liefert den ProcessHandle für den aktuellen Prozess.

*Lösung: Paket pid*



## 27 GUI-Programmierung mit Swing

Java hat zwei Pakete zur Programmierung grafischer Benutzerschnittstellen, auch GUI (*Graphical User Interface*) genannt:

`java.awt`

AWT steht für Abstract Window Toolkit. Es handelt sich hier um das ältere Paket. Mit Hilfe seiner Klassen können Fenster mit Menüs, Textfelder und Bedienelementen realisiert werden. Zur Darstellung vieler AWT-Komponenten wird auf das jeweilige Betriebssystem zurückgegriffen.

`javax.swing`

Swing-Komponenten sind komplett in Java implementiert und vom verwendeten Betriebssystem unabhängig (*lightweight*), sodass ein einheitliches Aussehen und Verhalten (*Look & Feel*) auf allen Rechnern ermöglicht wird. Swing ersetzt zwar alle Grundkomponenten des AWT, nutzt aber einige Leistungen des AWT, wie z. B. die Ereignisbehandlung.

In diesem Kapitel stehen die Programmbeispiele im Vordergrund. Anhand dieser Beispiele werden die diversen Möglichkeiten gezeigt und erläutert. Aufgrund der Vielzahl an Klassen und Methoden musste eine Auswahl getroffen werden, die aber die wichtigsten Features zeigt.

### Lernziele

In diesem Kapitel lernen Sie

- wichtige *GUI-Komponenten* für die Interaktion mit dem Benutzer kennen,
- wie Ereignisse, z. B. der Klick auf einen Button, behandelt werden (*Event-Handling*) und
- wie Komponenten in einem Container mit Hilfe von *Layout-Managern* angeordnet werden können.

### Komponenten und Container

Alle Swing-Klassen sind von `java.awt.Component` abgeleitet.

Die Klasse `Component` enthält grundlegende Methoden, die eine Komponente am Bildschirm darstellen und sie für die Benutzerinteraktion vorbereiten.

Objekte der Klasse `java.awt.Container` sind Komponenten, die selbst wiederum Komponenten aufnehmen können. Die Klasse stellt Methoden zur Verfügung, um Komponenten hinzuzufügen, zu positionieren oder zu entfernen.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_27](https://doi.org/10.1007/978-3-658-43574-5_27).

### Eigenschaften von Komponenten

Swing-Komponenten besitzen eine Reihe von Eigenschaften, die über Zugriffsmethoden, deren Namen mit `set` bzw. `get` beginnen, gesetzt bzw. abgefragt werden können. Hat die Eigenschaft den Typ `boolean`, so beginnt der Name der Zugriffsmethode zum Abfragen mit `is` oder `has`.

Beispiele:

Die Eigenschaft `size` der Komponente `Component` kann mit der Methode `setSize` gesetzt und mit `getSize` abgefragt werden.

Die boolesche Eigenschaft `visible` von `Component` kann mit `setVisible` gesetzt und mit `isVisible` abgefragt werden.

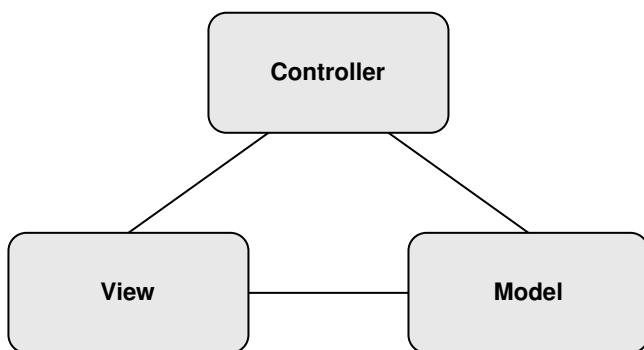
### MVC-Architektur

Viele Komponenten sind dazu da, Daten anzuzeigen, um diese dann über Benutzerinteraktion bequem auswählen bzw. ändern zu können.

Solche Komponenten basieren auf der klassischen *MVC-Architektur (Model-View-Controller-Architektur)*:

- Das Modell (*Model*) enthält die Daten, die angezeigt werden sollen,
- die Präsentation (*View*) visualisiert diese Daten in einer geeigneten Form,
- die Steuerung (*Controller*) ist für die Reaktion auf Benutzereingaben und die Ereignisbehandlung zuständig.

Bei Swing-Komponenten sind *Model* und *View* oft zu einer Einheit zusammengefasst.



**Abbildung 27-1:** MVC-Architektur

## 27.1 Ein Fenster erzeugen mit JFrame

Objekte der Klasse `javax.swing.JFrame` sind frei bewegliche Fenster mit Rahmen und Titelleiste.

Das folgende Programm zeigt ein Fenster mit grafischem Inhalt.

```
package frame;

import javax.swing.*;
import java.awt.*;

public class MyFrame extends JFrame {
    public MyFrame() {
        super("MyFrame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(new Content());
        setSize(300, 200);
        setLocation(100, 100);
        setVisible(true);
    }

    private static class Content extends JPanel {
        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.setColor(new Color(0, 0, 255, 100));
            g.fillRoundRect(20, 20, 260, 130, 30, 30);
            g.setColor(Color.WHITE);
            g.setFont(new Font("Monospaced", Font.BOLD, 48));
            g.drawString("Java", 90, 100);
        }
    }

    public static void main(String[] args) {
        new MyFrame();
    }
}
```

Die Klasse `MyFrame` ist von `JFrame` abgeleitet. Der Titel des Fensters wird mit `super` an den Konstruktor der Basisklasse weitergeleitet.

`setDefaultCloseOperation` legt die Operation fest, die ausgeführt wird, wenn das Fenster durch Anklicken des entsprechenden Buttons in der Titelleiste rechts oben geschlossen wird. Als Operation kann eine der folgenden `JFrame`-Konstanten gewählt werden:

`DO NOTHING ON CLOSE`, `DISPOSE ON CLOSE`, `EXIT ON CLOSE`, `HIDE ON CLOSE` (Voreinstellung).

`setSize` setzt die Größe der Komponente (hier das Fenster) auf den Wert Breite mal Höhe (in Pixel).

`getLocation` platziert die linke obere Ecke der Komponente an Position (100,100) der übergeordneten Komponente.

`setVisible` macht die Komponente sichtbar (`true`) oder unsichtbar (`false`).

Die Klasse `javax.swing.JPanel` ist die einfachste Form eines Containers. Als Subklasse von `javax.swing.JComponent` können Panels in andere Container eingefügt und auch ineinander verschachtelt werden. Um ein Panel anzeigen zu können, muss es in ein Fenster eingefügt werden.

Mit der Container-Methode `add` können Komponenten in einen Container aufgenommen werden.

Objekte vom Typ `JFrame` (und auch von `JDialog`) enthalten jeweils einen speziellen Container (*Content Pane*), an den der Aufruf von `add` zu richten ist. Diesen Container erhält man mit der `JFrame-` bzw. `JDialog-`Methode

```
Container getContentPane()
```

Beispiel:

Ist `component` eine Komponente und `frame` ein `JFrame`-Objekt, so wird die Komponente in das Fenster wie folgt aufgenommen:

```
frame.getContentPane().add(component);
```

Vereinfacht kann

```
frame.add(component);
```

genutzt werden.

In einem Panel können grafische Elemente ausgegeben werden. Diese Ausgabe erfolgt durch Überschreiben der `JComponent`-Methode `paintComponent`:

```
protected void paintComponent(Graphics g)
```

zeichnet die Komponente. `g` ist ein Objekt der Klasse `java.awt.Graphics`, das den sogenannten *Grafikkontext* darstellt und vom Java-Laufzeitsystem übergeben wird. Der Grafikkontext enthält diverse für das Zeichnen notwendige Basisinformationen.

`java.awt.Graphics` bietet Methoden zum Zeichnen von einfachen geometrischen Figuren und verwaltet Farbe und Schriftart, in der Grafik- und Textausgaben erfolgen.

`paintComponent` wird immer dann automatisch aufgerufen, wenn die Grafik ganz oder teilweise aktualisiert werden muss, was z. B. dann geschieht, wenn das Fenster zum ersten Mal angezeigt wird oder wenn es durch andere Elemente verdeckt war und wieder sichtbar wird.

Wenn `paintComponent` überschrieben wird, sollte als Erstes stets

```
super.paintComponent(g)
```

aufgerufen werden, damit auch der Hintergrund gezeichnet wird.

Anwendungen sollten `paintComponent` selbst *nie direkt* aufrufen, sondern stattdessen – wenn neu gezeichnet werden soll – die `Component`-Methode

```
void repaint()
```

nutzen. `repaint` darf ohne Probleme von jedem Thread aufgerufen werden.

Die verwendeten `Graphics`-Methoden:

`setColor` setzt die Farbe, die dann von allen folgenden Operationen benutzt wird.

`setFont` setzt die Schriftart, die von allen folgenden Operationen, die Text ausgeben, verwendet wird.

Der Konstruktor

```
Color(int r, int g, int b, int a)
```

erzeugt eine Farbe aus der Mischung der Rot-, Grün- und Blauanteile `r`, `g` und `b`, die Werte im Bereich von 0 bis 255 annehmen können. So liefert das Tripel (0, 0, 0) die Farbe Schwarz, (255, 255, 255) die Farbe Weiß, (255, 0, 0) Rot, (0, 255, 0) Grün und (0, 0, 255) Blau. `a` legt den Grad der Transparenz (Alpha-Kanal) im Bereich von 0 bis 255 fest.

Der Konstruktor

```
Font(String name, int style, int size)
```

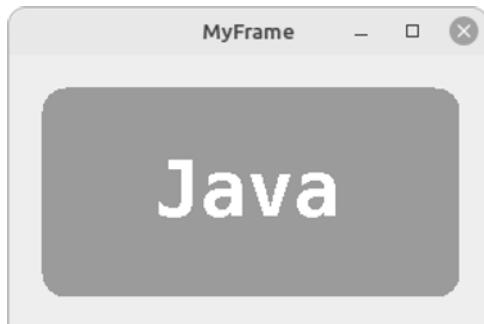
legt Schriftart, -stil und -größe fest.

```
void fillRoundRect(int x, int y, int width, int height,
                   int arcWidth, int arcHeight)
```

zeichnet ein ausgefülltes Rechteck mit abgerundeten Ecken. `arcWidth` ist der horizontale, `arcHeight` der vertikale Durchmesser des Bogens.

```
void drawString(String s, int x, int y)
```

zeichnet einen Text, wobei (`x`, `y`) das linke Ende der Grundlinie des ersten Zeichens von `s` ist.



**Abbildung 27-2:** Grafik im Fenster

## 27.2 Ereignisbehandlung

Die Kommunikation zwischen Benutzer und Anwendungsprogramm mit grafischer Oberfläche basiert auf einem Ereignismodell (*Event-Modell*).

Bisher haben wir das mit einem Fenster verbundene Programm mit Hilfe von

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
```

beendet.

Nun wollen wir durch Klicken auf das Schließen-Symbol oder durch Drücken der *ESC-Taste* das Fenster schließen und das Programm mit einer Abschlussaktion beenden.

### Ereignisse, Ereignisquellen und -empfänger

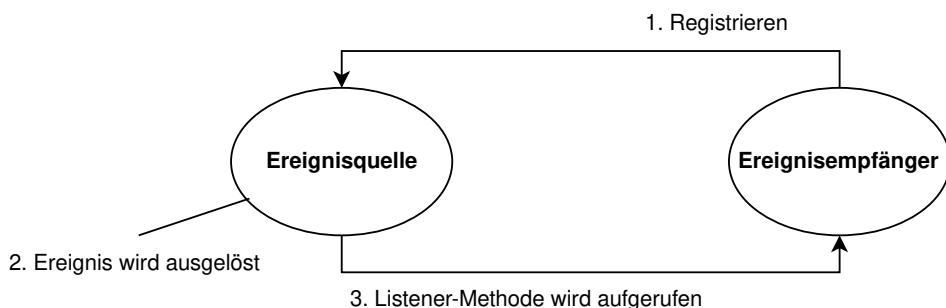
Das Paket `java.awt.event` enthält Interfaces und Klassen, die die Reaktion auf Ereignisse realisieren.

Eine Aktion, wie z. B. das Klicken auf einen Button, löst ein *Ereignis* (*Event*) aus, auf das das Programm in bestimmter Weise reagieren kann.

Ein Ereignis ist immer mit einer *Ereignisquelle* (z. B. einem Fenster oder einem Button) verbunden, die das Ereignis ausgelöst hat.

Ereignisse sind Objekte bestimmter Klassen. In unserem Beispiel handelt es sich um Objekte der Klassen `WindowEvent` und `KeyEvent`.

Die Reaktion auf Ereignisse erfolgt in sogenannten *Ereignisempfängern* (*Listener*), die bei der Ereignisquelle registriert werden müssen, wenn sie entsprechende Ereignisse mitgeteilt bekommen sollen.



**Abbildung 27-3:** Ereignismodell

Im nachfolgenden Beispiel sind die Ereignisempfänger Objekte von Klassen, die das Interface `WindowListener` bzw. `KeyListener` implementieren. Die zugehörigen Registrierungsmethoden, die für die Ereignisquelle aufgerufen werden müssen, sind:

```
void addWindowListener(WindowListener l)
void addKeyListener(KeyListener l)
```

Die Mitteilung eines Ereignisses besteht im Aufruf der passenden Interface-Methode des Listener-Objekts, die als Argument die Referenz auf das entsprechende Ereignisobjekt (`WindowEvent` bzw. `KeyEvent`) enthält.

Alle Ereignisklassen besitzen eine Methode, mit deren Hilfe das Objekt ermittelt werden kann, das das Ereignis ausgelöst hat:

```
Object getSource()
```

#### *Hinweis*

Zu allen Registrierungsmethoden `addXxxListener` existieren die dazu passenden Deregistrierungsmethoden `removeXxxListener`.

Das Interface `WindowListener` enthält die Methoden:

```
void windowActivated(WindowEvent e)
void windowClosed(WindowEvent e)
void windowClosing(WindowEvent e)
void windowDeactivated(WindowEvent e)
void windowDeiconified(WindowEvent e)
void windowIconified(WindowEvent e)
void windowOpened(WindowEvent e)
```

Das Interface `KeyListener` enthält die Methoden:

```
void keyPressed(KeyEvent e)
void keyReleased(KeyEvent e)
void keyTyped(KeyEvent e)
```

Unser Programm soll nur die beiden Methoden `windowClosing` und `keyPressed` im Ereignisempfänger implementieren, die übrigen Methoden haben deshalb leere Methodenrumpfen.

Zur Vereinfachung nutzen wir Adapterklassen. Eine *Adapterklasse* ist hier eine abstrakte Klasse, die ein vorgegebenes Interface mit nur leeren Methodenrumpfen implementiert.

WindowAdapter und KeyAdapter implementieren den WindowListener bzw. den KeyListener. Sie werden hier genutzt, um *nur die gewünschten* Methoden in Subklassen zu überschreiben und damit Schreibaufwand für die nicht relevanten Methoden des Interface zu sparen.

```
package events;

import javax.swing.*;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class CloseTest extends JFrame {
    public CloseTest() {
        super("CloseTest");

        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                print();
                dispose();
            }
        });

        addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
                    print();
                    dispose();
                }
            }
        });
    }

    setSize(300, 200);
    setVisible(true);
}

private void print() {
    System.out.println("Das Fenster wurde geschlossen.");
}

public static void main(String[] args) {
    new CloseTest();
}
```

Die KeyEvent-Methode getKeyCode gibt den Code der gedrückten Taste zurück, der dann mit KeyEvent.VK\_ESCAPE verglichen wird.

Die Methode dispose gibt alle Ressourcen des Fensters frei, damit wird hier dann das Programm beendet.

## Mausereignisse

Soll auf *Mausereignisse*, wie Drücken, Loslassen der Maustaste oder Bewegung des Mauszeigers, reagiert werden, so muss ein entsprechender Ereignisempfänger erstellt und registriert werden.

`java.awt.event.MouseEvent`-Methoden:

`Point getPoint()`

liefert die x- und y-Koordinate der Position des Mauszeigers als Objekt der Klasse `java.awt.Point`. Die Koordinaten werden relativ zum Ursprung der auslösenden Komponente angegeben.

`int getX()`

liefert die x-Koordinate der Position des Mauszeigers.

`int getY()`

liefert die y-Koordinate der Position des Mauszeigers.

`int getClickCount()`

liefert die Anzahl der hintereinander erfolgten Mausklicks.

`MouseEvent` ist von der Klasse `java.awt.event.InputEvent` abgeleitet. `InputEvent` besitzt folgende Methoden:

`boolean isAltDown()`

`boolean isShiftDown()`

`boolean isControlDown()`

Diese Methoden liefern den Wert `true`, wenn zusammen mit der Maustaste die Alt-, Shift- bzw. Control-Taste gedrückt wurde.

`boolean isMetaDown()`

liefert den Wert `true`, wenn die rechte Maustaste gedrückt wurde.

`java.awt.event.MouseListener`-Methoden:

`void mousePressed(MouseEvent e)`

wird beim Drücken der Maustaste aufgerufen.

`void mouseReleased(MouseEvent e)`

wird aufgerufen, wenn die gedrückte Maustaste losgelassen wurde.

`void mouseClicked(MouseEvent e)`

wird aufgerufen, wenn eine Maustaste gedrückt und wieder losgelassen wurde.

Die Methode wird nach `mouseReleased` aufgerufen.

`void mouseEntered(MouseEvent e)`

wird aufgerufen, wenn der Mauszeiger sich in den Bereich der auslösenden Komponente hineinbewegt.

`void mouseExited(MouseEvent e)`

wird aufgerufen, wenn der Mauszeiger sich aus dem Bereich der auslösenden Komponente herausbewegt.

java.awt.event.MouseMotionListener-Methoden:

`void mouseMoved(MouseEvent e)`

wird aufgerufen, wenn die Maus bewegt wird, ohne dass dabei eine der Mautasten gedrückt wurde.

`void mouseDragged(MouseEvent e)`

wird aufgerufen, wenn die Maus bei gedrückter Maustaste bewegt wird.

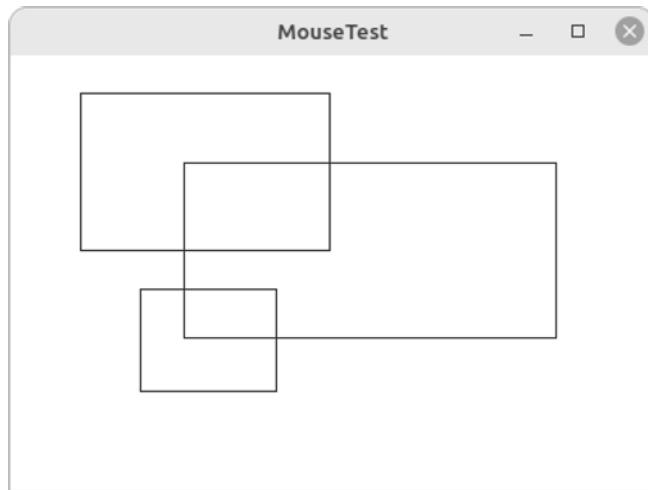
Die Adapterklasse `java.awt.event.MouseAdapter` implementiert die Interfaces `MouseListener` und `MouseMotionListener` mit leeren Methodenrumpfen.

Die Adapterklasse `java.awt.event.MouseMotionAdapter` implementiert das Interface `MouseMotionListener` mit leeren Methodenrumpfen.

Das Programm `MouseTest` demonstriert die Reaktion auf Mausereignisse. Es können Rechtecke auf einem Panel gezeichnet werden.

Das Drücken der Maustaste legt den Anfangspunkt des Rechtecks fest. Durch Ziehen der Maus wird die Größe des Rechtecks bestimmt. Beim Loslassen der Maustaste wird das Rechteck in die Liste der bereits gezeichneten Rechtecke eingetragen.

Bei jedem Aufruf von `paintComponent` bzw. `repaint` werden alle gespeicherten Rechtecke sowie das aktuelle Rechteck neu gezeichnet.



**Abbildung 27-4:** Rechtecke zeichnen

```
package events;

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.ArrayList;

public class MouseTest extends JFrame {
    public MouseTest() {
        super("MouseTest");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(new Content());
        setSize(400, 300);
        setVisible(true);
    }

    private static class Content extends JPanel {
        // aktuelles Rechteck
        private MyRectangle currentRectangle;

        // enthält alle komplett gezeichneten Rechtecke
        private final ArrayList<MyRectangle> rectangles = new ArrayList<>();

        public Content() {
            setBackground(Color.WHITE);
            MyMouseAdapter adapter = new MyMouseAdapter();
            addMouseListener(adapter);
            addMouseMotionListener(adapter);
        }

        class MyMouseAdapter extends MouseAdapter {
            @Override
            public void mousePressed(MouseEvent e) {
                // neues Rechteck erzeugen
                currentRectangle = new MyRectangle(e.getX(), e.getY(), 0, 0);
            }

            @Override
            public void mouseReleased(MouseEvent e) {
                // aktuelles Rechteck speichern
                if (currentRectangle.b > 0 && currentRectangle.h > 0)
                    rectangles.add(currentRectangle);
            }

            @Override
            public void mouseDragged(MouseEvent e) {
                int x = e.getX();
                int y = e.getY();

                // Breite und Höhe des aktuellen Rechtecks ermitteln
                if (x > currentRectangle.x && y > currentRectangle.y) {
                    currentRectangle.b = x - currentRectangle.x;
                    currentRectangle.h = y - currentRectangle.y;
                }

                repaint();
            }
        }
    }
}
```

```

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    // alle gespeicherten Rechtecke neu zeichnen
    for (MyRectangle r : rectangles) {
        g.drawRect(r.x, r.y, r.b, r.h);
    }

    // aktuelles Rechteck zeichnen
    if (currentRectangle != null) {
        g.drawRect(currentRectangle.x, currentRectangle.y,
                   currentRectangle.b, currentRectangle.h);
    }
}

static class MyRectangle {
    final int x;
    final int y;
    int b;
    int h;

    MyRectangle(int x, int y, int b, int h) {
        this.x = x;
        this.y = y;
        this.b = b;
        this.h = h;
    }
}
}

public static void main(String[] args) {
    new MouseTest();
}
}

```

Die folgenden `JComponent`-Methoden setzen die Hintergrund- bzw. Vordergrundfarbe einer Komponente:

```

void setBackground(Color c)
void setForeground(Color c)

```

## 27.3 Layouts

Die Anordnung von Komponenten in einem Container erfolgt mit Hilfe von *Layout-Managern*.

Es gibt verschiedene Layout-Manager, denen jeweils ein anderes Konzept zu Grunde liegt. Allen gemeinsam ist, dass eine Platzierung durch Angabe der genauen Pixelwerte nicht erforderlich ist. Ein Vorteil ist die automatische Anpassung der Größe der Komponenten bei Verkleinerung bzw. Vergrößerung des sie enthaltenden Containers.

## Die Methode

```
void setLayout(LayoutManager manager)
```

der Klasse `Container` legt den Layout-Manager für den Container fest. Alle Layout-Manager implementieren das Interface `java.awt.LayoutManager`.

## Dimensionierung von Komponenten

Die folgenden `JComponent`-Methoden geben einem Layout-Manager Hinweise für die Größe einer darzustellenden Komponente:

```
void setPreferredSize(Dimension preferredSize)
```

legt die bevorzugte Größe der Komponente fest.

```
void setMaximumSize(Dimension maximumSize)
```

legt die maximale Größe der Komponente fest.

```
void setMinimumSize(Dimension minimumSize)
```

legt die Mindestgröße der Komponente fest.

Ob diese Größenangaben berücksichtigt werden, hängt vom jeweilig verwendeten Layout-Manager ab. `FlowLayout` und `BoxLayout` richten sich nach diesen Vorgaben, `BorderLayout` und `GridLayout` jedoch nicht (siehe weiter unten).

## Die `java.awt.Window`-Methode

```
void pack()
```

bewirkt, dass die Größe des Fensters an die bevorzugte Größe und das Layout seiner Unterkomponenten angepasst wird.

Alle Beispiele dieses Unterkapitels nutzen Objekte der Klasse `MyPanel` für die einzufügenden Komponenten.

Die Klassen, die im Folgenden die verschiedenen Layout-Manager enthalten, sind von der abstrakten Klasse `BaseLayout` abgeleitet, die den allgemeinen Rahmen vorgibt.

```
package layouts;

import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {
    private final int id;

    public MyPanel(int id) {
        this.id = id;
        setBackground(Color.LIGHT_GRAY);
        setPreferredSize(new Dimension(50, 50));
    }
}
```

```
        setMaximumSize(new Dimension(50, 50));
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawString(String.valueOf(id), 5, 15);
    }
}

package layouts;

import javax.swing.*;
import java.awt.*;

public abstract class BaseLayout extends JFrame {
    public BaseLayout(String title) {
        super(title);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);
        setContent(contentPane);
        setSize(300, 200);
        setVisible(true);
    }

    public abstract void setContent(Container container);
}
```

## Null-Layout

Ein sogenanntes *Null-Layout* wird durch den Aufruf von `setLayout(null)` im Container erzeugt. Es wird kein Layout-Manager verwendet.

Alle Komponenten werden dann mit Hilfe der Component-Methoden `setLocation` und `setSize` oder einfacher mit `setBounds` positioniert:

```
void setBounds(int x, int y, int width, int height)

package layouts;

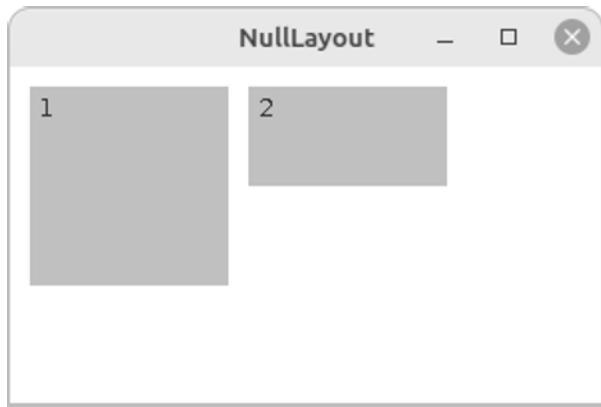
import java.awt.*;

public class NullLayoutTest extends BaseLayout {
    public NullLayoutTest() {
        super("NullLayout");
    }

    @Override
    public void setContent(Container container) {
        container.setLayout(null);
        MyPanel p1 = new MyPanel(1);
        MyPanel p2 = new MyPanel(2);
        p1.setBounds(10, 10, 100, 100);
        p2.setBounds(120, 10, 100, 50);
```

```
        container.add(p1);
        container.add(p2);
    }

    public static void main(String[] args) {
        new NullLayoutTest();
    }
}
```



**Abbildung 27-5:** Null-Layout

## FlowLayout

`java.awt.FlowLayout` ist der Standard-Layout-Manager für Objekte der Klasse `JPanel`. Er ordnet die Komponenten zeilenweise von oben links nach unten rechts an. Passt eine Komponente nicht mehr in eine Zeile (z. B. nach Verkleinerung des Containers), so wird sie automatisch in der nächsten Zeile angeordnet.

Im Konstruktor kann die Anordnung in einer Zeile angegeben werden: `CENTER` (zentriert), `LEFT` (linksbündig), `RIGHT` (rechtsbündig). Als Voreinstellung wird `CENTER` verwendet. Ebenso können der horizontale und der vertikale Abstand zwischen den Komponenten eingestellt werden. Die Voreinstellung ist 5.

```
package layouts;

import java.awt.*;

public class FlowLayoutTest extends BaseLayout {
    public FlowLayoutTest() {
        super("FlowLayout");
    }

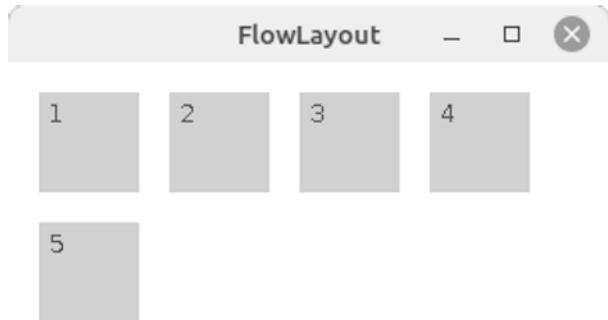
    @Override
    public void setContent(Container container) {
```

```

        container.setLayout(new FlowLayout(FlowLayout.LEFT, 15, 15));
        for (int i = 1; i <= 5; i++)
            container.add(new MyPanel(i));
    }

    public static void main(String[] args) {
        new FlowLayoutTest();
    }
}

```



**Abbildung 27-6:** FlowLayout

## BorderLayout

`java.awt.BorderLayout` ist der Standard-Layout-Manager für den *Content Pane* von `JFrame` und `JDialog`. Er ordnet maximal fünf Komponenten an den vier Seiten und im Zentrum des Containers an.

Die Platzierung einer Komponente mit `add(comp)` erfolgt grundsätzlich im Zentrum. Mit `add(comp, pos)` wird die Komponente im Bereich `pos` platziert, wobei für `pos` eine der Konstanten `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER` der Klasse `BorderLayout` stehen muss.

Im Konstruktor kann der Abstand zwischen den Komponenten angegeben werden.

```

package layouts;

import java.awt.*;

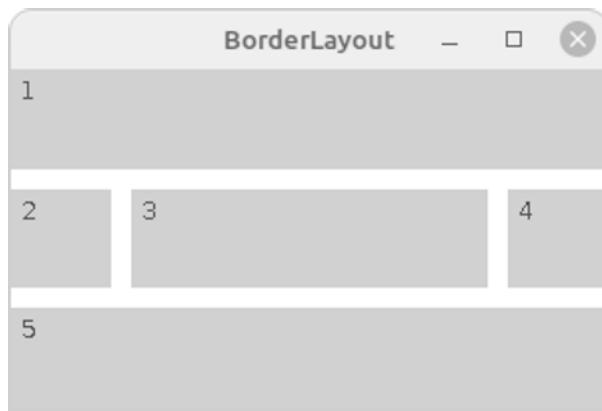
public class BorderLayoutTest extends BaseLayout {
    public BorderLayoutTest() {
        super("BorderLayout");
    }

    @Override
    public void setContent(Container container) {

```

```
container.setLayout(new BorderLayout(10, 10));
container.add(new MyPanel(1), BorderLayout.NORTH);
container.add(new MyPanel(2), BorderLayout.WEST);
container.add(new MyPanel(3), BorderLayout.CENTER);
container.add(new MyPanel(4), BorderLayout.EAST);
container.add(new MyPanel(5), BorderLayout.SOUTH);
}

public static void main(String[] args) {
    new BorderLayoutTest();
}
}
```



**Abbildung 27-7:** BorderLayout

### GridLayout

`java.awt.GridLayout` ordnet die Komponenten in einem Raster aus Zeilen gleicher Höhe und Spalten gleicher Breite an.

Konstruktoren erzeugen ein Layout, bei dem Komponenten in Zeilen und Spalten der Reihe nach verteilt sind. Zusätzlich kann der Abstand zwischen den Komponenten angegeben werden.

```
package layouts;

import java.awt.*;

public class GridLayoutTest extends BaseLayout {
    public GridLayoutTest() {
        super("GridLayout");
    }

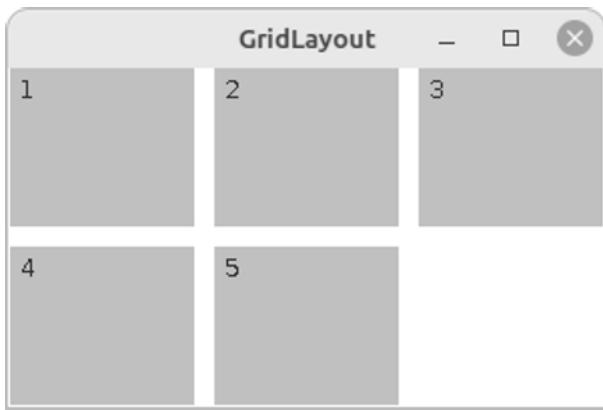
    @Override
    public void setContent(Container container) {
        container.setLayout(new GridLayout(2, 3, 10, 10));
    }
}
```

```

        for (int i = 1; i <= 5; i++)
            container.add(new MyPanel(i));
    }

    public static void main(String[] args) {
        new GridLayoutTest();
    }
}

```



**Abbildung 27-8:** GridLayout

## BoxLayout

Mit `javax.swing.BoxLayout` können alle Komponenten entweder in horizontaler oder in vertikaler Richtung angeordnet werden. Es wird dabei nie mehr als eine Zeile bzw. eine Spalte angelegt. Die Komponenten können unterschiedlich viel Platz belegen.

Der Konstruktor

```
BoxLayout(Container c, int axis)
```

erzeugt das Layout für den Container `c` mit der Richtung `axis`. Gültige Werte für `axis` sind die `BoxLayout`-Konstanten `X_AXIS` und `Y_AXIS`.

## Box

Die Klasse `javax.swing.Box` mit dem Konstruktor `Box(int axis)` ist ein Container, der das `BoxLayout` nutzt.

Die `Box`-Methoden

```
static Box createHorizontalBox()
static Box createVerticalBox()
```

liefern eine `Box` mit horizontaler bzw. vertikaler Ausrichtung.

```
static Component createHorizontalStrut(int width)
static Component createVerticalStrut(int height)
    liefern einen festen Zwischenraum der angegebenen Größe.

static Component createHorizontalGlue()
static Component createVerticalGlue()
    liefern eine leere Komponente, die sich horizontal bzw. vertikal ausdehnen
    kann.
```

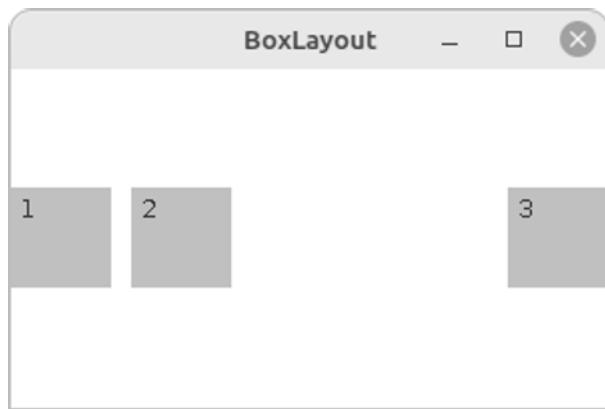
```
package layouts;

import javax.swing.*;
import java.awt.*;

public class BoxLayoutTest extends BaseLayout {
    public BoxLayoutTest() {
        super("BoxLayout");
    }

    @Override
    public void setContent(Container container) {
        container.setLayout(new BoxLayout(container, BoxLayout.X_AXIS));
        container.add(new MyPanel(1));
        container.add(Box.createHorizontalStrut(10));
        container.add(new MyPanel(2));
        container.add(Box.createHorizontalGlue());
        container.add(new MyPanel(3));
    }

    public static void main(String[] args) {
        new BoxLayoutTest();
    }
}
```



**Abbildung 27-9:** BoxLayout

## GridBagLayout

Im Unterschied zum GridLayout kann `java.awt.GridBagLayout` Zellen unterschiedlicher Größe für die Anordnung der Komponenten nutzen. Dabei wird eine Instanz der Klasse `java.awt.GridBagConstraints` verwendet, die Angaben zur Positionierung einer Komponente enthält.

Durch Aufruf der `GridBagLayout`-Methode

```
public void setConstraints(Component c, GridBagConstraints constr)
```

werden diese Angaben für eine Komponente übernommen.

`GridBagConstraints` besitzt die folgenden public-Attribute:

`java.awt.Insets insets`

legt den Pixelabstand der Komponente vom Zellenrand oben, links, unten und rechts fest. Standardwert ist `new Insets(0,0,0,0)`.

`int gridx`

legt die horizontale Position der Zelle fest, in der die Komponente beginnen soll. Die erste Zelle einer Zeile hat den Wert `0`. `GridBagConstraints.RELATIVE` (Standardwert) kennzeichnet, dass die Komponente die nächste freie Zelle in horizontaler Richtung belegen soll.

`int gridy`

legt die vertikale Position der Zelle fest, in der die Komponente beginnen soll. Die oberste Zelle einer Spalte hat den Wert `0`. `GridBagConstraints.RELATIVE` (Standardwert) kennzeichnet, dass die Komponente die nächste freie Zelle in vertikaler Richtung belegen soll.

`int gridwidth`

legt fest, wie viele Zellen die Komponente in einer Zeile belegen soll. Standardwert ist `1`. Der Wert `GridBagConstraints.REMAINDER` kennzeichnet, dass die Komponente die letzte in einer Zeile ist. Der Wert `GridBagConstraints.RELATIVE` kennzeichnet, dass die Komponente die nächste nach der letzten in der Zeile ist.

`int gridheight`

legt fest, wie viele Zellen die Komponente in einer Spalte belegen soll. Standardwert ist `1`. Der Wert `GridBagConstraints.REMAINDER` kennzeichnet, dass die Komponente die letzte in einer Spalte ist. Der Wert `GridBagConstraints.RELATIVE` kennzeichnet, dass die Komponente die nächste nach der letzten in der Spalte ist.

`int fill`

legt fest, wie die Größe der Komponente an den zur Verfügung stehenden Platz angepasst werden soll. Mögliche Werte sind die `GridBagConstraints`-Konstanten: `NONE`, `HORIZONTAL`, `VERTICAL` und `BOTH`. Standardwert ist `NONE`.

```
int anchor
```

legt fest, wo die Komponente innerhalb ihrer Zelle platziert werden soll, wenn sie kleiner als der zu Verfügung stehende Platz ist. Mögliche Werte sind die GridBagConstraints-Konstanten: CENTER, NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST und NORTHWEST. Standardwert ist CENTER.

```
double weightx
```

legt fest, wie der gesamte restliche horizontale Platz auf alle Zellen einer Zeile anteilig verteilt werden kann, wenn z. B. das Fenster vergrößert wird. Der Platzanteil einer Zelle errechnet sich nach der Formel: (weightx der Zelle) / (Summe von weightx aller Zellen). Standardwert ist 0.

```
double weighty
```

legt fest, wie der gesamte restliche vertikale Platz auf alle Zellen einer Spalte anteilig verteilt werden kann, wenn z. B. das Fenster vergrößert wird. Der Platzanteil einer Zelle errechnet sich nach der Formel: (weighty der Zelle) / (Summe von weighty aller Zellen). Standardwert ist 0.

```
int ipadx
```

vergrößert die Komponente an der linken und rechten Seite. Standardwert ist 0.

```
int ipady
```

vergrößert die Komponente an der oberen und unteren Seite. Standardwert ist 0.

```
package layouts;
```

```
import java.awt.*;
```

```
public class GridBagLayoutTest extends BaseLayout {
    public GridBagLayoutTest() {
        super("GridBagLayout");
    }

    private void addPanel(GridBagLayout gridbag, MyPanel p,
                          int x, int y, int w, int h) {

        GridBagConstraints constr = new GridBagConstraints();
        constr.insets = new Insets(2, 2, 2, 2);
        constr.gridx = x;
        constr.gridy = y;
        constr.gridwidth = w;
        constr.gridheight = h;
        constr.fill = GridBagConstraints.BOTH;
        constr.weightx = 1;
        constr.weighty = 1;
        gridbag.setConstraints(p, constr);
        add(p);
    }

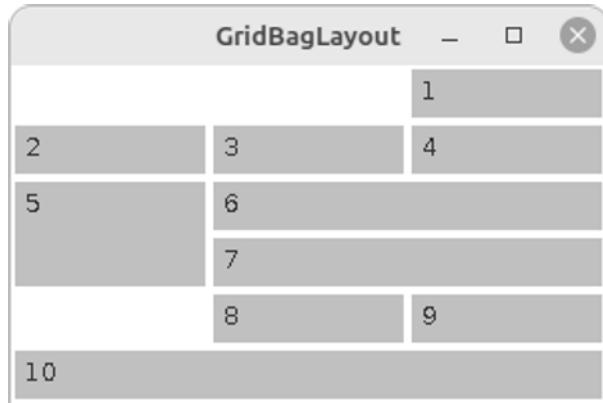
    @Override
    public void setContent(Container container) {
        MyPanel[] p = new MyPanel[10];
    }
}
```

```
for (int i = 0; i < p.length; i++)
    p[i] = new MyPanel(i + 1);

GridLayout gridbag = new GridLayout();
container.setLayout(gridbag);

addPanel(gridbag, p[0], 2, 0, 1, 1);
addPanel(gridbag, p[1], 0, 1, 1, 1);
addPanel(gridbag, p[2], 1, 1, 1, 1);
addPanel(gridbag, p[3], 2, 1, 1, 1);
addPanel(gridbag, p[4], 0, 2, 1, 2);
addPanel(gridbag, p[5], 1, 2, 2, 1);
addPanel(gridbag, p[6], 1, 3, 2, 1);
addPanel(gridbag, p[7], 1, 4, 1, 1);
addPanel(gridbag, p[8], 2, 4, 1, 1);
addPanel(gridbag, p[9], 0, 5, 3, 1);
}

public static void main(String[] args) {
    new GridBagLayoutTest();
}
}
```



**Abbildung 27-10:** GridBagLayout

### Geschachtelte Layouts

Container können ineinander geschachtelt werden. Das folgende Programm ordnet zwei JPanel-Objekte gemäß dem BorderLayout an. Die Komponenten des ersten Panels werden gemäß dem GridLayout, die Komponenten des zweiten Panels gemäß dem FlowLayout platziert.

Eine Komponente kann mit einem Rand ausgestattet werden. Der Konstruktor der Klasse javax.swing.border.EmptyBorder liefert einen transparenten Rand mit Abständen: `EmptyBorder(int top, int left, int bottom, int right)`

Der Konstruktor der Klasse `javax.swing.border.TitledBorder` liefert einen Rand mit Titel: `TitledBorder(String title)`

Schrift und Farbe können eingestellt werden.

Die beiden Klassen implementieren das Interface `javax.swing.border.Border`. Mit der `JComponent`-Methode `setBorder` kann der Rand für die Komponente festgelegt werden.

```
package layouts;

import javax.swing.*;
import javax.swing.border.TitledBorder;
import java.awt.*;

public class NestedLayouts extends JFrame {
    public NestedLayouts() {
        super("NestedLayouts");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel p1 = new JPanel();
        p1.setBackground(Color.WHITE);
        p1.setLayout(new GridLayout(2, 3, 5, 5));
        for (int i = 1; i <= 6; i++)
            p1.add(new MyPanel(i));

        TitledBorder b1 = new TitledBorder("Panel 1");
        b1.setTitleFont(new Font("Dialog", Font.PLAIN, 12));
        b1.setTitleColor(Color.blue);
        p1.setBorder(b1);

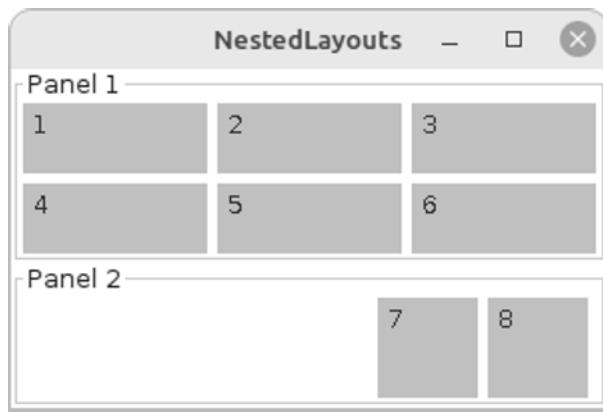
        JPanel p2 = new JPanel();
        p2.setBackground(Color.WHITE);
        p2.setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 0));
        p2.add(new MyPanel(7));
        p2.add(new MyPanel(8));

        TitledBorder b2 = new TitledBorder("Panel 2");
        b2.setTitleFont(new Font("Dialog", Font.PLAIN, 12));
        b2.setTitleColor(Color.BLUE);
        p2.setBorder(b2);

        add(p1, BorderLayout.CENTER);
        add(p2, BorderLayout.SOUTH);

        setSize(300, 200);
        setVisible(true);
    }

    public static void main(String[] args) {
        new NestedLayouts();
    }
}
```



**Abbildung 27-11:** Geschachtelte Layouts

## 27.4 Buttons und Labels

Es gibt verschiedene Arten von Buttons: `JButton`, `JCheckBox`, `JRadioButton`.

### `JButton`

Die Klasse `javax.swing.JButton` implementiert einen einfachen Button mit den Konstruktoren:

```
JButton()
JButton(String text)
JButton(Icon icon)
JButton(String text, Icon icon)
```

Die Konstruktoren erzeugen einen Button mit oder ohne Beschriftung bzw. Icon.

`javax.swing.Icon` ist ein Interface, das von der Klasse `javax.swing.ImageIcon` implementiert wird.

Ein `ImageIcon`-Objekt kann mit dem Konstruktor `ImageIcon(String filename)` aus einer Bilddatei (GIF-, JPEG- oder PNG-Format) erzeugt werden.

Mit der `JComponent`-Methode

```
void setToolTipText(String text)
```

wird eine Komponente mit einem sogenannten *Tooltip* ausgestattet, der beim Bewegen des Mauszeigers über diese Komponente angezeigt wird.

Es gibt auch Methoden, die die Beschriftung bzw. das Icon festlegen:

```
void setText(String text)
void setIcon(Icon icon)
```

Die beiden folgenden Methoden stammen aus der Klasse `Component` und sind für alle Komponenten aufrufbar:

`void setEnabled(boolean b)`

deaktiviert die Komponente, wenn `b` den Wert `false` hat. und schließt sie von der Ereignisbehandlung aus.

`boolean isEnabled()`

liefert den Wert `true`, wenn die Komponente aktiviert ist.

### ActionListener

Wird auf einen Button geklickt, so werden alle bei diesem Button registrierten `ActionListener` aufgerufen.

Das Funktionsinterface `java.awt.event.ActionListener` hat die Methode

`void actionPerformed(ActionEvent e)`

Dieser Listener kann beim Button mit

`void addActionListener(ActionListener l)`

registriert werden.

Die Methode `String getActionCommand()` der Klasse `java.awt.event.ActionEvent` liefert den Kommandonamen (standardmäßig die Beschriftung des Buttons). Bei mehreren Buttons kann man hiermit feststellen, welcher Button gedrückt wurde.

### JCheckBox und JRadioButton

Die speziellen Buttons `javax.swing.JCheckBox` und `javax.swing.JRadioButton` können jeweils den Zustand "selektiert" oder "nicht selektiert" annehmen. Neben den zu `JButton` analogen Konstruktoren existieren für `JCheckBox` und `JRadioButton` je drei weitere mit einem zusätzlichen Parameter vom Typ `boolean`, der angibt, ob der Button "selektiert" (`true`) sein soll.

Mit `void setSelected(boolean b)` wird der Button selektiert, wenn `b` den Wert `true` hat.

`boolean isSelected()` liefert `true`, wenn der Button selektiert ist.

Mehrere Objekte vom Typ `JRadioButton` können mit Hilfe eines Objekts der Klasse `javax.swing.ButtonGroup` zu einer Gruppe zusammengefasst werden.

Von den Buttons, die der gleichen Gruppe angehören, kann nur einer den Zustand "selektiert" haben.

Das Selektieren eines neuen Buttons der Gruppe führt automatisch zum Deselek- tieren des bisher selektierten Buttons.

```
void add(AbstractButton b) nimmt den Button b in die Gruppe auf.
```

### ItemListener

Neben der Erzeugung eines ActionEvent löst das Selektieren und Deselektieren einer Check-Box oder eines Radio-Buttons ein ItemEvent aus.

Das Funktionsinterface `java.awt.event.ItemListener` hat die Methode

```
void itemStateChanged(ItemEvent e)
```

Dieser Listener wird mit

```
void addItemListener(ItemListener l)
```

registriert.

Die `java.awt.event.ItemEvent`-Methode

```
int getStateChange()
```

liefert die ItemEvent-Konstante `SELECTED`, wenn der Button ausgewählt wurde, oder `DESELECTED`, wenn die Auswahl aufgehoben wurde.

### JLabel

Die Klasse `javax.swing.JLabel` kann einen Text und/oder ein Bild anzeigen.

Konstruktoren sind:

```
JLabel()  
JLabel(String text)  
JLabel(String text, int align)  
JLabel(Icon icon)  
JLabel(Icon icon, int align)  
JLabel(String text, Icon icon, int align)
```

Diese Konstruktoren erzeugen ein Label mit oder ohne Beschriftung bzw. Icon. `align` bestimmt die horizontale Ausrichtung des Inhalts auf der verfügbaren Fläche mit einer der Konstanten `LEFT`, `CENTER` oder `RIGHT`.

Es existieren die folgenden Eigenschaften mit den entsprechenden get- und set-Methoden:

<code>text</code>	Text des Labels ( <code>String</code> )
<code>icon</code>	Icon des Labels ( <code>Icon</code> )
<code>iconTextGap</code>	Abstand zwischen Icon und Text in Pixel ( <code>int</code> )
<code>horizontalAlignment</code>	horizontale Ausrichtung des Inhalts auf der verfügbaren Fläche ( <code>JLabel</code> -Konstanten <code>LEFT</code> , <code>CENTER</code> oder <code>RIGHT</code> )

---

verticalAlignment	vertikale Ausrichtung des Inhalts auf der verfügbaren Fläche (JLabel-Konstanten TOP, CENTER oder BOTTOM)
horizontalTextPosition	horizontale Position des Textes relativ zum Icon (JLabel-Konstanten LEFT, CENTER oder RIGHT)
verticalTextPosition	vertikale Position des Textes relativ zum Icon (JLabel-Konstanten TOP, CENTER oder BOTTOM)

### Die JComponent-Methode

```
void setOpaque(boolean b)
```

gibt der Komponente einen deckenden (`true`) oder durchsichtigen (`false`) Hintergrund.

Das folgende Programm demonstriert das Zeichnen eines Rechtecks oder Ovalen in verschiedenen Farben.

```
package buttons;

import javax.swing.*;
import java.awt.*;

public class ButtonTest extends JFrame {
    private final JCheckBox rectangle;
    private final JRadioButton red;
    private final JRadioButton yellow;
    private final JRadioButton green;

    public ButtonTest() {
        super("ButtonTest");
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        // Label mit Icon
        Icon icon = new ImageIcon(getClass().getResource("java.png"));
        JLabel label = new JLabel("Java", icon, JLabel.CENTER);
        label.setIconTextGap(20);
        label.setForeground(Color.BLUE);
        label.setFont(new Font("SansSerif", Font.BOLD, 24));
        label.setOpaque(true);

        // Radio Buttons
        red = new JRadioButton("rot", true);
        yellow = new JRadioButton("gelb");
        green = new JRadioButton("grün");

        ButtonGroup bg = new ButtonGroup();
        bg.add(red);
        bg.add(yellow);
        bg.add(green);
```

```
red.addItemListener(e -> repaint());
yellow.addItemListener(e -> repaint());
green.addItemListener(e -> repaint());

// Check Button
rectangle = new JCheckBox("Rechteck", true);
rectangle.addItemListener(e -> repaint());

// Button mit ActionListener
JButton button = new JButton("Beenden");
button.addActionListener(e -> dispose());

JPanel panel = new JPanel();
panel.add(red);
panel.add(yellow);
panel.add(green);
panel.add(rectangle);
panel.add(button);

add(panel, BorderLayout.SOUTH);
add(new MyCanvas(), BorderLayout.CENTER);
add(label, BorderLayout.NORTH);

setSize(500, 400);
setVisible(true);
}

private class MyCanvas extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        if (red.isSelected())
            g.setColor(Color.RED);
        else if (yellow.isSelected())
            g.setColor(Color.YELLOW);
        else if (green.isSelected())
            g.setColor(Color.GREEN);

        int w = getSize().width;
        int h = getSize().height;

        if (rectangle.isSelected())
            g.fillRect(10, 10, w - 20, h - 20);
        else
            g.fillOval(10, 10, w - 20, h - 20);
    }
}

public static void main(String[] args) {
    new ButtonTest();
}
}
```



**Abbildung 27-12:** Buttons und Label

Es ist darauf zu achten, dass *java.png* für die Ausführung des Programms im Verzeichnis des Bytecodes liegt.

## 27.5 Spezielle Container

In diesem Kapitel werden zwei spezielle Container vorgestellt:

- |                          |                                             |
|--------------------------|---------------------------------------------|
| <code>JScrollPane</code> | ein Container mit Scrollbalken              |
| <code>JTabbedPane</code> | ein Container für sogenannte Registerkarten |

### `JScrollPane`

Die Klasse `javax.swing.JScrollPane` repräsentiert einen Container mit horizontalem und vertikalem Scrollbalken, der nur eine Komponente aufnehmen kann. Mit Hilfe der Scrollbalken kann die Komponente im Bildausschnitt verschoben werden, falls die Komponente aufgrund ihrer Größe nicht vollständig sichtbar ist.

Scroll-Panes werden erzeugt mit:

```
JScrollPane(Component c)
JScrollPane(Component c, int v, int h)
```

`c` ist die darzustellende Komponente. `v` und `h` legen fest, ob und wann die Scrollbalken angezeigt werden sollen.

Gültige Werte für v sind die Konstanten:

VERTICAL\_SCROLLBAR\_AS\_NEEDED (Voreinstellung), VERTICAL\_SCROLLBAR\_NEVER und VERTICAL\_SCROLLBAR\_ALWAYS.

Gültige Werte für h sind die Konstanten:

HORIZONTAL\_SCROLLBAR\_AS\_NEEDED (Voreinstellung), HORIZONTAL\_SCROLLBAR\_NEVER und HORIZONTAL\_SCROLLBAR\_ALWAYS.

```
package container;

import javax.swing.*;

public class ScrollPaneTest extends JFrame {
    public ScrollPaneTest() {
        super("ScrollPaneTest");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel image = new JLabel(new ImageIcon(getClass()
            .getResource("wurst.jpg")));
        JScrollPane pane = new JScrollPane(image);
        add(pane);

        setSize(300, 200);
        setVisible(true);
    }

    public static void main(String[] args) {
        new ScrollPaneTest();
    }
}
```



**Abbildung 27-13:** JScrollPane

### JTabbedPane

`javax.swing.JTabbedPane` ist ein Container aus mehreren "Karten" (Tabs), die wie in einem Stapel übereinander gelegt und abwechselnd sichtbar gemacht werden können.

Die Konstruktoren

```
JTabbedPane()  
JTabbedPane(int placement)  
JTabbedPane(int placement, int policy)
```

erzeugen jeweils ein leeres Karteiregister.

`placement` bestimmt den Ort der Registerlaschen. Gültige Werte sind die Konstanten `TOP`, `BOTTOM`, `LEFT` und `RIGHT`. `policy` bestimmt die Darstellung, wenn nicht alle Registerlaschen im Fenster angezeigt werden können. Gültige Werte sind die Konstanten `WRAP_TAB_LAYOUT` und `SCROLL_TAB_LAYOUT`.

Folgende Methoden fügen neue Karten hinzu:

```
void addTab(String title, Component c)  
void addTab(String title, Icon icon, Component c)  
void addTab(String title, Icon icon, Component c, String tip)
```

Bei den beiden letzten Methoden kann auch der Titel oder das Icon `null` sein. `tip` ist ein Tooltip-Text.

```
void insertTab(String title, Icon icon, Component c, String tip, int pos)  
fügt die Komponente c an der Position pos ein. Titel oder Icon können auch  
null sein.
```

```
void removeTabAt(int pos)  
löscht die Komponente an der Position pos.
```

Es existieren die folgenden Eigenschaften mit den entsprechenden get- und set-Methoden:

<code>tabLayoutPolicy</code>	bestimmt die Darstellung, wenn nicht alle Registerlaschen im Fenster angezeigt werden können (siehe Konstruktor)
<code>selectedIndex</code>	Position der selektierten Karte (int)
<code>selectedComponent</code>	Komponente der selektierten Karte (Component)
<code>titleAt</code>	Titel der Karte an einer Position (int, String)
<code>iconAt</code>	Icon der Karte an einer Position (int, Icon)
<code>toolTipTextAt</code>	Tooltip-Text der Karte an einer Position (int, String)
<code>enabledAt</code>	Karte aktiviert an einer Position (int, boolean)

### ChangeListener

Die Selektion einer Karte löst ein `javax.swing.event.ChangeEvent` aus.

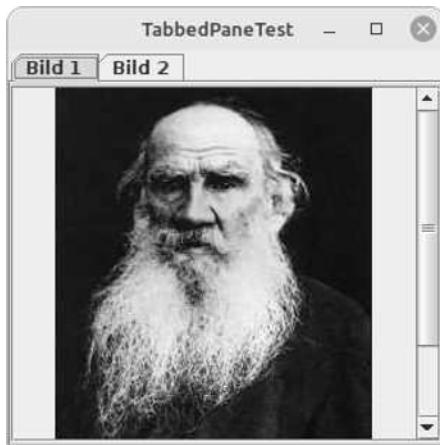
Das Funktionsinterface `javax.swing.event.ChangeListener` hat die Methode

```
void stateChanged(ChangeEvent e)
```

Dieser Listener wird mit

```
void addChangeListener(ChangeListener l)
```

registriert.



**Abbildung 27-14:** JTabbedPane

```
package container;

import javax.swing.*;

public class TabbedPaneTest extends JFrame {
    public TabbedPaneTest() {
        super("TabbedPaneTest");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        String[] tabs = {"Tolstoi", "Boot"};

        JTabbedPane pane = new JTabbedPane(JTabbedPane.TOP);
        JLabel image1 = new JLabel(new ImageIcon(getClass()
            .getResource("tolstoi.jpg")));
        JLabel image2 = new JLabel(new ImageIcon(getClass()
            .getResource("boat.png")));
        JScrollPane pane1 = new JScrollPane(image1);
        JScrollPane pane2 = new JScrollPane(image2);
        pane.addTab("Bild 1", pane1);
        pane.addTab("Bild 2", pane2);
```

```
pane.addChangeListener(e ->
    System.out.println(tabs[pane.getSelectedIndex()])
);
add(pane);

setSize(300, 300);
setVisible(true);

System.out.println(tabs[0]);
}

public static void main(String[] args) {
    new TabbedPaneTest();
}
}
```

## 27.6 Textkomponenten

Die Superklasse `javax.swing.text.JTextComponent` aller Textkomponenten bietet die folgenden Eigenschaften (get-/set-Methoden) und weitere Methoden an:

<code>text</code>	Textinhalt ( <code>String</code> )
<code>selectedText</code>	markierter Text ( <code>String</code> ), nur get-Methode
<code>selectionStart</code>	Position des ersten markierten Zeichens ( <code>int</code> ), nur get-Methode
<code>selectionEnd</code>	Position des letzten markierten Zeichens + 1 ( <code>int</code> ), nur get-Methode
<code>caretPosition</code>	Position der Einfügemarken ( <code>int</code> )
<code>editable</code>	ist editierbar ( <code>boolean</code> )

```
void selectAll()
    markiert den kompletten Text im Textfeld.

void moveCaretPosition(int pos)
    bewegt die Einfügemarken zur Position pos. Der Text ab der beim letzten Aufruf von setCaretPosition gesetzten Position bis pos-1 wird markiert.

void copy()
    kopiert den markierten Text in die Zwischenablage.

void cut()
    kopiert den markierten Text in die Zwischenablage und löscht ihn im Original.

void paste()
    ersetzt den markierten Text durch den Inhalt der Zwischenablage bzw. fügt diesen Inhalt an der aktuellen Position der Einfügemarken ein.
```

```
boolean print() throws java.awt.print.PrinterException
```

öffnet einen Druckdialog zum Drucken des Inhalts der Textkomponente. Der Rückgabewert ist `false`, falls der Druckvorgang vom Benutzer abgebrochen wurde.

### Die `JTextComponent`-Methoden

```
void read(Reader in, Object desc) throws IOException  
void write(Writer out) throws IOException
```

lesen Text in den Textbereich ein bzw. speichern ihn. `desc` beschreibt den Eingabestrom. `desc` kann auch den Wert `null` haben. Beim Einlesen wird ein evtl. schon vorher bestehender Text überschrieben.

## **JTextField**

Ein Objekt der Klasse `javax.swing.JTextField` erlaubt die Eingabe einer Textzeile.

### Die Konstruktoren

```
JTextField()  
JTextField(int cols)  
JTextField(String text)  
JTextField(String text, int cols)
```

erzeugen jeweils ein Textfeld, das den Text `text` enthält und `cols` Zeichen breit ist.

### Eigenschaften (get-/set-Methoden):

<code>columns</code>	Spaltenbreite des Textfelds (int)
<code>horizontalAlignment</code>	horizontale Ausrichtung des Textes ( <code>JTextField</code> -Konstanten <code>LEFT</code> , <code>CENTER</code> oder <code>RIGHT</code> )

## **JPasswordField**

Die Subklasse `javax.swing.JPasswordField` mit den zu `JTextField` analogen Konstruktoren implementiert ein Passwortfeld, in dem anstelle eines eingegebenen Zeichens ein "Echo-Zeichen" angezeigt wird, standardmäßig das Zeichen \*.

### Eigenschaft (get-/set-Methoden):

<code>echoChar</code>	Echo-Zeichen (char)
-----------------------	---------------------

```
char[] getPassword()
```

liefert den Inhalt des Passwortfeldes als `char`-Array.

Aus Sicherheitsgründen sollten nach Verarbeitung des Passworts alle Zeichen des Arrays auf `0` gesetzt werden.

### ActionListener

Wird die *Eingabe-Taste* innerhalb des Textfeldes gedrückt, so erzeugt das Textfeld ein ActionEvent.

Ein Textfeld erlaubt die Registrierung eines ActionListener-Objekts.

Die Methode `String getActionCommand()` der Klasse ActionEvent liefert hier den Inhalt des Textfeldes.

### JTextArea

Ein Objekt der Klasse `javax.swing.JTextArea` erlaubt die Eingabe mehrerer Textzeilen.

Die Konstruktoren

```
JTextArea()  
JTextArea(int rows, int cols)  
JTextArea(String text)  
JTextArea(String text, int rows, int cols)
```

erzeugen jeweils eine Textfläche, die den Text `text` enthält und `rows` sichtbare Zeilen sowie `cols` sichtbare Spalten hat.

Eigenschaften (get-/set-Methoden) und weitere Methoden:

<code>rows</code>	Anzahl Zeilen (int)
<code>columns</code>	Anzahl Spalten (int)
<code>lineWrap</code>	Zeilenumbruch, wenn true (boolean)
<code>wrapStyleWord</code>	Zeilenumbruch auf Wortgrenze, wenn true (boolean)

```
void append(String text)  
fügt text am Ende des bestehenden Textes an.
```

```
void insert(String text, int pos)  
fügt text an der Position pos im bestehenden Text ein.
```

```
void replaceRange(String text, int start, int end)  
ersetzt den Text zwischen start und end-1 durch die Zeichenkette text.
```

Mit dem folgenden Programm kann ein Artikel mit Bezeichnung, Preis und Beschreibung erfasst werden. Die erfassten Daten werden mit "OK" auf der Konsole ausgegeben. Die Eingaben zur Artikelbezeichnung und zum Preis werden geprüft.



Abbildung 27-15: JTextField und JTextArea

```
package text;

import javax.swing.*;
import javax.swing.border.EmptyBorder;
import java.awt.*;
import java.util.ArrayList;

public class Form extends JFrame {
    private final JTextField article;
    private final JTextField price;
    private final JLabel msg;

    public Form() {
        super("Form");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panelTop = new JPanel();
        panelTop.setLayout(new GridLayout(7, 1));
        panelTop.setBorder(new EmptyBorder(10, 10, 0, 10));

        panelTop.add(new JLabel("Artikelbezeichnung"));
        article = new JTextField();
        panelTop.add(article);
        panelTop.add(new JLabel()); // Abstand

        panelTop.add(new JLabel("Preis"));
```

```
price = new JTextField();
price.setHorizontalTextPosition(JTextField.RIGHT);
panelTop.add(price);
panelTop.add(new JLabel()); // Abstand

panelTop.add(new JLabel("Beschreibung"));
JTextArea description = new JTextArea(5, 50);
description.setWrapStyleWord(true);
description.setLineWrap(true);
JScrollPane scrollPane = new JScrollPane(description,
    JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
    JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
scrollPane.setBorder(new EmptyBorder(0, 10, 10, 10));

 JPanel panelBottom = new JPanel();
panelBottom.setLayout(new GridLayout(2, 1));
panelBottom.setBorder(new EmptyBorder(10, 10, 10, 10));

msg = new JLabel();
msg.setForeground(Color.RED);
panelBottom.add(msg);

 JPanel buttons = new JPanel();
 JButton ok = new JButton("OK");
 JButton clear = new JButton("Leeren");
buttons.add(ok);
buttons.add(clear);
panelBottom.add(buttons);

add(panelTop, BorderLayout.NORTH);
add(scrollPane, BorderLayout.CENTER);
add(panelBottom, BorderLayout.SOUTH);

ok.addActionListener(e -> {
    msg.setText("");
    ArrayList<String> messages = validateForm();
    if (messages.isEmpty()) {
        System.out.println(article.getText());
        System.out.println(price.getText());
        System.out.println(description.getText());
    } else {
        msg.setText("<html><body>" + String.join("<br>", messages)
            + "</html></body>");
    }
});

clear.addActionListener(e -> {
    msg.setText("");
    article.setText("");
    price.setText("");
    description.setText("");
});

setSize(400, 400);
setVisible(true);
}

public ArrayList<String> validateForm() {
    ArrayList<String> messages = new ArrayList<>();
```

```

        if (article.getText().isBlank()) {
            messages.add("Artikelbezeichnung fehlt");
        }
        try {
            Double.parseDouble(price.getText().trim());
        } catch (NumberFormatException e) {
            messages.add("Preis fehlt");
        }
    }
    return messages;
}

public static void main(String[] args) {
    new Form();
}
}

```

## 27.7 Auswahlkomponenten

Einzelige (`JComboBox`) und mehrzeilige Listenfelder (`JList`) sind Komponenten, die die Auswahl eines Eintrags aus einer Liste ermöglichen.

Die Auswahl einer Zahl aus einem Intervall geschieht mit Hilfe eines Schiebers (`JSlider`).

### `JComboBox`

Die Klasse `javax.swing.JComboBox<T>` realisiert ein Feld, das durch Mausklick eine Liste aufklappt, aus der ein Eintrag ausgewählt werden kann.

Die Konstruktoren

```

JComboBox()
JComboBox(T[] items)
JComboBox(Vector<T> items)

```

erzeugen jeweils ein Feld mit leerer Liste, Elementen aus einem Array bzw. aus einem Vektor.

Zum Einfügen und Löschen können die folgenden Methoden benutzt werden:

```

void addItem(Object item)
void insertItemAt(Object item, int pos)
void removeItemAt(int pos)
void removeItem(Object item)
void removeAllItems()

```

Eigenschaften (get-/set-Methoden):

<code>itemAt</code>	Eintrag an einer Position ( <code>int</code> ), nur get-Methode
<code>itemCount</code>	Anzahl der Einträge ( <code>int</code> ), nur get-Methode
<code>selectedIndex</code>	Position des ausgewählten Eintrags ( <code>int</code> )

---

<code>selectedItem</code>	ausgewählter Eintrag ( <code>Object</code> )
<code>maximumRowCount</code>	maximale Anzahl von sichtbaren Einträgen ( <code>int</code> )
<code>editable</code>	Eintrag ist änderbar ( <code>boolean</code> )

**ComboBoxEditor getEditor()**

liefert den Editor für das Feld. Mit der `javax.swing.ComboBoxEditor`-Methode `Object getItem()` kann dann die Eingabe abgefragt werden.

**ActionListener und ItemListener**

Nach der Auswahl eines Eintrags wird ein `ActionEvent` ausgelöst. Beim Wechsel des selektierten Eintrags wird ein `ItemEvent` ausgelöst. Die entsprechenden Listener können registriert werden.

**JList**

Die Klasse `javax.swing.JList<T>` ermöglicht die Auswahl von einem oder mehreren Einträgen aus einer Liste, die mit einem Sichtfenster einer bestimmten Größe versehen ist.

**Die Konstruktoren**

```
JList()
JList(T[] items)
JList(Vector<? extends T> items)
```

erzeugen jeweils eine leere Liste bzw. eine Liste mit Elementen aus einem Array oder aus einem Vektor. Es werden die mittels `toString()` erzeugten Strings der Objekte in der Liste angezeigt.

**Eigenschaften (get-/set-Methoden) und weitere Methoden:**

<code>visibleRowCount</code>	Anzahl Zeilen der Liste, die ohne Scrollbalken angezeigt werden ( <code>int</code> )
<code>selectionMode</code>	bestimmt, ob ein Eintrag (einfacher Mausklick) oder mehrere Einträge (Control-Taste und Mausklick bzw. Shift-Taste und Mausklick) ausgewählt werden können, <code>javax.swing.ListSelectionModel</code> -Konstanten: <code>SINGLE_SELECTION</code> , <code>SINGLE_INTERVAL_SELECTION</code> und <code>MULTIPLE_INTERVAL_SELECTION</code> (Voreinstellung)
<code>selectionBackground</code>	Hintergrundfarbe für ausgewählte Einträge ( <code>Color</code> )
<code>selectionForeground</code>	Vordergrundfarbe für ausgewählte Einträge ( <code>Color</code> )
<code>selectedIndex</code>	Position des ersten ausgewählten Eintrags ( <code>int</code> )

`selectedIndices` Positionen der ausgewählten Einträge (`int[]`)

`void setListData(T[] items)`

`void setListData(Vector<? extends T> items)`

füllt die Liste mit den Elementen aus dem Array bzw. aus dem Vektor `items`.

`void ensureIndexIsVisible(int pos)`

stellt sicher, dass der Eintrag an der Position `pos` sichtbar ist, wenn die Liste mit einem Scrollbalken versehen ist. Die Komponente muss hierfür bereits sichtbar sein.

`void clearSelection()`

hebt die aktuelle Auswahl in der Liste auf.

`boolean isSelectedIndex(int pos)`

liefert den Wert `true`, wenn der Eintrag an der Position `pos` ausgewählt ist.

`boolean isSelectionEmpty()`

liefert den Wert `true`, wenn nichts ausgewählt wurde.

`T getSelectedValue()`

liefert den ersten ausgewählten Eintrag oder `null`, wenn kein Eintrag ausgewählt wurde.

`List<T> getSelectedValuesList()`

liefert die ausgewählten Einträge.

## JSlider

`javax.swing.JSlider` ermöglicht die Auswahl einer ganzen Zahl aus einem Intervall mit Hilfe eines Schiebers.

Der Konstruktor

`JSlider(int orientation, int min, int max, int value)`

legt mit `orientation` die Ausrichtung des Schiebers fest: `JSlider.HORIZONTAL` oder `JSlider.VERTICAL`. Das Intervall ist durch `min` und `max` begrenzt. `value` ist der Startwert.

Eigenschaften (get-/set-Methoden):

`majorTickSpacing` Abstand zwischen großen Strichen auf der Skala (`int`)

`minorTickSpacing` Abstand zwischen kleinen Strichen auf der Skala (`int`)

`paintTicks` Skalenstriche werden angezeigt (`boolean`)

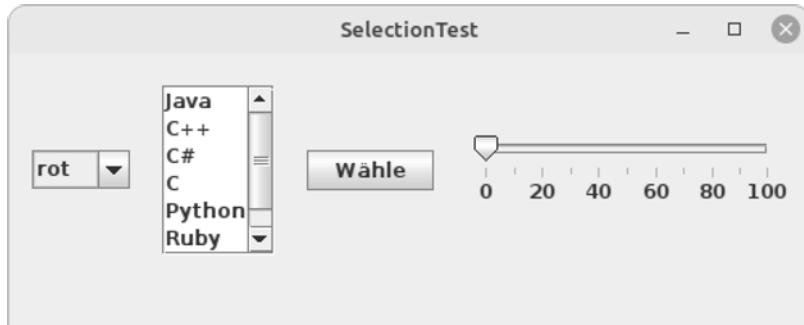
`paintLabels` Beschriftungen werden angezeigt (`boolean`)

`valueIsAdjusting` Schieber wird bewegt (`boolean`)

`value` aktueller Wert (`int`)

### ChangeListener

Das Bewegen des Schiebers löst ein ChangeEvent aus. Vergleiche die Ausführungen zu JTabbedPane in Kapitel 27.5.



**Abbildung 27-16:** JComboBox, JList und JSlider

```
package selection;

import javax.swing.*;
import java.awt.*;

public class SelectionTest extends JFrame {
    public SelectionTest() {
        super("SelectionTest");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = getContentPane();
        container.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));

        String[] comboBoxItems = {"rot", "gelb", "grün"};
        JComboBox<String> comboBox = new JComboBox<>(comboBoxItems);
        comboBox.setSelectedIndex(0);
        container.add(comboBox);

        comboBox.addActionListener(e ->
            System.out.println(comboBox.getSelectedItem())
        );

        String[] listItems = {"Java", "C++", "C#", "C", "Python", "Ruby", "PHP"};
        JList<String> list = new JList<>(listItems);
        list.setVisibleRowCount(6);
        container.add(new JScrollPane(list));

        JButton button = new JButton("Wähle");
        button.addActionListener(e -> {
            if (!list.isSelectionEmpty()) {
                int[] idx = list.getSelectedIndices();
                for (int value : idx)
                    System.out.println(listItems[value]);
            }
        });
    }
}
```

```
        list.clearSelection();
    }
});

container.add(button);

JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 0);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(10);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
slider.addChangeListener(e -> {
    if (!slider.getValueIsAdjusting())
        System.out.println(slider.getValue());
});
container.add(slider);

setSize(500, 200);
setVisible(true);
}

public static void main(String[] args) {
    new SelectionTest();
}
}
```

## 27.8 Menüs und Symbolleisten

Fenster können *Menüs* enthalten. Ihre Erzeugung wird durch spezielle Klassen des Pakets `javax.swing` unterstützt.

Die Klasse `JMenuBar` stellt die Menüleiste des Fensters dar, die Klasse `JMenu` ein einzelnes der darin enthaltenen Menüs.

Die Klassen `JMenuItem`, `JCheckBoxMenuItem` und `JRadioButtonMenuItem` implementieren die Menüeinträge eines Menüs.

Die Klassen `JFrame` und `JDialog` besitzen die Methode

```
void setJMenuBar(JMenuBar bar),
```

die die Menüleiste bar einbindet.

## JMenuBar

## Der Konstruktor

### JMenuBar()

erzeugt eine leere Menüleiste.

JMenu add(JMenu m)

fügt das Menü `m` der Menüleiste am Ende hinzu.

```
void remove(Component m)
    entfernt das Menü m aus der Menüleiste.
```

```
JMenu getMenu(int pos)
    liefert das Menü an der Position pos.
```

## JMenu

Die Konstruktoren

```
JMenu()
JMenu(String text)
```

erzeugen ein Menü ohne bzw. mit Titel. JMenu ist Subklasse von JMenuItem.

```
JMenuItem add(JMenuItem mi)
    fügt den Menüeintrag mi dem Menü am Ende hinzu. Dabei kann mi selbst ein
    Menü sein (Untermenü).
```

```
JMenuItem insert(JMenuItem mi, int pos)
    fügt einen Menüeintrag an der Position pos in das Menü ein.
```

```
void addSeparator()
    fügt eine Trennlinie dem Menü hinzu.
```

```
void insertSeparator(int pos)
    fügt eine Trennlinie an der Position pos in das Menü ein.
```

```
void remove(int pos)
void remove(JMenuItem mi)
void removeAll()
    entfernen den Menüpunkt an der Position pos, den Menüpunkt mi bzw. alle
    Menüpunkte aus dem Menü.
```

```
JMenuItem getItem(int pos)
    liefert den Menüeintrag an der Position pos.
```

## JMenuItem

Alle Einträge in einem Menü gehören zur Klasse JMenuItem.

Die Konstruktoren

```
JMenuItem()
JMenuItem(Icon icon)
JMenuItem(String text)
JMenuItem(String text, Icon icon)
```

erzeugen einen Menüeintrag mit oder ohne Beschriftung bzw. Icon.

Da JMenuItem Subklasse von AbstractButton ist, stehen auch die Methoden dieser Superklasse zur Verfügung, insbesondere boolean isSelected().

Menüeinträge können mit einem Tastaturkürzel (*Hotkey*) versehen werden. Damit kann dann ein Eintrag alternativ über die Tastatur ausgewählt werden.

Ein `javax.swing.KeyStroke`-Objekt repräsentiert einen Hotkey.

```
void setAccelerator(KeyStroke k)  
    stattet den Menüeintrag mit dem Hotkey k aus.
```

Die `KeyStroke`-Methode

```
static KeyStroke getKeyStroke(int code, int mod)
```

liefert einen Hotkey. Für `code` kann eine Konstante der Klasse `java.awt.event.KeyEvent` eingesetzt werden, z. B. `VK_ENTER`, `VK_TAB`, `VK_SPACE`, `VK_0`, ... , `VK_9`, `VK_A`, ..., `VK_Z`. Gültige Werte für `mod` sind die `java.awt.event.InputEvent`-Konstanten `CTRL_DOWN_MASK`, `SHIFT_DOWN_MASK`, `ALT_DOWN_MASK` oder eine additive Kombination hiervon.

### JCheckBoxMenuItem

Die Klasse `JCheckBoxMenuItem` implementiert einen Menüeintrag mit dem Zustand "selektiert" oder "nicht selektiert". `JCheckBoxMenuItem` hat die zu `JMenuItem` analogen Konstruktoren. Zusätzlich zu Text und Icon kann durch ein `boolean`-Argument der Auswahlzustand angegeben werden.

### JRadioButtonMenuItem

Die Klasse `JRadioButtonMenuItem` hat die zu `JMenuItem` analogen Konstruktoren. Mehrere dieser Menüeinträge können in einer Gruppe (`ButtonGroup`) zusammengefasst werden.

### ActionListener

Wenn ein Menüeintrag ausgewählt wird, erzeugt dieser ein `ActionEvent`. Ein Listener kann registriert werden.

### JToolBar

Symbolleisten werden von der Klasse `javax.swing.JToolBar` implementiert.

Die Konstruktoren

```
JToolBar()  
JToolBar(int orientation)
```

erzeugen eine Leiste mit horizontaler (`HORIZONTAL`) bzw. vertikaler (`VERTICAL`) Ausrichtung. Der parameterlose Konstruktor erzeugt eine horizontal ausgerichtete Leiste.

Die Symbolleiste kann mit der Maus verschoben werden. Mit "Schließen" kehrt die Leiste automatisch an ihren Ursprungsort zurück.

Symbolleisten werden mit der Container-Methode add gefüllt.

```
void setOrientation(int orientation)  
    legt die Ausrichtung fest.
```

```
void setFloatable(boolean b)  
    Hat b den Wert true, so kann die Symbolleiste mit der Maus an eine andere  
    Position gezogen werden.
```

```
void addSeparator()  
void addSeparator(Dimension size)  
    fügen Abstände hinzu. java.awt.Dimension(int width, int height) legt  
    Breite und Höhe fest.
```

### JPopupMenu

Kontextmenüs werden durch die Klasse javax.swing.JPopupMenu dargestellt.

Wie bei JMenu können Menüeinträge mit

```
JMenuItem add(JMenuItem mi)
```

hinzugefügt werden.

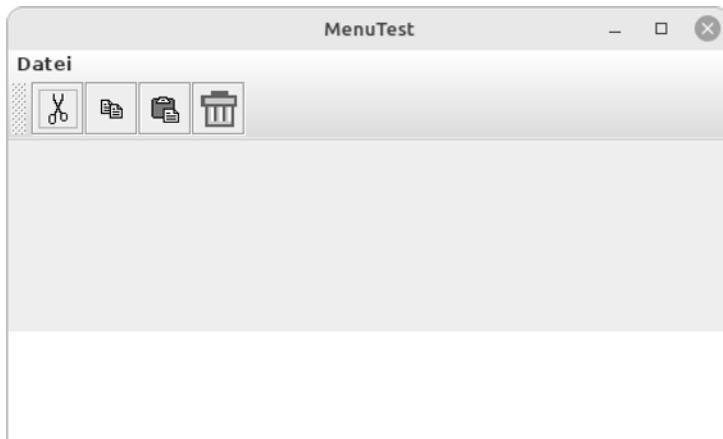
Ebenso existiert die Methode void addSeparator().

Mit der JComponent-Methode

```
void setComponentPopupMenu(JPopupMenu popup)
```

wird das Kontextmenü für diese Komponente gesetzt.

Das folgende Programm hat ein Menü, eine Symbolleiste und ein Kontextmenü für die Textfläche.



**Abbildung 27-17:** Menü, Symbolleiste und Kontextmenü

```
package menus;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MenuTest extends JFrame {
    private final JTextArea text;

    public MenuTest() {
        super("MenuTest");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        text = new JTextArea(5, 30);
        add(text, BorderLayout.SOUTH);

        buildMenu();
        buildToolBar();
        buildPopupMenu();

        setSize(500, 300);
        setVisible(true);
    }

    private void buildMenu() {
        JMenuBar bar = new JMenuBar();
        setJMenuBar(bar);

        JMenu file = new JMenu("Datei");
        bar.add(file);

        JMenuItem open = new JMenuItem("Öffnen");
        open.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_O, InputEvent.CTRL_DOWN_MASK));
        open.addActionListener(e -> System.out.println("Öffnen"));
        file.add(open);

        JMenuItem save = new JMenuItem("Speichern");
        save.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_S, InputEvent.CTRL_DOWN_MASK));
        save.addActionListener(e -> System.out.println("Speichern"));
        file.add(save);

        file.addSeparator();

        JMenuItem exit = new JMenuItem("Beenden");
        exit.addActionListener(e -> {
            System.out.println("Beenden");
            dispose();
        });
        file.add(exit);
    }

    private void buildToolBar() {
        JToolBar bar = new JToolBar();
        bar.setFloatable(true);
        add(bar, BorderLayout.NORTH);

        JButton cut = new JButton(new ImageIcon(getClass().getResource("cut.gif")));

```

```
cut.setToolTipText("Ausschneiden");
cut.addActionListener(e -> text.cut());
bar.add(cut);

JButton copy = new JButton(new ImageIcon(getClass()
    .getResource("copy.gif")));

copy.setToolTipText("Kopieren");
copy.addActionListener(e -> text.copy());
bar.add(copy);

JButton paste = new JButton(new ImageIcon(getClass()
    .getResource("paste.gif")));

paste.setToolTipText("Einfügen");
paste.addActionListener(e -> text.paste());
bar.add(paste);

JButton delete = new JButton(new ImageIcon(getClass()
    .getResource("delete.gif")));

delete.setToolTipText("Löschen");
delete.addActionListener(e -> text.setText(""));
bar.add(delete);
}

private void buildPopupMenu() {
    JPopupMenu popupMenu = new JPopupMenu();

    JMenuItem cut = new JMenuItem("Ausschneiden");
    cut.addActionListener(e -> text.cut());
    popupMenu.add(cut);

    JMenuItem copy = new JMenuItem("Kopieren");
    copy.addActionListener(e -> text.copy());
    popupMenu.add(copy);

    JMenuItem paste = new JMenuItem("Einfügen");
    paste.addActionListener(e -> text.paste());
    popupMenu.add(paste);

    JMenuItem delete = new JMenuItem("Löschen");
    delete.addActionListener(e -> text.setText(""));
    popupMenu.add(delete);

    text.setComponentPopupMenu(popupMenu);
}

public static void main(String[] args) {
    new MenuTest();
}
}
```

## 27.9 Dialoge

In diesem Abschnitt entwickeln wir einen Dialog für die Eingabe eines Passworts und nutzen des Weiteren vorgefertigte Dialoge für Standardanwendungsfälle.

### JDialog

Die Klasse `javax.swing.JDialog` repräsentiert ein spezielles Fenster, das abhängig von einem anderen Fenster geöffnet werden kann. Ein solches Fenster kann als *modales* oder *nicht modales* Fenster erstellt werden.

*Modal* bedeutet, dass man im anderen Fenster erst, nachdem das Dialogfenster geschlossen wurde, weiterarbeiten kann.

#### Die Konstruktoren

```
JDialog(JFrame owner)
JDialog(JFrame owner, boolean modal)
JDialog(JFrame owner, String title)
JDialog(JFrame owner, String title, boolean modal)
```

erzeugen ein Dialogfenster mit oder ohne Titel. Hat `modal` den Wert `true`, so wird ein modales Fenster erzeugt.

Wie bei `JFrame` gibt es auch hier die Methoden:

```
void setTitle(String title)
String getTitle()
void setResizable(boolean b)
boolean isResizable()
void setDefaultCloseOperation(int op)

void setModal(boolean b)
gibt an, ob das Fenster modal sein soll.

boolean isModal()
liefert den Wert true, wenn das Fenster modal ist.

void setLocationRelativeTo(Component c)
positioniert das Dialogfenster relativ zu c.
```

```
package dialogs;

import javax.swing.*;
import java.awt.*;

public class MyDialog extends JFrame {
    public MyDialog() {
        super("MyDialog");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
```

```
 JButton login = new JButton("Anmelden");
 login.addActionListener(e -> {
     LoginDialog dialog = new LoginDialog(this);
     if (dialog.isLoggedIn())
         System.out.println("logged in");
 });
 c.add(login);

 setSize(300, 200);
 setVisible(true);
}

private static class LoginDialog extends JDialog {
    private final JPasswordField passwordField;
    private boolean isLoggedIn;

    public LoginDialog(JFrame owner) {
        super(owner, "Login", true);

        setLocationRelativeTo(owner);
        setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        c.add(new JLabel("Passwort: "));
        passwordField = new JPasswordField(15);
        c.add(passwordField);

        JButton loginButton = new JButton("Login");
        loginButton.addActionListener(e -> {
            String pw = new String(passwordField.getPassword());
            if (pw.equals("secret")) {
                isLoggedIn = true;
                dispose();
            }
            passwordField.setText("");
        });
        c.add(loginButton);

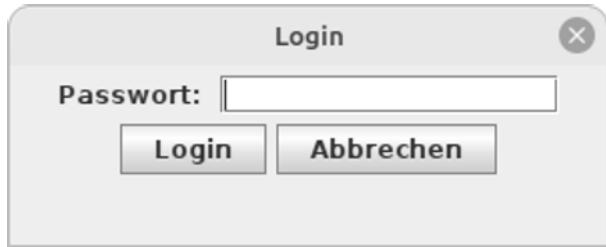
        JButton cancelButton = new JButton("Abbrechen");
        cancelButton.addActionListener(e -> dispose());
        c.add(cancelButton);

        setSize(300, 120);
        setResizable(false);
        setVisible(true);
    }

    public boolean isLoggedIn() {
        return isLoggedIn;
    }
}

public static void main(String[] args) {
    new MyDialog();
}
```

Der Aufruf von `setVisible(true)` in der Klasse `LoginDialog` macht das modale Dialogfenster sichtbar und blockiert die Ausführung des aufrufenden Threads, bis der Dialog mit `dispose()` beendet wird.



**Abbildung 27-18:** JDialog

## JOptionPane

Einfache Standarddialoge können mit Klassenmethoden der Klasse `javax.swing.JOptionPane` erstellt werden.

### Bestätigungsdialog

```
static int showConfirmDialog(Component owner, Object msg, String title,
    int optType, int msgType)
```

zeigt einen *Bestätigungsdialog* im übergeordneten Fenster `owner` mit Anzeigeobjekt `msg` und Titel `title`.

Die verfügbaren Buttons werden mittels `optType` angegeben.

Gültige Werte für `optType` sind die `JOptionPane`-Konstanten `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` und `OK_CANCEL_OPTION`.

Je nach ausgewähltem Button werden die `JOptionPane`-Konstanten `YES_OPTION`, `NO_OPTION`, `CANCEL_OPTION` oder `OK_OPTION` zurückgeliefert.

Der Nachrichtentyp `msgType` bestimmt das anzuzeigende Icon. Gültige Werte für `msgType` sind die `JOptionPane`-Konstanten:

`ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` und `PLAIN_MESSAGE`.

### Eingabedialog

```
static String showInputDialog(Component owner, Object msg, String title,
    int msgType)
```

zeigt einen *Eingabedialog* im übergeordneten Fenster mit Anzeigeobjekt, Titel und Nachrichtentyp (siehe oben). Der eingegebene Text wird zurückgeliefert.

### Mitteilungsdialog

```
static void showMessageDialog(Component owner, Object msg, String title,  
    int msgType)
```

zeigt einen *Mitteilungsdialog* im übergeordneten Fenster mit Anzeigeobjekt, Titel und Nachrichtentyp (siehe oben).

### JFileChooser

Die Klasse `javax.swing.JFileChooser` implementiert einen Dialog zur Auswahl von Dateien im Dateisystem.

Eigenschaften (get-/set-Methoden) und weitere Methoden:

<code>dialogTitle</code>	Titel des Dialogfensters ( <code>String</code> )
<code>currentDirectory</code>	aktuelles Verzeichnis ( <code>File</code> )
<code>selectedFile</code>	ausgewählte Datei ( <code>File</code> )
<code>multiSelectionEnabled</code>	Auswahl mehrerer Dateien bzw. Verzeichnisse ( <code>boolean</code> )

`File[] getSelectedFiles()`

liefert die ausgewählten Dateien, falls mehrere Dateien ausgewählt werden dürfen.

`void setFileSelectionMode(int mode)`

bestimmt, ob der Benutzer nur Dateien oder nur Verzeichnisse oder beides auswählen darf. Gültige Werte für `mode` sind die `JFileChooser`-Konstanten: `FILES_ONLY` (Voreinstellung), `DIRECTORIES_ONLY` und `FILES_AND_DIRECTORIES`.

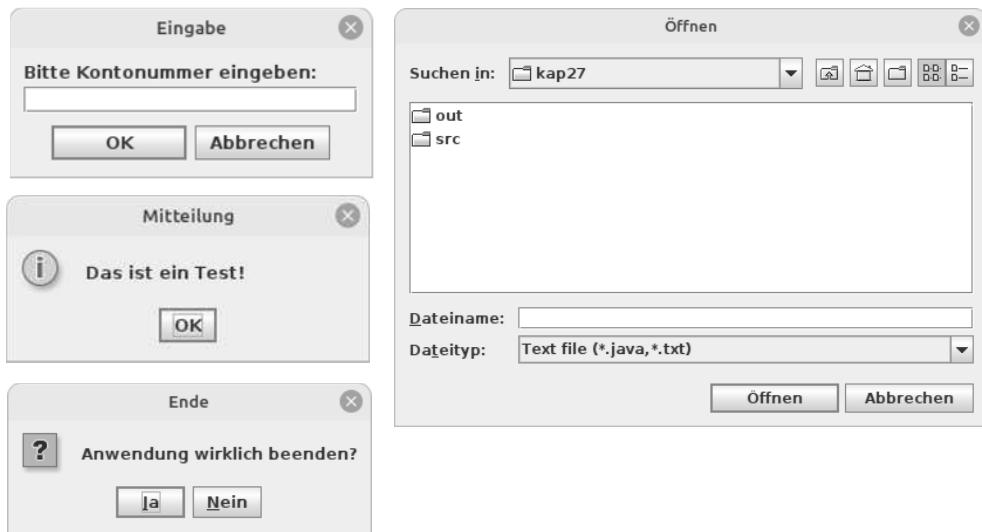
`int showOpenDialog(Component c)`

`int showSaveDialog(Component c)`

`int showDialog(Component c, String text)`

zeigen jeweils ein modales Dialogfenster an. Die Fenster unterscheiden sich nur in der Beschriftung der Titelleiste und des Buttons, mit dem die Auswahl bestätigt wird: "Öffnen", "Speichern" bzw. `text`.

Das Dialogfenster wird relativ zum übergeordneten Fenster `c` positioniert. `c` kann auch `null` sein. Zurückgeliefert wird eine der `JFileChooser`-Konstanten `APPROVE_OPTION`, `CANCEL_OPTION` oder `ERROR_OPTION` je nachdem, ob die Auswahl bestätigt, der Dialog abgebrochen wurde oder ein Fehler aufgetreten ist.



**Abbildung 27-19:** JOptionPane und JFileChooser

### FileFilter

```
void setFileFilter(FileFilter filter)
```

legt einen Filter fest, der dazu dient, Dateien von der Anzeige im Dialogfenster auszuschließen. filter ist Objekt einer Subklasse der abstrakten Klasse javax.swing.filechooser.FileFilter.

Folgende Methoden müssen in dieser Subklasse implementiert sein:

```
boolean accept(File file)
```

liefert true, wenn file ausgewählt werden darf.

```
String getDescription()
```

liefert eine Beschreibung des Filters.

Das nächste Programm zeigt die drei Standarddialoge und den Dialog zum Öffnen einer Datei (siehe Abbildung 27-19). Die ausgewählte Datei mit der Endung .java oder .txt wird am Bildschirm ausgegeben.

```
package dialogs;

import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.awt.*;
import java.io.File;
```

```
import java.io.IOException;
import java.nio.file.Files;
import java.util.List;

public class StandardDialogs extends JFrame {
    public StandardDialogs() {
        super("Standard-Dialoge");
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        JButton inputButton = new JButton("Eingabe");
        inputButton.addActionListener(e -> {
            String input = JOptionPane.showInputDialog(this,
                "Bitte Kontonummer eingeben:", "Eingabe",
                JOptionPane.PLAIN_MESSAGE);
            if (input != null)
                System.out.println(input);
        });
        c.add(inputButton);

        JButton messageButton = new JButton("Mitteilung");
        messageButton.addActionListener(e ->
            JOptionPane.showMessageDialog(this, "Das ist ein Test!",
                "Mitteilung", JOptionPane.INFORMATION_MESSAGE)
        );
        c.add(messageButton);

        JButton openButton = new JButton("Öffnen");
        openButton.addActionListener(e -> {
            JFileChooser fileChooser = new JFileChooser();
            fileChooser.setCurrentDirectory(new File("."));
            fileChooser.setFileFilter(new FileFilter() {
                @Override
                public boolean accept(File f) {
                    if (f.isDirectory())
                        return true;
                    String name = f.getName();
                    if (name.endsWith(".java"))
                        return true;
                    else return name.endsWith(".txt");
                }
                @Override
                public String getDescription() {
                    return "Text file (*.java, *.txt)";
                }
            });

            if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
                File file = fileChooser.getSelectedFile();
                try {
                    List<String> lines = Files.readAllLines(file.toPath());
                    lines.forEach(System.out::println);
                } catch (IOException ex) {
                    System.err.println(ex.getMessage());
                }
            }
        });
    }
}
```

```

        c.add(openButton);

        JButton confirmButton = new JButton("Beenden");
        confirmButton.addActionListener(e -> {
            int result = JOptionPane.showConfirmDialog(this,
                "Anwendung wirklich beenden?", "Ende",
                JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
            if (result == JOptionPane.YES_OPTION)
                dispose();
        });
        c.add(confirmButton);

        setSize(500, 100);
        setVisible(true);
    }

    public static void main(String[] args) {
        new StandardDialogs();
    }
}

```

## 27.10 Tabellen

Mit Hilfe der Klasse `javax.swing.JTable` können Daten tabellarisch angezeigt und einzelne Tabelleneinträge geändert werden.

Die Anzeigekomponente einer Tabelle ist von der Struktur und dem Inhalt der Daten, die angezeigt und geändert werden können, getrennt (MVC-Architektur).

Aus der umfangreichen Sammlung von Methoden zur Erstellung und Präsentation von Tabellendaten zeigt dieses Unterkapitel nur eine Auswahl.

### JTable

Der Konstruktor

```
JTable(TableModel model)
```

erzeugt eine Tabelle auf der Basis des Datenmodells `model`.

Wird der Standardkonstruktor `JTable()` verwendet, so kann das Datenmodell mit der `JTable`-Methode

```
void setModel(TableModel model)
```

gesetzt werden.

Das Interface `javax.swing.table.TableModel` spezifiziert Methoden, die für die Anzeige und Änderung der Daten genutzt werden.

### AbstractTableModel

Die abstrakte Klasse `javax.swing.table.AbstractTableModel` implementiert das Interface `TableModel` nur zum Teil.

Konkrete Klassen, die von `AbstractTableModel` abgeleitet sind, müssen zumindest die folgenden `AbstractTableModel`-Methoden implementieren:

```
int getColumnCount()  
    liefert die Anzahl der Spalten im Datenmodell.
```

```
int getRowCount()  
    liefert die Anzahl der Zeilen im Datenmodell.
```

```
Object getValueAt(int row, int col)  
    liefert den aktuellen Wert in Zeile row und Spalte col im Datenmodell.
```

Weitere `AbstractTableModel`-Methoden sind:

```
String getColumnName(int col)  
    liefert den Namen der Spalte col im Datenmodell.
```

```
Class<?> getColumnClass(int col)  
    liefert das Class-Objekt der Klasse, der alle Objekte der Spalte col angehören.  
    JTable nutzt diese Information, um die Werte entsprechend darzustellen (z. B.  
    rechtsbündige Ausrichtung bei Integer-Objekten). Wenn diese Methode nicht  
    überschrieben wird, wird jeder Wert als Zeichenkette dargestellt und links-  
    bündig ausgerichtet.
```

```
boolean isCellEditable(int row, int col)  
    liefert true, falls die Zelle editiert werden kann. Die Standard-Implementierung  
    liefert false.
```

```
void setValueAt(Object value, int row, int col)  
    setzt den Wert einer Zelle im Datenmodell. Diese Methode ist standardmäßig  
    mit einem leeren Rumpf implementiert.
```

### TableModelListener

Für die Benachrichtigung bei Änderung des Datenmodells wird ein Event vom Typ `javax.swing.event.TableModelEvent` erzeugt.

Das Funktionsinterface `javax.swing.event.TableModelListener` hat die Methode

```
void tableChanged(TableModelEvent e)
```

Dieser Listener wird mit der `AbstractTableModel`-Methode

```
void addTableModelListener(TableModelListener l)
```

registriert.

Die `AbstractTableModel`-Methode

```
void fireTableDataChanged()
```

benachrichtigt alle registrierten `TableModelListener`-Instanzen darüber, dass Daten geändert wurden.

### ListSelectionListener

Die *Auswahl von Zeilen* in der Tabelle löst ein Event vom Typ `javax.swing.event.ListSelectionEvent` aus.

Listener hierfür müssen sich bei einem `javax.swing.ListSelectionModel`-Objekt, das von der `JTable`-Methode `getSelectionModel` bereitgestellt wird, registrieren.

Registrierungsmethode:

```
void addListSelectionListener(ListSelectionListener l)
```

Das Funktionsinterface `javax.swing.event.ListSelectionListener` definiert die Methode

```
void valueChanged(ListSelectionEvent e)
```

Weiter `JTable`-Methoden sind:

```
ListSelectionModel getSelectionModel()
```

liefert das `ListSelectionModel`-Objekt. Dieses verfügt auch über die von `JList` her bekannte Methode `void setSelectionMode(int mode)`.

```
int[] getSelectedRows()
```

liefert die Indizes aller selektierten Zeilen oder ein leeres Array, wenn keine Zeile selektiert wurde.

Das Programm `TableTest` stellte eine Tabelle mit den Spalten *Artikel*, *Preis*, *Menge* und *Summe* dar. Preise und Mengen können geändert werden. Einzelsummen und Gesamtsumme werden bei Änderung automatisch aktualisiert. Wählt man einzelne Zeilen aus, werden deren Summen aufaddiert und das Ergebnis wird angezeigt.

```
package table;

import javax.swing.*;
import javax.swing.border.EmptyBorder;
import javax.swing.table.AbstractTableModel;
import java.awt.*;
import java.util.List;

public class TableTest extends JFrame {
    private final JLabel total;
    private final JLabel selected;
```

```
public TableTest() {
    super("TableTest");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    Model model = new Model();

    JTable table = new JTable(model);
    JScrollPane scrollPane = new JScrollPane(table);
    scrollPane.setBorder(new EmptyBorder(10, 10, 10, 10));
    add(scrollPane, BorderLayout.CENTER);

    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(2, 1));
    panel.setBorder(new EmptyBorder(10, 10, 10, 10));
    total = new JLabel("Gesamt: " + String
        .format("%5d", model.total()), JLabel.RIGHT);
    total.setFont(new Font("Monospaced", Font.BOLD, 14));
    selected = new JLabel("Ausgewählt: " + String
        .format("%5d", 0), JLabel.RIGHT);
    selected.setFont(new Font("Monospaced", Font.BOLD, 14));
    panel.add(total);
    panel.add(selected);
    add(panel, BorderLayout.SOUTH);

    model.addTableModelListener(e ->
        total.setText("Gesamt: " + String.format("%5d", model.total())));

    table.getSelectionModel().addListSelectionListener(e -> {
        int[] rows = table.getSelectedRows();
        int sum = model.sum(rows);
        selected.setText("Ausgewählt: " + String.format("%5d", sum));
    });
}

setSize(400, 220);
setVisible(true);
}

private static class Artikel {
    String name;
    int preis;
    int menge;

    Artikel(String name, int preis, int menge) {
        this.name = name;
        this.preis = preis;
        this.menge = menge;
    }
}

private static class Model extends AbstractTableModel {
    private final String[] names = {"Artikel", "Preis", "Menge", "Summe"};
    private final List<Artikel> list = List.of(
        new Artikel("A4711", 100, 10),
        new Artikel("A4721", 80, 5),
        new Artikel("A4731", 10, 20),
        new Artikel("A4741", 12, 5),
        new Artikel("A4751", 250, 4)
    );
}
```

```
@Override
public int getColumnCount() {
    return names.length;
}

@Override
public int getRowCount() {
    return list.size();
}

@Override
public String getColumnName(int col) {
    return names[col];
}

@Override
public boolean isCellEditable(int row, int col) {
    return col == 1 || col == 2;
}

@Override
public Object getValueAt(int row, int col) {
    Artikel artikel = list.get(row);
    return switch (col) {
        case 0 -> artikel.name;
        case 1 -> artikel.preis;
        case 2 -> artikel.menge;
        case 3 -> artikel.preis * artikel.menge;
        default -> null;
    };
}

@Override
public void setValueAt(Object value, int row, int col) {
    Artikel artikel = list.get(row);
    switch (col) {
        case 1:
            artikel.preis = (Integer) value;
        case 2:
            artikel.menge = (Integer) value;
    }
    fireTableDataChanged();
}

@Override
public Class<?> getColumnClass(int col) {
    return col == 0 ? String.class : Integer.class;
}

public int total() {
    int sum = 0;
    for (Artikel artikel : list) {
        sum += artikel.menge * artikel.preis;
    }
    return sum;
}
```

```

public int sum(int[] rows) {
    int sum = 0;
    for (int row : rows) {
        sum += list.get(row).menge * list.get(row).preis;
    }
    return sum;
}

public static void main(String[] args) {
    new TableTest();
}
}

```

**TableTest**

Artikel	Preis	Menge	Summe
A4711	100	10	1000
A4721	80	5	400
A4731	10	20	200
A4741	12	5	60
A4751	250	4	1000

**Gesamt:** 2660  
**Ausgewählt:** 1400

**Abbildung 27-20:** Artikeltabelle

## 27.11 Event-Queue und Event-Dispatcher

Für die Verarbeitung von Benutzereingaben und die Aktualisierung der sichtbaren Komponenten eines Fensters ist ein eigener System-Thread, der sogenannte *Event-Dispatcher* verantwortlich.

So werden vom Benutzer bzw. Programm initiierte Ereignisse zunächst in eine *Event-Queue* eingestellt. Die gesammelten Ereignisse werden dann vom Event-Dispatcher der Reihe nach abgearbeitet.

Beispielsweise werden die *JComponent*-Methode *paintComponent* und Listener-Methoden wie z. B. *actionPerformed* von diesem Dispatcher ausgeführt.

Die *java.awt.EventQueue*-Methode

```
static boolean isDispatchThread()
```

liefert true, wenn der aufrufende Thread der Event-Dispatcher ist.

Das folgende Beispiel soll mögliche Probleme bei der Aktualisierung von Ausgaben demonstrieren.

Das Programm `Test1` zeigt in Abständen von 100 Millisekunden die aktuelle Systemzeit an. Diese Anzeige erfolgt in einem eigenen Anwendungs-Thread. Es kann auf einen Button geklickt werden, dessen Beschriftung sich erst nach einer Verzögerung von zwei Sekunden ändert. Mit dieser Verzögerung soll eine länger laufende Hintergrundverarbeitung simuliert werden. Während der Wartezeit wird die Zeitanzeige nicht aktualisiert. Die Oberfläche "friert ein".

```
package threading;

import javax.swing.*;
import java.awt.*;

public class Test1 extends JFrame {
    private final JLabel label;
    private final JButton button;
    private int count;

    public Test1() {
        super("Test1");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        label = new JLabel(" ", JLabel.CENTER);
        add(label, BorderLayout.CENTER);

        button = new JButton("0");
        button.addActionListener(e -> {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException ignored) {
            }
            button.setText(String.valueOf(++count));
        });
        add(button, BorderLayout.SOUTH);

        setSize(300, 200);
        setVisible(true);
    }

    Thread t = new Thread(() -> {
        while (true) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException ignored) {
            }
            label.setText(String.valueOf(System.currentTimeMillis()));
        }
    });
    t.start();
}
```

```
public static void main(String[] args) {
    new Test1();
}
```

Im Programm Test2 wird die Verzögerung um zwei Sekunden vor der Aktualisierung der Button-Beschriftung in einem *Anwendungs-Thread* implementiert und findet dann nicht mehr im Event-Dispatcher statt.

Die Aktualisierung der GUI-Oberfläche (Zeitanzeige und Buttonbeschriftung) wird komplett an den *Event-Dispatcher* delegiert.

Hierzu stehen die Klassenmethoden `invokeAndWait` und `invokeLater` der Klasse `EventQueue` zur Verfügung.

Die Ausführung des Programms zeigt, dass nunmehr keine Behinderung mehr auftritt.

```
package threading;

import javax.swing.*;
import java.awt.*;

public class Test2 extends JFrame {
    private final JLabel label;
    private final JButton button;
    private int count;

    public Test2() {
        super("Test2");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        label = new JLabel(" ", JLabel.CENTER);
        add(label, BorderLayout.CENTER);

        button = new JButton("0");
        button.addActionListener(e -> {
            Runnable runner = () -> {
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException ignored) {
                }
                EventQueue.invokeLater(() -> button.setText(String.valueOf(++count)));
            };
            Thread t = new Thread(runner);
            t.start();
        });
        add(button, BorderLayout.SOUTH);

        setSize(300, 200);
        setVisible(true);
    }

    Thread t = new Thread(() -> {
```

```

        while (true) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException ignored) {
            }
            EventQueue.invokeLater(() ->
                label.setText(String.valueOf(System.currentTimeMillis())));
        }
    });
    t.start();
}

public static void main(String[] args) {
    new Test2();
}
}

```

### EventQueue-Methoden:

`static void invokeAndWait(Runnable task) throws InterruptedException,  
InvocationTargetException`

Die `run`-Methode von `task` wird vom Event-Dispatcher ausgeführt, sobald noch nicht behandelte AWT- und Swing-Ereignisse verarbeitet sind. Die Methode `invokeAndWait` blockiert so lange, bis die `run`-Methode ausgeführt ist. Die Methode läuft also *synchron*.

Wird der Wartevorgang unterbrochen, so wird `InterruptedException` ausgelöst. Löst die `run`-Methode eine Ausnahme im Dispatcher aus, die nicht abgefangen ist, so wird `java.lang.reflect.InvocationTargetException` ausgelöst.

`invokeAndWait` darf nicht im Event-Dispatcher selbst aufgerufen werden.

`static void invokeLater(Runnable task)`

Im Gegensatz zu `invokeAndWait` blockiert der Aufruf von `invokeLater` nicht. `task` wird lediglich ans Ende der Event-Queue des Dispatchers angefügt. Die Methode wird also *asynchron* ausgeführt.

### Fazit

Um Problemen bei der Aktualisierung der Oberfläche vorzubeugen, sollten Swing-Komponenten in der Regel nur über den Event-Dispatcher manipuliert werden. Andere, länger laufende Aktionen sollten in Anwendungs-Threads ausgelagert werden.

Spätestens ab dem Aufruf von `setVisible(true)` sollte die GUI-Aktualisierung so abgesichert werden.

Das folgende Programm nutzt diese Technik, um einen Fortschrittsbalken laufend zu aktualisieren.

### JProgressBar

Mit Hilfe von `javax.swing.JProgressBar` kann der Fortschritt irgendeiner laufenden Arbeit visualisiert werden. Der Fertigstellungsgrad der Arbeit kann in Prozent angezeigt werden.

Der Konstruktor

```
JProgressBar(int min, int max)
```

erzeugt einen Fortschrittsbalken, `min` und `max` geben den Start- und Endwert des Fortschritts an.

```
void setValue(int n)
```

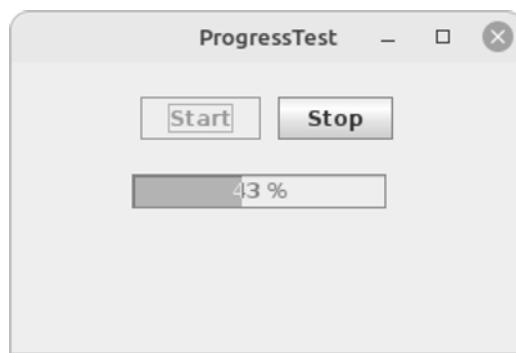
setzt den aktuellen Fortschrittwert auf den Wert `n`.

```
int getValue()
```

liefert den aktuellen Fortschrittwert.

```
void setStringPainted(boolean b)
```

legt fest, ob der Fertigstellungsgrad (in Prozent) angezeigt werden soll.



**Abbildung 27-21:** JProgressBar

```
package threading;

import javax.swing.*;
import java.awt.*;

public class ProgressTest extends JFrame {
    private JProgressBar progressBar;
    private final JButton start;
    private JButton stop;
    private JLabel message;
    private Thread thread;
```

```
private final int MAX = 100_000_000;

public ProgressTest() {
    super("ProgressTest");

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container container = getContentPane();
    container.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 20));

    start = new JButton("Start");
    start.addActionListener(e -> {
        progressBar.setValue(0);
        message.setText("");
        start.setEnabled(false);
        stop.setEnabled(true);
        thread = new Thread(new LongTask());
        thread.start();
    });
    container.add(start);

    stop = new JButton("Stop");
    stop.addActionListener(e -> {
        if (thread != null) {
            thread.interrupt();
            thread = null;
        }
    });
    stop.setEnabled(false);
    container.add(stop);

    progressBar = new JProgressBar(0, MAX);
    progressBar.setStringPainted(true);
    container.add(progressBar);

    message = new JLabel("");
    container.add(message);

    setSize(300, 200);
    setVisible(true);
}

private class LongTask implements Runnable {
    private int value;

    public void run() {
        double z = 0;
        for (int i = 0; i < MAX; i++) {
            if (Thread.currentThread().isInterrupted())
                break;
            z += Math.sin(i) + Math.cos(i);
            if (i % 100_000 == 0) {
                value = i;
                EventQueue.invokeLater(() ->
                    progressBar.setValue(value)
                );
            }
        }
    }
}
```

```
        String result = Thread.currentThread()
                        .isInterrupted() ? "" : String.valueOf(z);

        EventQueue.invokeLater(() -> {
            message.setText(result);
            start.setEnabled(true);
            stop.setEnabled(false);
        });
    }

    public static void main(String[] args) {
        new ProgressTest();
    }
}
```

## 27.12 Aufgaben

### Aufgabe 1

Implementieren Sie für ein Fenster alle `WindowListener`-Methoden und testen Sie, wann welche Methode aufgerufen wird.

*Lösung: Paket window\_events*

### Aufgabe 2

In einem Fenster soll das aktuelle Datum mit Uhrzeit im Sekundentakt angezeigt werden. Durch Betätigung eines Buttons soll die Anzeige gestoppt und wieder gestartet werden können. Tipp: Nutzen Sie einen Thread, der die aktuelle Zeit ermittelt sowie `repaint` und `Thread.sleep` nutzt.

*Lösung: Paket time\_display*

### Aufgabe 3

Erstellen Sie eine Textfläche und drei Buttons "Copy", "Cut" und "Paste", mit denen markierter Text in die Zwischenablage bzw. aus ihr kopiert werden kann.

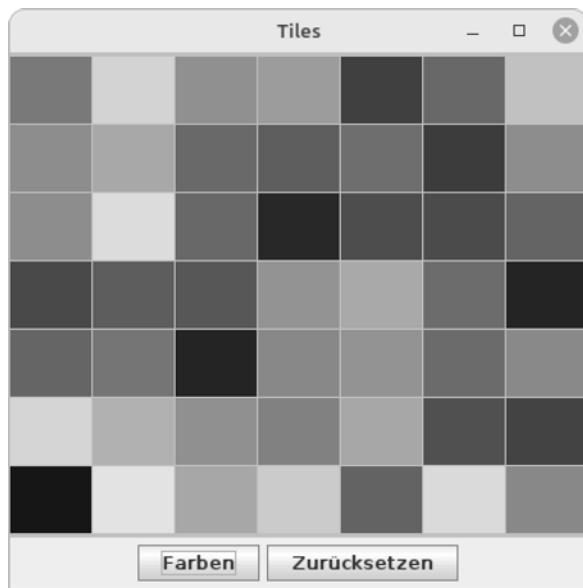
*Lösung: Paket clipboard*

### Aufgabe 4

7 x 7 Kacheln sollen in einem Fenster angezeigt werden.

Über Buttons können die Farben aller Kacheln durch Zufall bestimmt werden bzw. auf die Farbe Weiß zurückgesetzt werden. Ebenso soll der Mausklick auf eine Kachel eine zufällige Farbe für diese eine Kachel bestimmen.

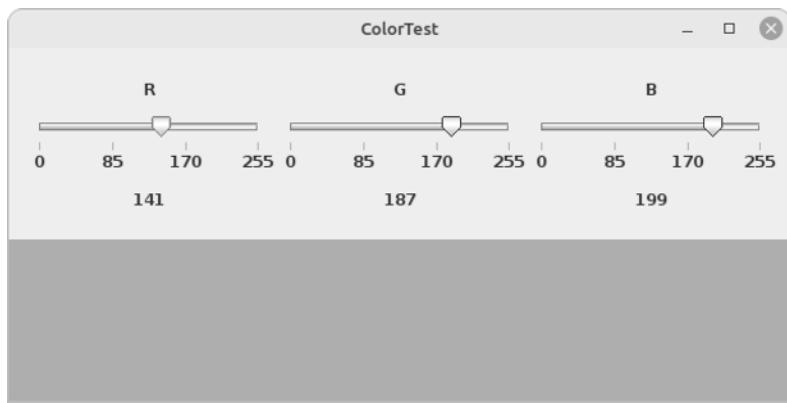
*Lösung: Paket tiles*



### Aufgabe 5

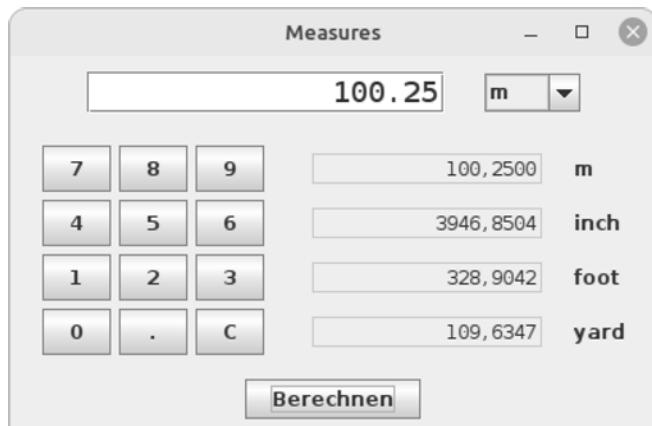
Erstellen Sie ein Programm, mit dem die drei Anteile einer Farbe nach dem RGB-Modell über `JSlider`-Komponenten eingestellt werden können.

*Lösung: Paket colors*



### Aufgabe 6

Schreiben Sie ein Programm, das Längenangaben in *m*, *inch*, *foot* und *yard* ineinander umrechnet.



Es gelten folgende Beziehungen:

$$1 \text{ inch} = 0,0254 \text{ m}; 1 \text{ foot} = 12 \text{ inch} = 0,3048 \text{ m}; 1 \text{ yard} = 3 \text{ foot} = 0,9144 \text{ m}$$

Die Eingabe der umzurechnenden Längenangabe kann direkt über Tastatur in einem Textfeld oder durch Drücken der Tasten eines Tastenfeldes erfolgen. Die Taste "C" löscht den Inhalt des Textfeldes. Die Maßeinheit kann über eine Auswahlliste eingestellt werden.

*Lösung: Paket measures*

### Aufgabe 7

Durch Drücken der Maustaste soll ein Punkt an der Position des Mauszeigers in einem Panel gezeichnet werden. Speichern Sie die gezeichneten Punkte in einer Liste vom Typ `ArrayList`. Die Methode `paintComponent` ist so zu implementieren, dass alle Punkte der Liste gezeichnet werden.

*Lösung: Paket points*

### Aufgabe 8

Erweitern Sie die Funktionalität des Programm `MouseTest` aus Kapitel 27.2:

Ein Quadrat kann gezeichnet werden, wenn die Shift-Taste während des Aufziehens der Figur mit der Maus gedrückt wird. Die x-Koordinate des Mauszeigers bestimmt die Seitenlänge des Quadrats. Mit Hilfe einer Check-Box (`JCheckBox`) kann bestimmt werden, ob die Figur mit der eingestellten Farbe ausgefüllt gezeichnet werden soll. Die Zeichenfarbe kann über eine Auswahlliste (`JComboBox`) eingestellt werden. Alle Figuren können gelöscht werden (`JButton`).

Die Zeichnung soll in einer Datei gespeichert werden können (Button: Speichern). Aus dieser Datei soll die Zeichnung wieder rekonstruiert werden können (Button:

Laden). Nutzen Sie dazu die Klassen `XMLEncoder` und `XMLDecoder` aus dem Aufgabenteil in Kapitel 25.

*Lösung: Paket painting*

### Aufgabe 9

Erstellen Sie einen Bildbetrachter zur Anzeige von Bildern im gif-, jpg- oder png-Format. Die Bilddateien sollen in einem Dialog ausgewählt werden können (`JFileChooser`). Mit Hilfe eines Filters sollen nur die geeigneten Dateien zur Auswahl angeboten werden. Tipp: Zeigen Sie das Bild als Icon innerhalb eines mit Scrollbalken ausgestatteten Labels an.

*Lösung: Paket image\_viewer*

### Aufgabe 10

Entwickeln Sie einen Texteditor, mit dem Textdateien mit der Endung ".txt" und ".java" editiert werden können. Das Menü "Datei" soll die Punkte "Neu", "Öffnen...", "Speichern", "Speichern unter..." und "Beenden" mit den üblichen Funktionen enthalten. Die Auswahl von Dateien zum Öffnen und Schließen soll über einen Dialog (`JFileChooser`) erfolgen.

*Lösung: Paket editor*

### Aufgabe 11

In einem Fenster sollen der Reihe nach alle gif-, jpg- bzw. png-Bilder eines Verzeichnisses mit voreingestellten Anzeigedauer angezeigt werden (Diashow). Tipp: Nutzen Sie die `File`-Methode `File[] listFiles(FileFilter filter)`, um alle Bilder des aktuellen Verzeichnisses zu bestimmen.

*Lösung: Paket diashow*

### Aufgabe 12

Der Fortschritt beim Laden einer größeren Datei soll mit einer `JProgress`-Komponente (0 bis 100 %) visualisiert werden. Die Dateiauswahl soll über einen Dialog (`JFileChooser`) erfolgen. Tipp: Damit das System nicht zu sehr belastet wird, sollte die Fortschrittsbalkenanzeige in Abständen von 500 gelesenen Bytes aktualisiert werden.

*Lösung: Paket progress*

### Aufgabe 13

Nutzen Sie das `GridBagLayout`, um die Komponenten aus der folgenden Abbildung zu platzieren.



Lösung: Paket gridbag

#### Aufgabe 14

Schreiben Sie ein Programm, das eine ToDo-Liste (`JList`) verwaltet.

Die Listeneinträge sollen beim Beenden des Programms automatische serialisiert in eine Datei geschrieben werden. Beim Starten des Programms sollen diese Einträge aus der Datei geladen werden (Deserialisierung).

Nutzen Sie hierzu die Klassen  `ObjectOutputStream / ObjectInputStream` bzw. `XMLEncoder / XMLDecoder` aus dem Aufgabenteil in Kapitel 25.

Das Programm soll die folgenden Funktionen anbieten:

*Hinzufügen:*

Neuen Eintrag ans Ende der Liste anhängen.

*Kopieren:*

Ein ausgewählter Eintrag wird im Eingabefeld angezeigt.

*Eintrag löschen:*

Ein ausgewählter Eintrag wird gelöscht.

*Drucken:*

Die Liste wird gedruckt. Hierzu werden die Listeneinträge in eine `JTextArea` übernommen.

*Liste löschen:*

Alle Einträge werden gelöscht.

Listeneinträge können mit dem Mauszeiger verschoben werden. Hierzu sind die Ereignismethoden `mousePressed` und `mouseDragged` geeignet zu implementieren.



*Lösung: Paket todo*

### Aufgabe 15

Realisieren Sie ein Programm mit einer Textfläche zur Eingabe von Texten. Mit der Tastenkombination Strg + "+" sollen die Schriftzeichen um eine bestimmte Einheit vergrößert, mit Strg + "-" verkleinert werden. Dies soll auch mit Hilfe der *Mausradbewegung* ermöglicht werden. Mit Strg + "0" soll die Größe auf den Ausgangswert zurückgesetzt werden können.

Das Funktionsinterface `java.awt.event.MouseWheelListener` enthält die folgende Methode:

```
void mouseWheelMoved(MouseWheelEvent e)
```

Die `java.awt.event.MouseWheelEvent`-Methode

```
int getWheelRotation()
```

liefert die Anzahl "Klicks", um die das Rad bewegt wurde. Negative Werte treten auf, wenn man das Rad nach vorne, vom Benutzer weg bewegt.

Dieser Listener kann mit der Component-Methode

```
void addMouseWheelListener(MouseWheelListener l)
```

registriert werden.

*Lösung: Paket zoom*



## 28 Einführung in JavaFX

JavaFX (alternative Bezeichnung *OpenJFX*) ist das aktuelle Java-Framework zur Erstellung von GUIs, das die klassischen GUI-Technologien AWT und Swing ablösen soll.

Mit JavaFX können Oberflächen komplett in Java programmiert oder alternativ mit einer XML-Sprache beschrieben werden, sodass eine Trennung zwischen Datenmodell (*Model*), Oberfläche (*View*) und Steuerung (*Controller*) nach dem *MVC-Konzept* gut umgesetzt werden kann.

Alle User-Interface-Komponenten können mit Hilfe der aus der Webseitenentwicklung bekannten Sprache CSS (*Cascading Style Sheets*) gestaltet werden, wodurch auch hier eine Trennung zwischen Design und Programmierung möglich ist.

JavaFX ist ab Version 11 nicht mehr Bestandteil von Java und muss deshalb separat installiert werden.

### Lernziele

In diesem Kapitel lernen Sie

- wie JavaFX installiert werden kann,
- welche Besonderheiten bei Übersetzung und Ausführung zu beachten sind,
- wie eine JavaFX-Anwendung grundsätzlich aufgebaut ist,
- wie die Oberfläche mit einem GUI-Design-Tool erstellt werden kann,
- wie CSS zur Gestaltung von Oberflächenelementen genutzt werden kann,
- wie lang laufende Aktivitäten verarbeitet werden können und
- wie Diagramme und Tabellen mit JavaFX erstellt werden können.

### 28.1 Installation und Konfiguration

Das *JavaFX-SDK* kann von der Website

<https://openjfx.io>

in Form einer ZIP-Datei für das jeweilige Betriebssystem heruntergeladen werden.

Beispiel (hier die zum Zeitpunkt der Drucklegung für Linux aktuellen Version):

`openjfx-21.0.1_linux-x64_bin-sdk.zip`

Die ZIP-Datei muss dann entpackt werden. Es entsteht das neue Verzeichnis:

`javafx-sdk-21.0.1`

Richten Sie dann die persistente Umgebungsvariable `PATH_TO_FX` ein, die auf das Verzeichnis `lib` von `javafx-sdk-21.0.1` zeigt.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_28](https://doi.org/10.1007/978-3-658-43574-5_28).

Beispiel (für Linux):

```
export PATH_TO_FX=/home/dietmar/Programme/javafx-sdk-21.0.1/lib
```

### Übersetzung und Ausführung in der Entwicklungsumgebung IntelliJ IDEA

In *File > Project Structure* ist eine neue *Global Library* aufzunehmen, z. B. mit dem Namen *javafx*. Hierzu muss das Verzeichnis *javafx-sdk-21.0.1/lib* ausgewählt werden.

Für jedes neu angelegte *JavaFX-Projekt* muss dann diese Library für die Übersetzung in *Dependencies* unter *Modules* mit *Add* hinzugefügt werden.

Um die Ausführung von JavaFX-Programmen in der Entwicklungsumgebung zu erleichtern muss in *File > Settings > Appearance & Behavior > Path Variables* *PATH\_TO\_FX* auf den Wert wie oben gesetzt werden.

Die Laufzeitumgebung eines ausführbaren Programms muss unter *Edit Configurations* konfiguriert werden. Unter *VM options* ist einzutragen:

```
-p ${PATH_TO_FX} --add-modules javafx.controls
```

bzw. falls *FXML-Dateien* verwendet werden:

```
-p ${PATH_TO_FX} --add-modules javafx.controls,javafx.fxml
```

### Übersetzung und Ausführung in der Shell (Terminal)

Beispiel (für Linux):

```
javac -d out/production/kap28 \
-p $PATH_TO_FX --add-modules=javafx.controls \
src/first/Demo.java

java -cp out/production/kap28 \
-p $PATH_TO_FX --add-modules=javafx.controls \
first.Demo
```

Wenn FXML genutzt wird: `--add-modules=javafx.controls,javafx.fxml`

### GUI-Design-Tool Scene Builder

Mit dem Tool *Scene Builder* kann das Layout einer JavaFX-Anwendung per Drag-and-Drop visuell erstellt werden. Das Tool erzeugt als Ergebnis eine XML-Datei, die das Layout der Anwendung beschreibt.

Der *Scene Builder* kann von der Website

<https://gluonhq.com/products/scene-builder/>

für das jeweilige Betriebssystem heruntergeladen werden.

In *IntelliJ IDEA* kann eine FXML-Datei direkt im *Scene Builder* geöffnet werden. Hierzu muss der Pfad unter

*File > Settings > Languages & Frameworks > JavaFX*  
eingetragen werden.

### Dokumentation (hier für die Version 21)

Die API-Dokumentation zu JavaFX (Javadoc) findet man unter:

<https://openjfx.io/javadoc/21/>

JavaFX CSS Reference Guide:

<https://openjfx.io/javadoc/21/javafx.graphics/javafx/scene/doc-files/cssref.html>

## 28.2 Ein erstes Beispiel

Das erste Programmbeispiel zeigt den grundsätzlichen Aufbau einer JavaFX-Anwendung.

Jede JavaFX-Anwendung ist von der Klasse `javafx.application.Application` abgeleitet. Die geerbte abstrakte Methode `start` muss implementiert werden. Sie konstruiert das GUI. Die durch `main` aufgerufene Methode `launch` startet die JavaFX-Anwendung, was dazu führt, dass nun die implementierte Methode `start` mit einer `Stage`-Instanz aufgerufen wird.

Die Methode `init` kann aufgerufen werden, um eine Initialisierung (z. B. das Herstellen einer Datenbankverbindung) vor dem eigentlichen Start der Anwendung (Methode `start`) vorzunehmen. Die Methode `stop` wird unmittelbar vor dem Ende der Anwendung aufgerufen. Hier können Ressourcen, wie z. B. eine Datenbankverbindung, freigegeben werden.

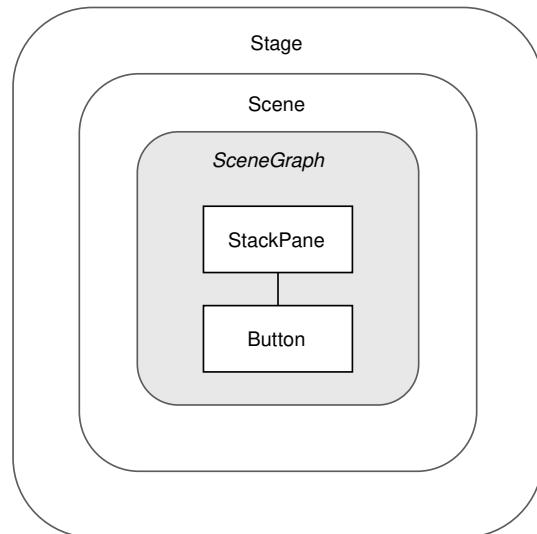
Jede JavaFX-Anwendung durchläuft also drei Methoden: `init – start – stop`. `init` und `stop` müssen nicht überschrieben werden. Die Originalmethoden haben "leere" Implementierungen.

```
package first;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.event.Event;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class Demo extends Application {
```

```
@Override  
public void init() {  
    System.out.println("init");  
}  
  
@Override  
public void start(Stage stage) {  
    System.out.println("start");  
    StackPane pane = new StackPane();  
  
    Button button = new Button();  
    button.setText("Hier klicken");  
    button.setOnAction(e -> Platform.exit());  
    pane.getChildren().add(button);  
  
    stage.setScene(new Scene(pane, 400, 100));  
    stage.setOnCloseRequest(Event::consume);  
    stage.setTitle("Demo1");  
    stage.setResizable(false);  
    stage.show();  
}  
  
@Override  
public void stop() {  
    System.out.println("stop");  
}  
  
public static void main(String[] args) {  
    launch(args);  
}  
}
```



**Abbildung 28-1:** Stage, Scene und SceneGraph

`javafx.stage.Stage` ist die "Bühne" für eine JavaFX-Anwendung und stellt den Rahmen für diese Anwendung dar (vergleichbar mit `JFrame` in Swing).

Die Benutzungsoberfläche ist wie bei Swing hierarchisch aufgebaut. Der sogenannte *SceneGraph* ist ein Baum, der aus Knoten vom Typ `javafx.scene.Node` besteht. Die Anordnung der Knoten (hier im Beispiel ein Button) wird durch spezielle Container bestimmt. Der hier benutzte Container `javafx.scene.layout.StackPane` ordnet verschiedene Elemente übereinander an. Mit `getChildren()` werden die Knoten im Container ermittelt. Ein neuer Knoten wird mit der Methode `add` hinzugefügt.

`javafx.scene.Scene` ist der Container für alle Inhalte des *SceneGraph*. Der hier verwendete Konstruktor erzeugt ein Scene-Objekt aus dem Wurzelknoten vom Typ `StackPane` mit einer bestimmten Breite und Höhe des Fensters.

Die Stage-Methode `void setScene(Scene scene)` legt die "Szene" fest, die auf der "Bühne" gespielt wird.

Die Stage-Methode `setTitle` setzt den Fenstertitel, `setResizable` legt fest, ob die Größe des Fensters verändert werden kann, `show` macht das Fenster sichtbar.

`setOnCloseRequest` bestimmt, was beim Schließen des Fensters (Close-Button oben rechts) geschehen soll. Im Beispiel wird das Ereignis sofort *konsumiert*, ohne dass etwas geschieht, sodass das Fenster auf diesem Weg nicht geschlossen werden kann.

Die Methode `setText` der Klasse `javafx.scene.control.Button` beschriftet den Button. `setOnAction` bestimmt, was beim Klicken passieren soll. Die `javafx.application.Platform`-Methode `exit` beendet die Anwendung.

Das Programm kann nun entweder in *IntelliJ IDEA* nach Konfiguration, wie in Kapitel 28.1 beschrieben, gestartet werden oder im Terminal mit dem Kommando `java ...` aufgerufen werden.



**Abbildung 28-2:** Das erste Beispiel

## 28.3 Beispiel Brutto-Rechner

Wir entwickeln im Folgenden ein Programm, das zu einem Nettobetrag und einem Steuersatz den Bruttbetrag berechnet.



**Abbildung 28-3:** Brutto-Rechner

Bei der Eingabe des Nettobetrags kann das Dezimaltrennzeichen als Punkt oder Komma eingegeben werden. Nach fehlerhafter Eingabe wird das Feld geleert und erhält den Eingabefokus. Sobald das Eingabefeld oder die Auswahlbox zur Auswahl des Steuersatzes angeklickt wird, wird ein evtl. vorher berechneter Bruttbetrag gelöscht.

Wir werden zunächst das komplette Layout programmieren und später in einer zweiten Version das Design-Tool *Scene Builder*, das die Oberfläche in XML-Form beschreibt, einsetzen.

### Programmatischer Zusammenbau

Der SceneGraph besteht aus den Containern `javafx.scene.layout.AnchorPane` und `javafx.scene.layout.GridPane` und den Komponenten `Label`, `TextField`, `ComboBox` und `Button`, die alle zum Paket `javafx.scene.control` gehören.

Mit `AnchorPane` können Komponenten in einem bestimmten Abstand vom Fensterrand positioniert werden. Hierzu nutzen wir die beiden Klassenmethoden `setLeftAnchor` und `setTopAnchor`. Zudem legen wir die Größe des Containers fest. In einem `AnchorPane` können Komponenten wie z. B. Labels und Buttons auch absolut mit `setLayoutX` und `setLayoutY` positioniert werden.

Mit `GridPane` können Komponenten in einem Raster analog zum `GridLayout` aus Swing angeordnet werden. Hierzu wird die folgende Methode verwendet:

```
void add(Node child, int columnIndex, int rowIndex)
```

Abstände zwischen Spalten und Zeilen können mit `setHgap` bzw. `setVgap` festgelegt werden. Die Funktionalität von `Label`, `TextField` und `ComboBox` ist aus Swing bekannt.

Mit

```
cbUmsatzsteuer.getItems().addAll(items);
```

werden die Einträge für die `ComboBox cbUmsatzsteuer` festgelegt.

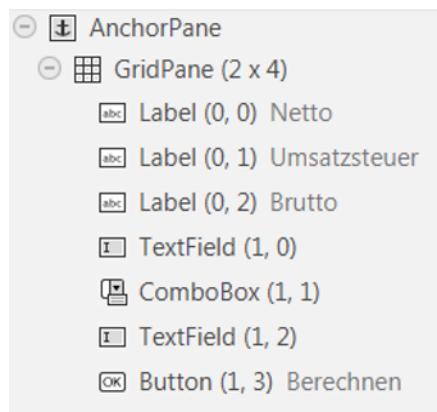
Mit

```
cbUmsatzsteuer.getSelectionModel().select(0);
```

wird der erste Eintrag ausgewählt.

Für das Textfeld zur Eingabe des Nettobetrags und für die `ComboBox` wird jeweils ein *Event-Handler* registriert, der auf das Drücken der Maustaste innerhalb der jeweiligen Komponente reagiert: `setOnMousePressed`

Mit dem Drücken des Buttons (`setOnAction`) erfolgt die Berechnung.



**Abbildung 28-4:** Der SceneGraph der Anwendung BruttoRechner

```
package brutto1;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
```

```
public class BruttoRechner extends Application {  
    private TextField tfNetto;  
    private TextField tfBrutto;  
    private ComboBox<String> cbUmsatzsteuer;  
    private final String[] items = { "19 %", "16 %", "7 %" };  
    private final int[] values = { 19, 16, 7 };  
  
    @Override  
    public void start(Stage stage) {  
        AnchorPane anchorPane = new AnchorPane();  
        GridPane gridPane = new GridPane();  
  
        tfNetto = new TextField();  
        tfNetto.setOnMousePressed(event -> tfBrutto.setText(""));  
  
        cbUmsatzsteuer = new ComboBox<>();  
        cbUmsatzsteuer.getItems().addAll(items);  
        cbUmsatzsteuer.getSelectionModel().select(0);  
        cbUmsatzsteuer.setOnMousePressed(event -> tfBrutto.setText(""));  
  
        tfBrutto = new TextField();  
        tfBrutto.setEditable(false);  
        tfBrutto.setId("brutto");  
  
        Button button = new Button();  
        button.setText("Berechnen");  
        button.setOnAction(e -> compute());  
  
        gridPane.setHgap(20);  
        gridPane.setVgap(20);  
        gridPane.add(new Label("Netto"), 0, 0);  
        gridPane.add(new Label("Umsatzsteuer"), 0, 1);  
        gridPane.add(new Label("Brutto"), 0, 2);  
        gridPane.add(tfNetto, 1, 0);  
        gridPane.add(cbUmsatzsteuer, 1, 1);  
        gridPane.add(tfBrutto, 1, 2);  
        gridPane.add(button, 1, 3);  
  
        anchorPane.getChildren().add(gridPane);  
        anchorPane.setPrefWidth(400);  
        anchorPane.setPrefHeight(250);  
        AnchorPane.setLeftAnchor(gridPane, 30.);  
        AnchorPane.setTopAnchor(gridPane, 30.);  
  
        Scene scene = new Scene(anchorPane);  
        scene.getStylesheets().add(getClass()  
            .getResource("BruttoRechner.css").toExternalForm());  
        stage.setScene(scene);  
        stage.setTitle("Brutto-Rechner");  
        stage.show();  
    }  
  
    private void compute() {  
        String input = tfNetto.getText().trim();  
        input = input.replace(",", ".");  
  
        double netto;
```

```
try {
    netto = Double.parseDouble(input);
} catch (NumberFormatException e) {
    tfNetto.setText("");
    tfNetto.requestFocus();
    return;
}

int idx = cbUmsatzsteuer.getSelectionModel().getSelectedIndex();
int value = values[idx];
double brutto = netto * (1 + value / 100.);
tfBrutto.setText(String.format("%.2f", brutto));
}

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

Optische Aspekte einer JavaFX-Anwendung können wie bei HTML-Seiten mit *CSS (Cascading Style Sheets)* bestimmt werden.

Das im Beispiel verwendete *Stylesheet* kann wie folgt registriert werden:

```
scene.getStylesheets().add(getClass()
    .getResource("BruttoRechner.css").toExternalForm());
```

Um zu erfahren, wie die Anwendung ohne CSS aussieht, können Sie diese Anweisung auf Kommentar setzen.

Das Stylesheet *BruttoRechner.css*:

```
AnchorPane {
    -fx-background-color: lightblue;
}

.label {
    -fx-font-size: 18.0;
    -fx-font-weight: bold;
}

.button {
    -fx-font-size: 18.0;
    -fx-font-weight: bold;
    -fx-background-color: blue;
    -fx-text-fill: white;
    -fx-background-radius: 8.0;
}

.button:pressed {
    -fx-background-color: lightblue;
}

#brutto {
    -fx-text-fill: red;
}
```

Mit Hilfe von *Selektoren* werden die zu formatierenden Elemente ausgewählt und Eigenschaftswerte zugewiesen. In der Datei finden Sie vier unterschiedliche Arten von Selektoren.

Der *Type-Selektor* (hier `AnchorPane`) stylt die Container vom Typ `AnchorPane`.

*Style-Klassen* werden durch einen führenden Punkt kenntlich gemacht (hier `.label` und `.button`). Die Klassen `Label` und `Button` haben diese Style-Klassen bereits voreingestellt. Jeder Komponente können weitere eigene Style-Klassen zugeordnet werden.

Style-Klassen können mit einem *Zustand* (hier `:pressed`) verbunden werden. Im Beispiel wird der Button im gedrückten Zustand hierdurch gestyliert.

Individuelle Komponenten können über die eindeutige *Id* der Komponente (hier `#brutto`) gestyliert werden. Die *Id* wird mit der Methode `setId` gesetzt.

Im Beispiel: `tfBrutto.setId("brutto");`

## Deklarativer Aufbau mit FXML

### Scene Builder

Mit dem Tool *Scene Builder* wird als Ergebnis des Design-Prozesses eine XML-Datei, die das Layout der Anwendung beschreibt, erzeugt.

Das Fenster des *Scene Builder* ist in die folgenden Bereiche unterteilt:

- *Menüleiste* oben,
- *Editor* in der Mitte,
- *Komponentenbibliothek*, *Komponentenbaum* (`SceneGraph`) und *Controller* links,
- *Inspector*-Fenster rechts mit den Teilbereichen *Properties*, *Layout* und *Code*.

Der Scene Builder speichert das Ergebnis in einer Datei mit der Endung `.fxml` (hier `BruttoRechner.fxml`), die das Layout in einem besonderen XML-Format beschreibt: *FXML (JavaFX Markup Language)*.

Im Folgenden sind die Schritte kurz beschrieben, die in dieser Reihenfolge im *Scene Builder* durchzuführen sind.

1. Container `AnchorPane` in den Editor ziehen, falls er noch nicht im Komponentenbaum vorhanden ist.
2. Layout `AnchorPane`: Pref Width: 400, Pref Height: 250
3. Container `GridPane` in den Editor ziehen.

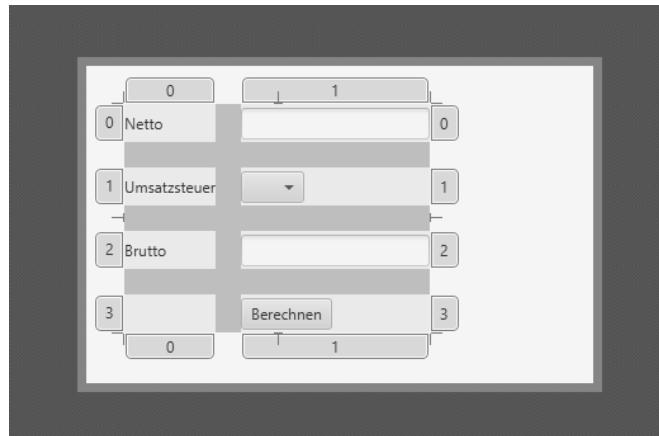


Abbildung 28-5: Editor im *Scene Builder*

4. Im Komponentenbaum mit rechter Maustaste auf GridPane klicken. Dann unter GridPane "Add row below" wählen.
5. Layout GridPane: AnchorPane Constraints links: 30, oben: 30; Hgap: 20, Vgap: 20; Layout X: 0, Layout Y: 0
6. Control Label in Zelle mit Spalte 0, Zeile 0 ziehen.
7. Properties Label: Text: Netto
8. Control Label in Zelle mit Spalte 0, Zeile 1 ziehen.
9. Properties Label: Text: Umsatzsteuer
10. Control Label in Zelle mit Spalte 0, Zeile 2 ziehen.
11. Properties Label: Text: Brutto
12. Control TextField in Zelle mit Spalte 1, Zeile 0 ziehen.
13. Code TextField: fx:id: tfNetto
14. Control ComboBox in Zelle mit Spalte 1, Zeile 1 ziehen.
15. Code ComboBox: fx:id: cbUmsatzsteuer
16. Control TextField in Zelle mit Spalte 1, Zeile 2 ziehen.
17. Code TextField: fx:id: tfBrutto
18. Properties TextField: Editable ohne Häkchen; Id (im Bereich JavaFX CSS): brutto
19. Control Button in Zelle mit Spalte 1, Zeile 3 ziehen.
20. Properties Button: Text: Berechnen
21. Auf Spalte 0 des GridPane im Editorfenster klicken.
22. Layout ColumnConstraints: Pref Width: USE\_COMPUTED\_SIZE

23. Auf Spalte 1 des GridPane im Editorfenster klicken.
24. Layout ColumnConstraints: Pref Width: USE\_COMPUTED\_SIZE
25. Auf ComboBox (1, 1) klicken.
26. Layout ComboBox: Pref Width: USE\_COMPUTED\_SIZE
27. Auf TextField (1, 0) klicken.
28. Code TextField: On Mouse Pressed: handleNettoMousePressed
29. Auf ComboBox (1, 1) klicken.
30. Code ComboBox: On Mouse Pressed: handleUmsatzsteuerMousePressed
31. Auf Button (1, 3) klicken.
32. Code Button: On Action: handleButtonAction
33. Auf Bereich Controller links unten klicken.
34. Controller class: brutto2.BruntoRechnerController

Das Layout kann mit *Preview > Show Preview in Window* getestet werden.

### Die FXML-Datei BruttoRechner.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
    minWidth="-Infinity" prefHeight="250.0" prefWidth="400.0"
    xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="brutto2.BruntoRechnerController">

    <children>
        <GridPane hgap="20.0" vgap="20.0" AnchorPane.leftAnchor="30.0"
            AnchorPane.topAnchor="30.0">
            <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0"/>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0"/>
            </columnConstraints>
            <rowConstraints>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES"/>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES"/>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES"/>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES"/>
            </rowConstraints>
            <children>
                <Label text="Netto"/>
                <Label text="Umsatzsteuer" GridPane.rowIndex="1"/>
                <Label text="Brutto" GridPane.rowIndex="2"/>
                <TextField fx:id="tfNetto" onMousePressed="#handleNettoMousePressed"
                    GridPane.columnIndex="1"/>
                <ComboBox fx:id="cbUmsatzsteuer"
                    onMousePressed="#handleUmsatzsteuerMousePressed"
                    GridPane.columnIndex="1" GridPane.rowIndex="1"/>
                <TextField id="brutto" fx:id="tfBrutto" editable="false"
                    GridPane.columnIndex="1" GridPane.rowIndex="2"/>
            </children>
        </GridPane>
    </children>
</AnchorPane>
```

```
<Button mnemonicParsing="false" onAction="#handleButtonAction"
    text="Berechnen" GridPane.columnIndex="1"
    GridPane.rowIndex="3"/>
</children>
</GridPane>
</children>
</AnchorPane>
```

Unter `fx:controller` ist im `AnchorPane`-Tag die Klasse

```
brutto2.BruttoRechnerController
```

eingetragen.

Die noch zu erstellende ausführbare Klasse `BruttoRechner` erzeugt zur Laufzeit beim Laden der FXML-Datei eine Instanz des Controllers und verbindet diese mit der FXML-Datei. Die unter `fx:id` eingetragenen Ids werden für die Verknüpfung mit dem Controller benötigt.

### Der Controller `BruttoRechnerController`

Über den Menüpunkt *View > Show Sample Controller Skeleton* im *Scene Builder* kann eine Quellcode-Vorlage kopiert werden. Diese ist dann noch anzupassen.

```
package brutto2;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.ComboBox;
import javafx.scene.control.TextField;
import javafx.scene.input.MouseEvent;

import java.net.URL;
import java.util.ResourceBundle;

public class BruttoRechnerController implements Initializable {
    @FXML
    private TextField tfNetto;

    @FXML
    private ComboBox<String> cbUmsatzsteuer;

    @FXML
    private TextField tfBrutto;

    private final String[] items = { "19 %", "16 %", "7 %" };
    private final int[] values = { 19, 16, 7 };

    @FXML
    void handleButtonAction(ActionEvent event) {
        compute();
    }
}
```

```

@FXML
void handleNettoMousePressed(MouseEvent event) {
    tfBrutto.setText("");
}

@FXML
void handleUmsatzsteuerMousePressed(MouseEvent event) {
    tfBrutto.setText("");
}

@Override
public void initialize(URL url, ResourceBundle bundle) {
    cbUmsatzsteuer.getItems().addAll(items);
    cbUmsatzsteuer.getSelectionModel().select(0);
}

private void compute() {
    String input = tfNetto.getText().trim();
    input = input.replace(",", ".");

    double netto;
    try {
        netto = Double.parseDouble(input);
    } catch (NumberFormatException e) {
        tfNetto.setText("");
        tfNetto.requestFocus();
        return;
    }

    int idx = cbUmsatzsteuer.getSelectionModel().getSelectedIndex();
    int value = values[idx];
    double brutto = netto * (1 + value / 100.);
    tfBrutto.setText(String.format("%.2f", brutto));
}
}

```

Mit Hilfe der Annotation `@FXML` wird eine Verbindung zwischen den Attributen und den Komponenten in der FXML-Datei über die Namen (Attributname stimmt mit Wert von `fx:id` überein) hergestellt. Ebenso werden gleichnamige Methoden verbunden.

Der Controller implementiert das Interface `javafx.fxml.Initializable`. Die Methode `initialize` wird vom `FXMLLoader` (siehe Klasse `BruttoRechner`) aufgerufen, nachdem alle Attribute des Controllers initialisiert wurden. Im Beispiel wird hier die `ComboBox` mit Einträgen versorgt.

## Die Anwendung BruttoRechner

```

package brutto2;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;

```

```
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.io.IOException;

public class BruttoRechner extends Application {
    @Override
    public void start(Stage stage) throws IOException {
        Parent root = FXMLLoader.load(getClass()
            .getResource("BruttoRechner.fxml"));

        Scene scene = new Scene(root);
        scene.getStylesheets().add(getClass()
            .getResource("BruttoRechner.css").toExternalForm());

        stage.setScene(scene);
        stage.setTitle("Brutto-Rechner");
        stage.show();
    }

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

Die `javafx.fxml.FXMLLoader`-Methode `load` lädt die FXML-Datei:

```
Parent root = FXMLLoader.load(getClass()
    .getResource("BruttoRechner.fxml"));
```

## 28.4 Asynchrone Verarbeitung

Alle Benutzereingaben und Änderungen der Komponenten der Oberfläche (Scene-Graph) werden alleine vom *JavaFX-Application-Thread* (UI-Thread) ausgeführt. Wenn dieser zu sehr beschäftigt ist, wird die Oberfläche träge und ist dann nicht mehr bedienbar.

Deshalb sollten alle länger laufenden Aktivitäten in eigene Threads verlagert werden. Aktualisierungen der Oberfläche aus einem solchen Thread heraus (z. B. Setzen eines Label-Textes) sind auf direktem Weg jedoch nicht möglich. In einem solchen Fall wird zur Laufzeit eine Ausnahme geworfen.

Abhilfe schafft hier analog zur `EventQueue`-Methode `invokeLater` in Swing die folgende `javafx.application.Platform`-Methode:

```
static void runLater(Runnable runnable)
```

Die beiden folgenden Programme demonstrieren den Sachverhalt. Mit "Start" wird eine lang laufende Aktivität gestartet, deren Fortschritt visualisiert wird. Während der Verarbeitung kann die Aktivität mit "Stop" abgebrochen werden. Am Ende der Berechnung werden das Ergebnis und die benötigte Rechenzeit in Millisekunden ausgegeben.

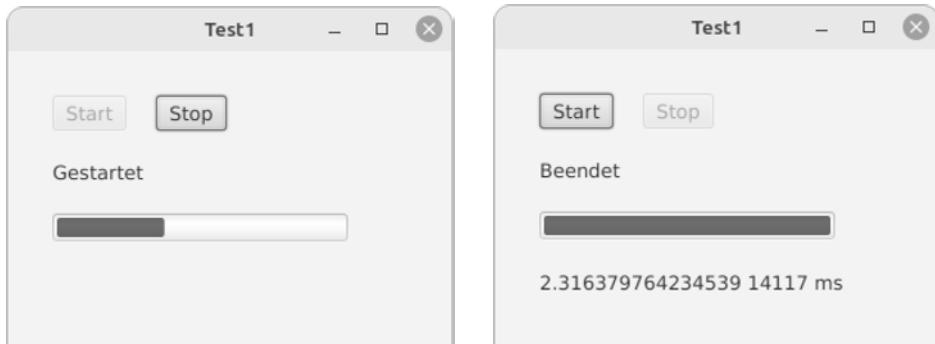


Abbildung 28-6: Die Berechnung läuft (links) und liefert ein Ergebnis (rechts)

```
package async;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Test1 extends Application {
    private Button start;
    private Button stop;
    private Label lbMessage;
    private Label lbResult;
    private ProgressBar pgb;
    private Thread thread;
    private final int MAX = 100_000_000;
    private final int INTERVAL = 1_000;

    @Override
    public void start(Stage stage) {
        start = new Button();
        start.setText("Start");
        start.setOnAction(e -> doWork());

        stop = new Button();
        stop.setText("Stop");
        stop.setOnAction(e -> stopWork());
    }
}
```

```
stop.setDisable(true);

lbMessage = new Label();
lbResult = new Label();
pgb = new ProgressBar(0);
pgb.setPrefWidth(200);

HBox hbox = new HBox();
hbox.setSpacing(20);
hbox.getChildren().addAll(start, stop);

VBox vbox = new VBox();
vbox.setSpacing(20);
vbox.getChildren().addAll(hbox, lbMessage, pgb, lbResult);

AnchorPane anchorPane = new AnchorPane();
anchorPane.getChildren().add(vbox);
AnchorPane.setLeftAnchor(vbox, 30.);
AnchorPane.setTopAnchor(vbox, 30.);

stage.setScene(new Scene(anchorPane, 300, 200));
stage.setTitle("Test1");
stage.show();
}

private void doWork() {
    start.setDisable(true);
    stop.setDisable(false);

    thread = new Thread() {
        double value;

        public void run() {
            Platform.runLater(() -> lbMessage.setText("Gestartet"));
            Platform.runLater(() -> lbResult.setText(""));

            double z = 0;
            long begin = System.currentTimeMillis();
            for (int i = 0; i < MAX; i++) {
                if (isInterrupted())
                    break;

                z += Math.sin(i) + Math.cos(i);

                if (i % INTERVAL == 0) {
                    value = (double) i / MAX;
                    Platform.runLater(() -> pgb.setProgress(value));
                }
            }

            if (isInterrupted()) {
                Platform.runLater(() -> lbMessage.setText("Gestoppt"));
            } else {
                Platform.runLater(() -> pgb.setProgress(1));
                long end = System.currentTimeMillis();
                String result = z + " " + (end - begin) + " ms";
                Platform.runLater(() -> lbMessage.setText("Beendet"));
                Platform.runLater(() -> lbResult.setText(result));
            }
        }
    };
}
```

```
        Platform.runLater(() -> start.setDisable(false));
        Platform.runLater(() -> stop.setDisable(true));
    }
};

thread.setDaemon(true);
thread.start();
}

private void stopWork() {
    if (thread != null) {
        thread.interrupt();
        thread = null;
    }
}

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

Die Container `javafx.scene.layout.HBox` und `javafx.scene.layout.VBox` ordnen ihre Komponenten nebeneinander bzw. untereinander an. Mit `setSpacing` wird der Abstand zwischen den Komponenten festgelegt.

`javafx.scene.control.ProgressBar` zeigt den Fortschritt mit Hilfe der Lnge des Fortschrittsbalkens an. Die `ProgressBar`-Methode `setProgress` setzt den Fortschrittswert als `double`-Zahl zwischen 0 und 1 (das entspricht 0 % bzw. 100 %). Der hier verwendete Konstruktor erzeugt einen Fortschrittsbalken mit einem vorgegebenen Wert.

Damit mit dem Schließen des Fensters eine evtl. noch laufende Berechnung abgebrochen werden kann, muss der Thread, der die Berechnung ausführt, als *Daemon-Thread* deklariert werden (siehe Kapitel 26).

## Programmvariante

Die nun folgende Variante nutzt die abstrakte Klasse `javafx.concurrent.Task<V>` für die einmalige Ausführung einer länger dauernden Aktivität.

## Die abstrakte Methode

abstract V call() throws Exception

muss überschrieben werden. Hier wird die eigentliche Arbeit geleistet und das Ergebnis zurückgegeben.

Des Weiteren nutzen wir im Beispiel die Methoden `succeeded` und `cancelled`, die ausgeführt werden, wenn die Arbeit erfolgreich beendet wurde bzw. wenn sie mittels der Task-Methode `cancel` abgebrochen wurde.

Die Methode `isCancelled` liefert `true`, wenn die Arbeit abgebrochen wurde.

Hier machen wir nun Gebrauch von *JavaFX Properties* und *Binding*. Alle Standardkomponenten in JavaFX stellen Properties bereit.

*Property Binding* ist ein mächtiger Mechanismus, mit dem zwei Properties automatisch synchronisiert werden können. Ändert sich eine Property, wird die andere automatisch aktualisiert.

Siehe auch "Erläuterndes Beispiel zu Properties und Binding" in diesem Abschnitt.

Hier zunächst der Quellcode:

```
package async;

import javafx.application.Application;
import javafx.concurrent.Task;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Test2 extends Application {
    private Button start;
    private Button stop;
    private Label lbMessage;
    private Label lbResult;
    private ProgressBar pgb;
    private Task<String> task;
    private final int MAX = 100_000_000;
    private final int INTERVAL = 1_000;

    @Override
    public void start(Stage stage) {
        start = new Button();
        start.setText("Start");
        start.setOnAction(e -> doWork());

        stop = new Button();
        stop.setText("Stop");
        stop.setOnAction(e -> stopWork());
        stop.setDisable(true);

        lbMessage = new Label();
        lbResult = new Label();
        pgb = new ProgressBar(0);
        pgb.setPrefWidth(200);

        HBox hbox = new HBox();
```

```
hbox.setSpacing(20);
hbox.getChildren().addAll(start, stop);

VBox vbox = new VBox();
vbox.setSpacing(20);
vbox.getChildren().addAll(hbox, lbMessage, pgb, lbResult);

AnchorPane anchorPane = new AnchorPane();
anchorPane.getChildren().add(vbox);
AnchorPane.setLeftAnchor(vbox, 30.);
AnchorPane.setTopAnchor(vbox, 30.);

stage.setScene(new Scene(anchorPane, 300, 200));
stage.setTitle("Test2");
stage.show();
}

private void doWork() {
    task = new Task<>() {
        @Override
        protected String call() throws Exception {
            updateMessage("Gestartet");

            double z = 0;
            long begin = System.currentTimeMillis();
            for (int i = 0; i < MAX; i++) {
                if (isCancelled())
                    break;

                z += Math.sin(i) + Math.cos(i);

                if (i % INTERVAL == 0)
                    updateProgress(i, MAX);
            }

            String result = "";
            if (!isCancelled()) {
                updateProgress(MAX, MAX);
                long end = System.currentTimeMillis();
                long duration = end - begin;

                result = z + " " + duration + " ms";
            }
            return result;
        }

        @Override
        protected void succeeded() {
            super.succeeded();
            updateMessage("Beendet");
        }
    };
}
```

```
    @Override
    protected void cancelled() {
        super.cancelled();
        updateMessage("Gestoppt");
    }
};

lbMessage.textProperty().bind(task.messageProperty());
lbResult.textProperty().bind(task.valueProperty());
pgb.progressProperty().bind(task.progressProperty());
start.disableProperty().bind(task.runningProperty());
stop.disableProperty().bind(task.runningProperty().not());

Thread t = new Thread(task);
t.setDaemon(true);
t.start();
}

private void stopWork() {
    task.cancel();
}

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

Die Task-Methode `updateMessage` aktualisiert die *Message-Property* der Task.

Mit

```
lbMessage.textProperty().bind(task.messageProperty());
```

ist die *Text-Property* des Labels `lbMessage` an die *Message-Property* der Task gebunden. Damit wird die Anzeige automatisch mit der Message der Task synchronisiert.

Ähnliches gilt für

```
lbResult.textProperty().bind(task.valueProperty());
```

und

```
pgb.progressProperty().bind(task.progressProperty());
```

`valueProperty()` liefert das Ergebnis der Berechnung (Rückgabewert von `call`). Die Fortschrittsanzeige ist an den Fortschritt der Task gebunden.

Die Task-Methode `updateProgress` aktualisiert die *Progress-Property* der Task:

```
void updateProgress(long workDone, long max)
```

Der Quotient `workDone/max` entspricht dem Fortschritt in Prozent.

Das wechselseitige Aktivieren und Deaktivieren der beiden Buttons kann ebenso mit *Bindings* realisiert werden:

```
start.disableProperty().bind(task.runningProperty());
stop.disableProperty().bind(task.runningProperty().not());
```

Mit `not()` wird ein boolescher Wert negiert.

Ein einmal gebundener Wert kann nicht mit Aufruf einer `set`-Methode neu gesetzt werden. Das gelingt erst, wenn man die Bindung wieder aufhebt, z. B.

```
lbMessage.textProperty().unbind();
```

## Erläuterndes Beispiel zu Properties und Binding

Das Beispiel zeigt,

- wie Veränderungen einer Property beobachtet werden können und
- wie die an eine Property gebundene zweite Property, die Änderung der ersten widerspiegelt.

Die Klasse `SimpleIntegerProperty` kapselt einen `int`-Wert. An dieser Property wird ein Listener angemeldet, der bei Wertänderung eine Meldung auf der Konsole ausgibt. Eine zweite Property vom selben Typ wird an die erste gebunden. Wird nun der Wert der ersten geändert, ändert sich auch der Wert der zweiten.

```
package properties;

import javafx.beans.property.SimpleIntegerProperty;

public class PropertiesTest {
    public static void main(String[] args) {
        SimpleIntegerProperty prop1 = new SimpleIntegerProperty();
        prop1.addListener((observable, oldValue, newValue) ->
            System.out.println("prop1 geändert von " + oldValue + " nach " +
                newValue));
        prop1.set(1);
        prop1.set(2);

        SimpleIntegerProperty prop2 = new SimpleIntegerProperty();

        // prop2 wird an prop1 gebunden
        prop2.bind(prop1);

        System.out.println("prop2: " + prop2.get());
        prop1.set(3);
        System.out.println("prop2: " + prop2.get());
    }
}
```

Ausgabe:

```
prop1 geändert von 0 nach 1
prop1 geändert von 1 nach 2
prop2: 2
prop1 geändert von 2 nach 3
prop2: 3
```

## 28.5 Diagramme

JavaFX bietet *Diagramme (Charts)* in verschiedenen Formen an.

Die folgenden Programmbeispiele demonstrieren Kreis- und Balkendiagramme. Diagramme können wie alle anderen visuellen Komponenten dem *SceneGraph* als Node hinzugefügt werden.

Wir beginnen mit einem *Kreisdiagramm* (`javafx.scene.chart.PieChart`), das den prozentualen Anteil einer Kategorie an einer Gesamtmenge visualisiert. In diesem Beispiel werden die Anzahl Einkäufe in den verschiedenen Abteilungen eines Warenhauses dargestellt:

Abteilung	Anzahl Einkäufe
Damenbekleidung	211
Elektrogeräte	166
Herrenbekleidung	142
Kinderbekleidung	158
Lebensmittel	192
Spielwaren	131

```
package charts;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Stage;

public class Chart1 extends Application {
    private ObservableList<PieChart.Data> dataList;

    @Override
    public void start(Stage stage) {
        dataList = FXCollections.observableArrayList();
        initialize();
    }
}
```

```
AnchorPane pane = new AnchorPane();
PieChart chart = new PieChart();
chart.setData(dataList);
chart.setTitle("Anzahl Einkäufe");
chart.setPrefWidth(1200);
chart.setPrefHeight(800);

Label label = new Label();
label.setLayoutX(50);
label.setLayoutY(50);
label.setId("value");

for (PieChart.Data data : dataList) {
    Node node = data.getNode();

    String text = String.format("%8.0f %8.2f %%",
        data.getPieValue(), data.getPieValue() * 100. / sum());

    node.setOnMouseEntered(e -> label.setText(text));
    node.setOnMouseExited(e -> label.setText(""));
}

pane.getChildren().addAll(chart, label);

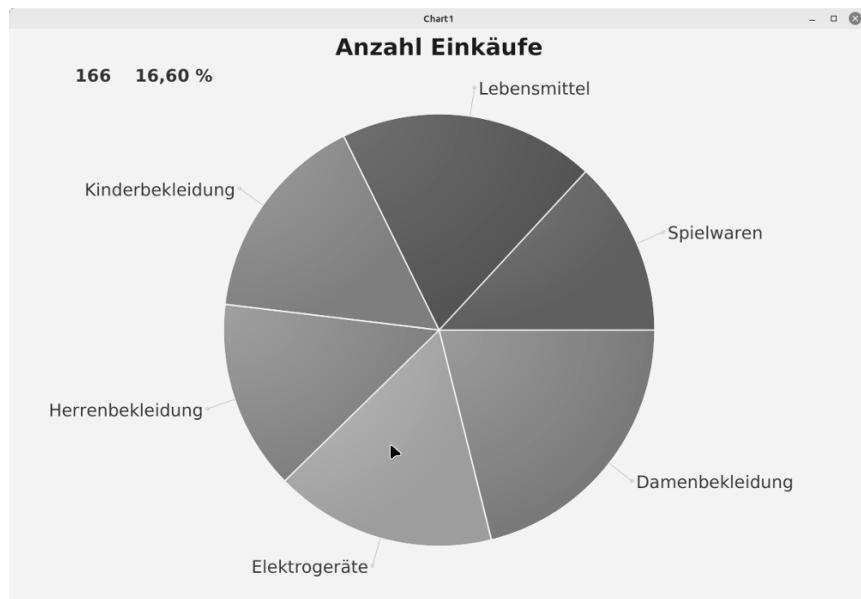
Scene scene = new Scene(pane);
scene.getStylesheets().add(getClass()
    .getResource("chart1.css").toExternalForm());

stage.setScene(scene);
stage.setTitle("Chart1");
stage.show();
}

private void initialize() {
    dataList.add(new PieChart.Data("Damenbekleidung", 211));
    dataList.add(new PieChart.Data("Elektrogeräte", 166));
    dataList.add(new PieChart.Data("Herrenbekleidung", 142));
    dataList.add(new PieChart.Data("Kinderbekleidung", 158));
    dataList.add(new PieChart.Data("Lebensmittel", 192));
    dataList.add(new PieChart.Data("Spielwaren", 131));
}

private double sum() {
    double sum = 0;
    for (PieChart.Data data : dataList) {
        sum += data.getPieValue();
    }
    return sum;
}

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



**Abbildung 28-7:** Ein Kreisdiagramm  
Der Mauszeiger zeigt auf das Segment "Elektrogeräte".

Die Daten des Diagramms werden in einer Liste vom Typ

```
javafx.collections.ObservableList<PieChart.Data>
```

hinterlegt, die mit der statischen Methode `observableArrayList` der Klasse `javafx.collections.FXCollections` erzeugt wird.

Mit der `PieChart`-Methode `setData` wird die Liste dem Diagramm hinzugefügt.

Beim Eintritt des Mauszeigers in ein Kreissegment werden zugehörige Daten (Anzahl und Prozentwert) in einem Label angezeigt. Die `PieChart.Data`-Methode `getNode` liefert den `Node`, der das jeweilige Kreissegment repräsentiert. Hier werden dann mit `setOnMouseEntered` und `setOnMouseExited` entsprechende *Event-Handler* registriert.

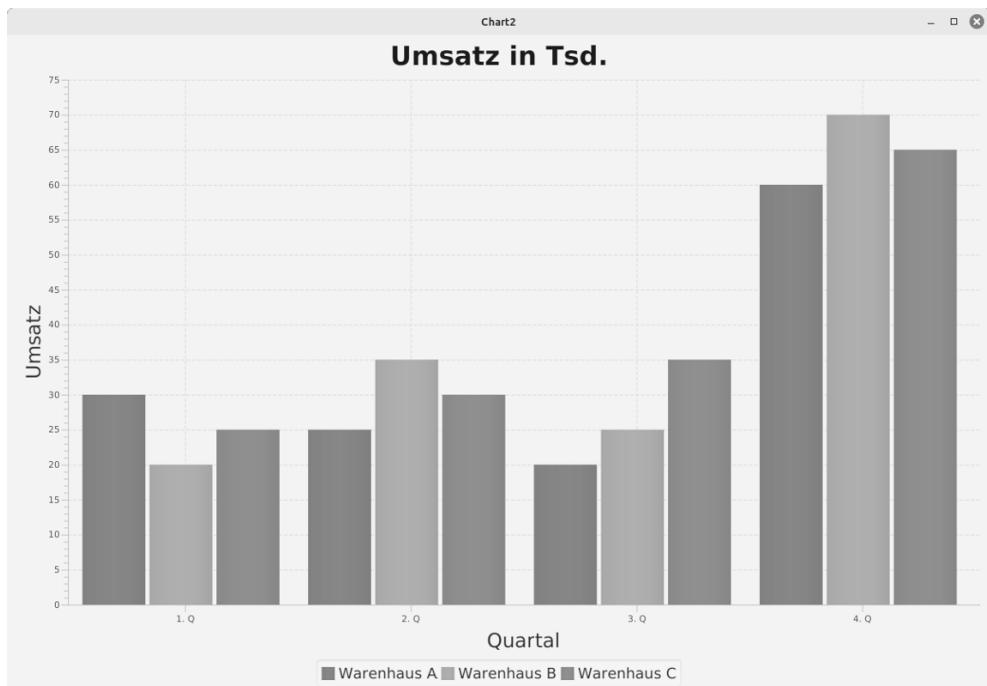
Diverse Gestaltungen werden in CSS beschrieben: Schriftbild, Sichtbarkeit der Legende, die Länge der Linie vom Kreissegment zum Label.

*chart1.css:*

```
.chart-title {  
    -fx-text-fill: blue;  
    -fx-font-weight: bold;  
    -fx-font-size: 32.0;  
}  
}
```

```
.chart-pie-label {  
    -fx-font-size: 24.0;  
}  
  
.chart {  
    -fx-legend-visible: false;  
    -fx-label-line-length: 40.0  
}  
  
#value {  
    -fx-font-weight: bold;  
    -fx-font-size: 24.0;  
}
```

Das zweite Beispiel zeigt die Entwicklung der Umsätze dreier Warenhäuser in einem Jahr an.



**Abbildung 28-8:** Ein Balkendiagramm  
Der Mauszeiger zeigt auf den ersten Balken links.

```
package charts;  
  
import javafx.application.Application;  
import javafx.scene.Node;  
import javafx.scene.Scene;
```

```
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Stage;

public class Chart2 extends Application {
    private XYChart.Series<String, Number> seriesA;
    private XYChart.Series<String, Number> seriesB;
    private XYChart.Series<String, Number> seriesC;

    @Override
    public void start(Stage stage) {
        seriesA = new XYChart.Series<>();
        seriesB = new XYChart.Series<>();
        seriesC = new XYChart.Series<>();

        AnchorPane pane = new AnchorPane();

        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("Quartal");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Umsatz");

        BarChart<String, Number> chart = new BarChart<>(xAxis, yAxis);
        chart.setTitle("Umsatz in Tsd.");
        chart.setPrefWidth(1200);
        chart.setPrefHeight(800);

        initialize();
        chart.getData().addAll(seriesA, seriesB, seriesC);

        Label label = new Label();
        label.setLayoutX(150);
        label.setLayoutY(100);
        label.setId("value");

        for (XYChart.Series<String, Number> series : chart.getData()) {
            for (XYChart.Data<String, Number> data : series.getData()) {
                Node node = data.getNode();
                String text = data.getXValue() + " " + series.getName() +
                    "\nUmsatz: " + data.getYValue();
                node.setOnMouseEntered(e -> label.setText(text));
                node.setOnMouseExited(e -> label.setText(""));
            }
        }

        pane.getChildren().addAll(chart, label);

        Scene scene = new Scene(pane);
        scene.getStylesheets().add(getClass()
            .getResource("chart2.css").toExternalForm());
        stage.setScene(scene);
        stage.setTitle("Chart2");
        stage.show();
    }
}
```

```

private void initialize() {
    seriesA.setName("Warenhaus A");
    seriesA.getData().add(new XYChart.Data<>("1. Q", 30));
    seriesA.getData().add(new XYChart.Data<>("2. Q", 25));
    seriesA.getData().add(new XYChart.Data<>("3. Q", 20));
    seriesA.getData().add(new XYChart.Data<>("4. Q", 60));

    seriesB.setName("Warenhaus B");
    seriesB.getData().add(new XYChart.Data<>("1. Q", 20));
    seriesB.getData().add(new XYChart.Data<>("2. Q", 35));
    seriesB.getData().add(new XYChart.Data<>("3. Q", 25));
    seriesB.getData().add(new XYChart.Data<>("4. Q", 70));

    seriesC.setName("Warenhaus C");
    seriesC.getData().add(new XYChart.Data<>("1. Q", 25));
    seriesC.getData().add(new XYChart.Data<>("2. Q", 30));
    seriesC.getData().add(new XYChart.Data<>("3. Q", 35));
    seriesC.getData().add(new XYChart.Data<>("4. Q", 65));
}
}

public static void main(String[] args) {
    launch(args);
}
}
}

```

Das *Balkendiagramm javafx.scene.chart.BarChart* wird mit X- und Y-Achse konstruiert:

`javafx.scene.chart.CategoryAxis` (Quartal) und  
`javafx.scene.chart.NumberAxis` (Umsatz).

Die Klasse `javafx.scene.chart.XYChart.Series` repräsentiert eine Datenserie (hier ein Warenhaus). Die Namen der Serien werden in der Legende dargestellt.

Eine Serie enthält Datenpunkte vom Typ `xychart.Data` jeweils mit x-Wert (Quartal) und y-Wert (Umsatz).

Mit

`chart.getData().addAll(seriesA, seriesB, seriesC);`

werden die drei Serien dem Diagramm hinzugefügt.

Für jeden Datenpunkt einer Serie werden analog zum vorhergehenden Beispiel *Event-Handler* registriert.

*chart2.css*:

```

.chart-title {
    -fx-text-fill: blue;
    -fx-font-weight: bold;
    -fx-font-size: 32.0;
}

```

```
.bar-chart {  
    -fx-bar-gap: 5;  
    -fx-category-gap: 25;  
}  
  
.chart-legend {  
    -fx-font-size: 18.0;  
}  
  
.axis-label {  
    -fx-font-size: 24.0;  
}  
  
.chart {  
    -fx-legend-side: bottom  
}  
  
#value {  
    -fx-font-weight: bold;  
    -fx-font-size: 20.0;  
}
```

## 28.6 Tabellen

Artikel		
Artikelnummer	Artikelbezeichnung	Artikelpreis
1001	Bezeichnung für Artikel 1001	1,00
1002	Bezeichnung für Artikel 1002	2,00
1003	Bezeichnung für Artikel 1003	3,00
1004	Bezeichnung für Artikel 1004	4,00
1005	Bezeichnung für Artikel 1005	5,00
1006	Bezeichnung für Artikel 1006	6,00
1007	Bezeichnung für Artikel 1007	7,00
1008	Bezeichnung für Artikel 1008	8,00
1009	Bezeichnung für Artikel 1009	9,00
1010	Bezeichnung für Artikel 1010	10,00
1011	Bezeichnung für Artikel 1011	11,00
1012	Bezeichnung für Artikel 1012	12,00
1013	Bezeichnung für Artikel 1013	13,00
1014	Bezeichnung für Artikel 1014	14,00
1015	Bezeichnung für Artikel 1015	15,00
1016	Bezeichnung für Artikel 1016	16,00
1017	Bezeichnung für Artikel 1017	17,00
1018	Bezeichnung für Artikel 1018	18,00
1019	Bezeichnung für Artikel 1019	19,00

Abbildung 28-9: Artikeltabelle

`javafx.scene.control.TableView` stellt Daten in Form einer Tabelle mit Spaltenüberschriften dar.

Als Modell wird eine `ObservableList` verwendet, deren Elemente den Zeilen der Tabelle entsprechen. Die Spaltenwerte entsprechen den Attributwerten des jeweiligen Elements. In unserem Beispiel verwenden wir Objekte vom Typ `Article`.

```
package table;

public class Article {
    private final int id;
    private final String name;
    private final double price;

    public Article(int id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}

package table;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.TableCell;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Stage;

public class ArticleTable extends Application {
    @Override
    public void start(Stage stage) {
        AnchorPane root = new AnchorPane();
        TableView<Article> tableView = new TableView<>();
        TableColumn<Article, Integer> colId = new TableColumn<>("Artikelnummer");
        TableColumn<Article, String> colName = new TableColumn<>(
                "Artikelbezeichnung");
        TableColumn<Article, Double> colPrice = new TableColumn<>("Artikelpreis");
    }
}
```

```
colId.setId("id");
colName.setId("name");
colPrice.setId("price");

tableView.getColumns().addAll(colId, colName, colPrice);
tableView.setColumnResizePolicy(TableView
    .CONSTRAINED_RESIZE_POLICY_FLEX_LAST_COLUMN);
colName.setMinWidth(400);

colId.setCellValueFactory(new PropertyValueFactory<>("id"));
colName.setCellValueFactory(new PropertyValueFactory<>("name"));
colPrice.setCellValueFactory(new PropertyValueFactory<>("price"));

colPrice.setCellFactory(column -> new TableCell<>() {
    protected void updateItem(Double item, boolean empty) {
        super.updateItem(item, empty);
        if (empty || item == null) {
            setText(null);
        } else {
            setText(String.format("%8.2f", item));
            setAlignment(Pos.CENTER_RIGHT);
        }
    }
});
tableView.setItems(getList());

root.getChildren().addAll(tableView);
root.setPrefWidth(800);
root.setPrefHeight(500);

AnchorPane.setTopAnchor(tableView, 12.);
AnchorPane.setLeftAnchor(tableView, 12.);
AnchorPane.setRightAnchor(tableView, 12.);
AnchorPane.setBottomAnchor(tableView, 12.);

Scene scene = new Scene(root);
scene.getStylesheets().add(getClass()
    .getResource("ArticleTable.css").toExternalForm());

stage.setTitle("Artikel");
stage.setScene(scene);
stage.show();
}

private ObservableList<Article> getList() {
    ObservableList<Article> list = FXCollections.observableArrayList();
    for (int i = 1; i <= 100; i++) {
        list.add(new Article(i + 1000, "Bezeichnung für Artikel " +
            (i + 1000), i));
    }
    return list;
}

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

TableColumn-Objekte stellen die Spalten der Tabelle mit Spaltenüberschrift dar, beispielsweise

```
 TableColumn<Article, Integer> colId = new TableColumn<>()  
        "Artikelnummer");
```

Der zweite Typparameter bestimmt den Datentyp der Spaltenwerte.

Die TableColumn-Objekte müssen mit den Attributen der Artikel-Objekte verknüpft werden. das geschieht hier mit Hilfe der PropertyValueFactory z. B. wie folgt:

```
 colId.setCellValueFactory(new PropertyValueFactory<>("id"));
```

Hier wird der Wert des Attributs id aus dem Article-Objekt einer Zeile ermittelt.

Die Darstellung der einzelnen Zellen einer Spalte kann angepasst werden. Das geschieht hier für die Spalte "Artikelpreis":

```
 colPrice.setCellFactory(...);
```

Der Dezimalwert wird formatiert und rechtsbündig ausgerichtet.

Article-Objekte werden mit der Methode getList erzeugt. Mit der TableView-Methode setItems wird die ObservableList als Datenmodell gesetzt.

Die Tabellendaten werden nach Artikelnummer, Artikelbezeichnung oder Artikelpreis sortiert, wenn der Benutzer auf die entsprechende Spaltenüberschrift klickt.

Die Spaltenüberschriften werden mit CSS ausgerichtet.

*ArtikelTabelle.css:*

```
.table-view .column-header#nr .label {  
    -fx-alignment: CENTER_LEFT;  
}  
  
.table-view .column-header#bez .label {  
    -fx-alignment: CENTER_LEFT;  
}  
  
.table-view .column-header#preis .label {  
    -fx-alignment: CENTER_RIGHT;  
}
```

## 28.7 Aufgaben

### Aufgabe 1

Realisieren Sie ein Programm ohne Verwendung von FXML, das Datensätze über ein Formular erfasst und diese in einer Datei im CSV-Format speichert.

Ein Artikelsatz besteht aus einer Artikelnummer (int-Wert), einer Bezeichnung und einem Preis (double-Wert). Alle Felder müssen erfasst werden. Fehlermeldungen sollen in einem Label angezeigt werden.

Im main-Programm soll in der Methode `init` ein `PrintWriter` erzeugt werden. Dieser soll in der Methode `stop` geschlossen werden.

Beispiel Artikelsatz:

4711;Hammer;5.99

Lösung: Paket erfassung1



## Aufgabe 2

Realisieren Sie eine Variante zu Aufgabe 1, indem Sie den *Scene Builder* und FXML nutzen. Damit im Controller der `PrintWriter` zum Schreiben genutzt werden kann, muss er zu Beginn der Methode `start` im main-Programm an den Konstruktor des Controllers übergeben werden. Aus diesem Grund ist das Vorgehen zum Laden der FXML-Datei im Vergleich zum Programm im Kapitel 28.3 geändert:

```
ErfassungController controller = new ErfassungController(writer);
FXMLLoader loader = new FXMLLoader(getClass().getResource("Erfassung.fxml"));
loader.setController(controller);
Parent root = loader.load();
```

Die FXML-Datei darf den Eintrag `fx:controller="..."` nicht enthalten.

Lösung: Paket erfassung2

## Aufgabe 3

Der Fortschritt beim Laden einer größeren Datei soll mit dem Fortschrittsbalken `ProgressBar` visualisiert werden. Die Dateiauswahl soll über einen Dialog (`javafx.stage.FileChooser`) erfolgen.

```
FileChooser fileChooser = new FileChooser();
File selectedFile = fileChooser.showOpenDialog(stage);
```

Die Fortschrittsbalkenanzeige soll in Abständen von 500 gelesenen Bytes aktualisiert werden.

*Lösung: Paket progress*

#### Aufgabe 4

Die Teilnehmer der Java-Vorlesung antworten auf die Frage, wie viele Objekte eine Applikation aus einer bestimmten Klasse erzeugen kann, wie folgt:

- a) Nur eins pro Konstruktor: 8
- b) Beliebig viele: 24
- c) Nur eins pro Klasse: 2
- d) Ein Objekt pro Variable: 6

Die Zahlen geben jeweils die Anzahl der Antworten an.

Realisieren Sie ein Programm, das dieses Ergebnis in Form eines Kreisdiagramms darstellt.

*Lösung: Paket chart*

#### Aufgabe 5

Ergänzen Sie das Programm im Kapitel 28.6 um die folgende Funktionalität:

Mehrere Tabellenzeilen können per Mausklick ausgewählt werden. Mit Klick auf den Button "Speichern" werden die ausgewählten Artikel im CSV-Format in eine Datei geschrieben (vgl. Aufgabe 1).

Hinweis:

Mit

```
tableView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

wird festgelegt, dass mehrere Tabellenzeilen ausgewählt werden können.

Mit

```
ObservableList<Artikel> selectedItems =
    tableView.getSelectionModel().getSelectedItems();
```

werden die ausgewählten Artikel in eine Liste übertragen.

*Lösung: Paket table*

#### Aufgabe 6

Das Spiel "Schiffe versenken" hat ein Spielfeld, das aus einem Raster von 12 x 12 grauen Feldern (Typ Label) besteht. Dahinter sind "Schiffe" mit zufälligen Positionen versteckt: vier Schiffe bestehen aus einem Feld, drei Schiffe aus zwei Feldern, zwei Schiffe aus drei Feldern und ein Schiff aus vier Feldern.

Wird auf ein Feld mit der Maus geklickt, ändert sich die graue Farbe in Weiß, wenn kein Schiff getroffen wurde (also nur Wasser), in Blau, falls an dieser Stelle ein Schiff liegt.

Der Punktestand wird am unteren Rand angezeigt. Wenn ein Schiff nicht getroffen wurde, wird ein Punkt abgezogen. Für jeden Treffer gibt es drei Punkte.

Es ist zu beachten, dass einmal verbuchte Punkte beim Anklicken desselben Feldes nicht nochmals verrechnet werden.

Die Aufgabe besteht darin, den im Begleitmaterial vorhandenen Quellcode zu verstehen und auszuprobieren.

*Lösung: Paket game*



## 29 Datenbankzugriffe mit JDBC

Ein *Datenbanksystem* besteht aus der Software zur Datenverwaltung (*DBMS*, *Datenbankmanagementsystem*) und den in ein oder mehreren *Datenbanken* gespeicherten Daten.

In *relationalen Datenbanksystemen* besteht eine Datenbank aus Tabellen mit einer festen Anzahl von Spalten (Attributen) und einer variablen Anzahl von Zeilen (Datensätzen).

In der Regel hat jede Tabelle einen *Primärschlüssel*. Hierbei handelt es sich um ein Attribut (oder eine Kombination von Attributen), durch dessen Wert ein Datensatz eindeutig identifiziert werden kann.

Primärschlüssel werden auch benutzt, um Beziehungen zwischen Tabellen herzustellen.

### Lernziele

In diesem Kapitel lernen Sie

- wie eine Datenbank mit Tabellen erstellt werden kann,
- wie eine Verbindung zur Datenbank in einem Java-Programm hergestellt werden kann und
- wie Daten in Tabellen eingefügt, geändert, gelöscht und abgefragt werden können.

### 29.1 Voraussetzungen

Es existiert eine Vielzahl von relationalen Datenbankmanagementsystemen auf dem Markt – kommerzielle Lizenzprodukte und Open-Source-Systeme.

In diesem Kapitel verwenden wir das DBMS *SQLite*.

SQLite ist weit verbreitet (z. B. standardmäßig in Android-Geräten verfügbar), kostenfrei zu beziehen und leicht zu installieren. Eine SQLite-Datenbank ist eine lokale Datenbank, die ohne Server betrieben und über die weit verbreitete Datenbanksprache SQL (Structured Query Language) bearbeitet werden kann.

### JDBC

Das Paket `java.sql` bietet eine Programmierschnittstelle (API) für den Zugriff auf relationale Datenbanken mit Hilfe von SQL. Die hierzu erforderlichen Klassen und Methoden werden als *JDBC API* bezeichnet.

Der Name *JDBC* wird auch als Abkürzung für *Java Database Connectivity* verwendet. Ein Programm kann mittels JDBC unabhängig vom verwendeten Datenbankmanagementsystem entwickelt werden, sofern man sich an den SQL-Standard

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_29](https://doi.org/10.1007/978-3-658-43574-5_29).

hält. Somit ist ein Wechsel des DBMS ohne Änderung des Java-Programms möglich.

Die Datenbankanbindung wird über einen DBMS-spezifischen *JDBC-Treiber* realisiert.

Dieser Treiber implementiert das JDBC-API, übersetzt die JDBC-Aufrufe in DBMS-spezifische Datenbank-Befehle und leitet sie an das DBMS zur Ausführung weiter.

## SQLite

Der JDBC-Treiber für SQLite kann von der Website

<https://github.com/xerial/sqlite-jdbc/releases>

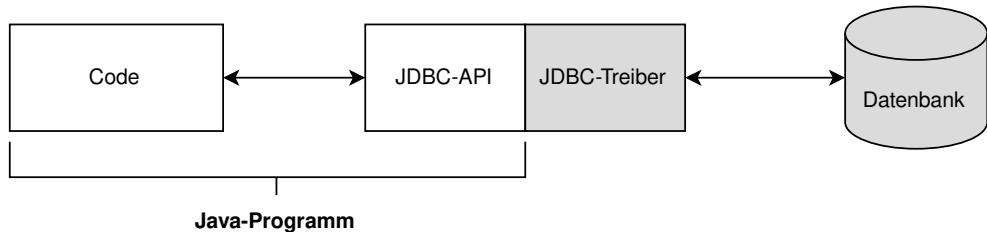
heruntergeladen werden.

Die jar-Datei `sqlite-jdbc-3.42.0.0.jar` muss zur Ausführung der Programme mit DB-Zugriff in die Java-Projekte eingebunden werden. Diese Datei enthält alles, was man für das Erstellen von und den Zugriff auf SQLite-Datenbanken mit Java braucht.

Das Bgleitmaterial enthält diese jar-Datei im Verzeichnis *kap29/lib*.

*DB Browser for SQLite* ist ein visuelles Open-Source-Tool zum Erstellen, Entwerfen und Bearbeiten von SQLite-Datenbanken. Es kann unabhängig von Java verwendet werden:

<https://sqlitebrowser.org>



**Abbildung 29-1:** JDBC-Architektur

## 29.2 Datenbank erstellen

Die neue Datenbank *mydb.db* soll die Tabelle *artikel* mit den Feldern *id* (Artikelnummer), *name* (Artikelbezeichnung), *preis* (Artikelpreis) und *menge* (Lagermenge) enthalten.

SQLite besitzt andere Datentypen als Java, z. B. `integer` für ganze Zahlen, `text` für Zeichenketten und `real` für Fließkommazahlen.

Das Feld *id* ist der *Primärschlüssel* und wird mit *primary key* gekennzeichnet. Das bedeutet, dass die Werte von *id* in verschiedenen Zeilen der Tabelle unterschiedlich sein müssen. Mit *id* kann also eindeutig auf einen bestimmten Artikel zugegriffen werden.

Die SQL-Anweisung zum Anlegen dieser Tabelle lautet:

```
create table if not exists artikel (
    id integer primary key,
    name text,
    preis real,
    menge integer
)
```

Die optionale Klausel "if not exists" bedeutet, dass die Tabelle nur dann angelegt wird, wenn sie noch nicht existiert.

Als Erstes muss eine Verbindung zur Datenbank hergestellt werden.

Die Methode

```
static Connection getConnection(String url) throws SQLException
```

der Klasse `java.sql.DriverManager` stellt eine Verbindung zur Datenbank her und liefert ein Objekt vom Typ des Interface `java.sql.Connection`.

Die `Connection`-Methode

```
void close() throws SQLException
```

schließt die Verbindung zur Datenbank.

`Connection` erweitert `AutoCloseable`. Deshalb kann *try with resources* eingesetzt werden.

`url` ist der Pfad für den Ablageort der Datenbank:

```
jdbc:sqlite:pfad/dbname.db
```

Hier verwenden wir: `jdbc:sqlite:mydb.db`

`mydb.db` liegt also im Verzeichnis, in dem die Anwendung aufgerufen wird.

## Die Connection-Methode

```
Statement createStatement() throws SQLException
```

erzeugt ein Objekt vom Typ des Interface `java.sql.Statement`. Dieses Objekt repräsentiert eine SQL-Anweisung.

## Die Statement-Methode

```
int executeUpdate(String sql) throws SQLException
```

führt eine SQL-Anweisung aus, die die Datenbank (Tabellenstruktur oder Daten) verändert. Hier wird sie benutzt, um die Tabelle anzulegen.

Statements werden mit Aufruf von `close()` geschlossen. `Statement` erweitert `AutoCloseable`.

Abbildung 29-2 zeigt die Datenbankstruktur im *DB Browser for SQLite* nach der Ausführung des folgenden Programms.

Name	Typ	Schema
<input type="checkbox"/> <input checked="" type="checkbox"/> Tabellen (1)		
<input type="checkbox"/> <input checked="" type="checkbox"/> artikel		CREATE TABLE artikel ( id integer primary key, name text, preis real, menge integer)
<input type="checkbox"/> id	integer	"id" integer
<input type="checkbox"/> name	text	"name" text
<input type="checkbox"/> preis	real	"preis" real
<input type="checkbox"/> menge	integer	"menge" integer
<input type="checkbox"/> Indizes (0)		
<input type="checkbox"/> Ansichten (0)		
<input type="checkbox"/> Trigger (0)		

**Abbildung 29-2:** Die Tabelle *artikel*

```
package create;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class CreateDB {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        try (Connection con = DriverManager.getConnection(url);
             Statement stmt = con.createStatement()) {
```

```
String sql = """
    create table if not exists artikel (
        id integer primary key,
        name text,
        preis real,
        menge integer)
    """;
stmt.executeUpdate(sql);
} catch (SQLException e) {
    System.err.println(e.getMessage());
}
}
```

## 29.3 Daten einfügen

Es sollen nun einige Artikel in die Tabelle eingetragen werden. Hierzu dient die SQL-Anweisung:

```
insert into Tabelle (Spalten) values (Werte)
```

*Spalten* kann mehrere durch Kommas getrennte Feldnamen, *Werte* mehrere Werte für die entsprechenden Felder enthalten.

Beispiel:

```
insert into artikel (id, name, preis, menge)
    values (4711, 'Hammer', 3.9, 10)
```

Mit `executeUpdate` kann die `insert`-Anweisung ausgeführt werden.

```
package insert;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class Insert1 {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        try (Connection con = DriverManager.getConnection(url);
            Statement stmt = con.createStatement()) {
            String sql = """
                insert into artikel (id, name, preis, menge)
                    values (4711, 'Hammer', 3.9, 10)
                """;
            stmt.executeUpdate(sql);
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Es gibt eine im Allgemeinen einfachere Variante zu der obigen `insert`-Anweisung, die insbesondere dann Vorteile hat, wenn mehrere dieser Anweisungen hintereinander ausgeführt werden sollen.

Anstatt feste Werte einzufügen, werden Fragezeichen ? als Platzhalter verwendet.

Beispiel:

```
insert into artikel (id, name, preis, menge) values (?, ?, ?, ?)
```

Die Connection-Methode

```
PreparedStatement prepareStatement(String sql) throws SQLException
```

erzeugt ein Objekt vom Typ des Interface `java.sql.PreparedStatement`.

`PreparedStatement` erweitert `AutoCloseable`.

Mit dem Aufruf von geeigneten `set`-Methoden können dann die Platzhalter mit Werten belegt werden (siehe das folgende Programm).

Die `PreparedStatement`-Methode

```
int executeUpdate() throws SQLException
```

führt die Anweisung aus. Rückgabewert ist die Anzahl der neuen, geänderten oder gelöschten Zeilen.

```
package insert;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Insert2 {
    private static void neuerArtikel(
        PreparedStatement stmt, int id, String name, double preis, int menge)
        throws SQLException {
        stmt.setInt(1, id);
        stmt.setString(2, name);
        stmt.setDouble(3, preis);
        stmt.setInt(4, menge);
        stmt.executeUpdate();
    }

    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        String sql = """
            insert into artikel (id, name, preis, menge)
            values (?, ?, ?, ?)
            """;
        try (Connection con = DriverManager.getConnection(url);
             PreparedStatement stmt = con.prepareStatement(sql)) {
```

```

        neuerArtikel(stmt, 4712, "Zange", 2.9, 20);
        neuerArtikel(stmt, 4713, "Schraubendreher", 4., 15);
        neuerArtikel(stmt, 4714, "Akku-Bohrer", 25., 30);
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
}
}

```

	id	name	preis	menge
	Filtern	Filtern	Filtern	Filtern
1	4711	Hammer	3.9	10
2	4712	Zange	2.9	20
3	4713	Schraubendreher	4.0	15
4	4714	Akku-Bohrer	25.0	30

**Abbildung 29-3:** "Daten durchsuchen" im *DB Browser for SQLite*

## 29.4 Daten ändern und löschen

Die allgemeine Form der SQL-Anweisung zum Ändern von Daten ist:

```
update Tabelle set Spalte1 = Wert1, Spalte2 = Wert2 where ...
```

Es können mehrere Spaltenwerte geändert werden. Die `where`-Klausel ist optional. Fehlt sie, werden alle Zeilen der Tabelle geändert.

Beispiele:

Preisänderung für Artikel 4711:

```
update artikel set preis = 4. where id = 4711
```

Lagerzugang +5 für die Artikel 4712 und 4714:

```
update artikel set menge = menge + 5 where id = 4712 or id = 4714
```

Mit der `where`-Klausel werden Zeilen über eine Bedingung für die Änderung ausgewählt, hier sind es genau zwei Zeilen. Operatoren zur Verknüpfung einzelner Bedingungen sind `and` und `or`. Mit `not` wird eine Bedingung negiert.

`executeUpdate` führt die Änderung aus. Auch können hier Platzhalter wie oben verwendet werden.

```

package update;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class Update {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        try (Connection con = DriverManager.getConnection(url);
             Statement stmt = con.createStatement()) {
            String sql1 = "update artikel set preis = 4. where id = 4711";
            String sql2 = """
                            update artikel set menge = menge + 5
                            where id = 4712 or id = 4714
                            """;
            stmt.executeUpdate(sql1);
            stmt.executeUpdate(sql2);
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}

```

	<b>id</b>	<b>name</b>	<b>preis</b>	<b>menge</b>
	Filtern	Filtern	Filtern	Filtern
1	4711	Hammer	4.0	10
2	4712	Zange	2.9	25
3	4713	Schraubendreher	4.0	15
4	4714	Akku-Bohrer	25.0	35

**Abbildung 29-4:** Geänderte Artikel

Mit der `delete`-Anweisung können komplette Zeilen gelöscht werden:

```
delete from Tabelle where ...
```

Fehlt die `where`-Klausel, werden alle Zeilen der Tabelle gelöscht.

Auch hier können wieder Platzhalter bei Verwendung von `PreparedStatement` genutzt werden.

```

package delete;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

```

```

public class Delete {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        try (Connection con = DriverManager.getConnection(url)) {
            Statement stmt = con.createStatement();
            String sql = "delete from artikel where id = 4713";
            stmt.executeUpdate(sql);
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}

```

## 29.5 Daten abrufen

Die SQL-Anweisung

```
select Spalten from Tabelle where ... order by ...
```

liefert Zeilen der Tabelle mit den angegebenen, durch Kommas getrennten Feldnamen (Spalten). Die `where`-Klausel ist optional, ebenso `order by` mit den Feldnamen, nach denen sortiert werden soll. Die Sortierrichtung ist standardmäßig aufsteigend (`asc`), absteigend kann mit dem Zusatz `desc` (`descending`) sortiert werden.

Die Statement-Methode

```
ResultSet executeQuery(String sql) throws SQLException
```

führt die `select`-Anweisung `sql` aus. Das Ergebnis wird als `java.sql.ResultSet`-Objekt geliefert.

Einige Methoden des Interface `ResultSet`:

```
boolean next() throws SQLException
```

stellt den nächsten Datensatz zur Verfügung und liefert `true` oder `false`, wenn keine weiteren Datensätze vorliegen. Der erste Aufruf der Methode liefert den ersten Datensatz (falls vorhanden).

```
String getString(int n) throws SQLException
```

```
int getInt(int n) throws SQLException
```

```
double getDouble(int n) throws SQLException
```

liefern den Wert in der `n`-ten Spalte als `String`, `int`-Wert bzw. `double`-Wert. Die Spaltennummerierung beginnt bei 1.

```
void close()
```

gibt belegte Ressourcen frei. Ein `ResultSet`-Objekt wird automatisch geschlossen, wenn das zugehörige `Statement`-Objekt geschlossen wird.

`ResultSet` erweitert `AutoCloseable`.

Das folgende Programm erzeugt eine nach Preisen absteigend sortierte Liste der Artikel.

```
package select;

import java.sql.*;

public class Select {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        try (Connection con = DriverManager.getConnection(url);
            Statement stmt = con.createStatement()) {
            String sql =
                "select id, name, menge, preis from artikel order by preis desc";
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                System.out.printf("%4d %15s %4d %8.2f%n",
                    rs.getInt(1),
                    rs.getString(2),
                    rs.getInt(3),
                    rs.getDouble(4));
            }
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Ausgabe des Programms:

4714	Akku-Bohrer	35	25,00
4711	Hammer	10	4,00
4712	Zange	25	2,90

`PreparedStatement` besitzt auch die Methode `executeQuery`, sodass Platzhalter ? in der where-Klausel verwendet werden können (vgl. auch Programm `Insert2`).

## 29.6 Daten in einer Tabelle anzeigen

Artikeldaten sollen in einer JavaFX-Tabelle angezeigt werden.

Das Programm `create` erstellt die Datenbank `artikel.db` mit der Tabelle `artikel` wie in Kapitel 29.2, lädt Daten aus einer Datei und fügt sie in die Tabelle ein.

Die Datei `artikel.txt` hat den folgenden Aufbau:

```
1120#Signa-Color 120-A4 weiß#25.#100
1122#Signa-Color 80-A5 weiß#10.#50
1515#Signa-Color 70-A4 weiß#12.#200
...
```

Sie befindet sich im Begleitmaterial zu diesem Buch.

```
package query;

public class Artikel {
    private final int id;
    private final String name;
    private final double preis;
    private final int menge;

    public Artikel(int id, String name, double preis, int menge) {
        this.id = id;
        this.name = name;
        this.preis = preis;
        this.menge = menge;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getPreis() {
        return preis;
    }

    public int getMenge() {
        return menge;
    }
}
```

```
package query;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class Create {
    private static final String FILE = "artikel.txt";
    private static final String DB_URL = "jdbc:sqlite:artikel.db";

    private static void create(Connection con) throws SQLException {
        try (Statement stmt = con.createStatement()) {
            String sql = """
                create table artikel (
                    id integer primary key,
                    name text,
                    preis real,
                    menge integer)
                """;
            stmt.executeUpdate(sql);
        }
    }
}
```

```

        stmt.executeUpdate(sql);
    }
}

private static List<Artikel> loadData() throws IOException {
    Path in = Paths.get(FILE);
    List<String> lines = Files.readAllLines(in);
    List<Artikel> list = new ArrayList<>();
    for (String line : lines) {
        String[] fields = line.split("#");
        list.add(new Artikel(Integer.parseInt(fields[0]), fields[1],
                             Double.parseDouble(fields[2]), Integer.parseInt(fields[3])));
    }
    return list;
}

public static void main(String[] args) {
    try (Connection con = DriverManager.getConnection(DB_URL)) {
        create(con);

        String sql =
            "insert into artikel (id, name, preis, menge) values (?, ?, ?, ?)";
        PreparedStatement stmt = con.prepareStatement(sql);

        List<Artikel> list = loadData();
        for (Artikel a : list) {
            stmt.setInt(1, a.getId());
            stmt.setString(2, a.getName());
            stmt.setDouble(3, a.getPreis());
            stmt.setInt(4, a.getMenge());
            stmt.executeUpdate();
        }

        stmt.close();
    } catch (SQLException | IOException e) {
        System.err.println(e.getMessage());
    }
}
}

```

Im Programm ArtikelQuery werden die Artikel in Form einer Tabelle mit JavaFX angezeigt.

Um nach Artikeln suchen zu können, enthält die Oberfläche ein Suchfeld. Es muss nur ein Teil des Namens zum Suchen eingegeben werden. Wird das Eingabefeld leer gelassen, so werden alle Artikel angezeigt.

Die Artikelliste wird nach *Id*, *Name*, *Preis* oder *Menge* sortiert, wenn der Benutzer auf die entsprechende Spaltenüberschrift klickt.

Es wird die folgende, mit einem Platzhalter versehene SQL-Anweisung verwendet:

```
select id, name, preis, menge from artikel where name like ? order by id
```

Diese liefert alle Datensätze der Tabelle *artikel*, deren Name mit einem Muster übereinstimmt.

Ein Beispiel für den Datenwert, der für ? gesetzt wird:

%a4%

% steht für ein oder mehrere beliebige Zeichen. Es werden also alle Sätze gefunden, deren Name die Zeichenkette "a4" enthält, unabhängig von Klein- oder Großschreibung.

javafx.scene.control.Alert bietet vorgefertigte Dialogtypen, z. B.

AlertType.CONFIRMATION, AlertType.INFORMATION, AlertType.ERROR.

Die Methode showAndWait zeigt den Dialog an und wartet auf die Eingabe des Benutzers.

The screenshot shows a JavaFX application window titled "Artikel". At the top, there is a search bar containing the text "a4" and a "Suchen" button. Below the search bar is a table with four columns: "Id", "Name", "Preis", and "Menge". The table contains 14 rows of data. The data is as follows:

Id	Name	Preis	Menge
1120	Signa-Color 120-A4 weiß	25,00	100
1515	Signa-Color 70-A4 weiß	12,00	200
1517	Signa-Color 70-A4 hellgrün	13,00	100
1616	Signa-Color 80-A4 weiß	14,00	250
1825	Signa-Color 80-A4 hellgrün	15,00	100
2113	Eco-Color 80-A4 weiß	14,00	100
2920	Eco-Color 70-A4 weiß	22,00	100
3718	Laser-Color 80-A4 hellgrün	16,00	100
3721	Eco-Color 70-A4 hellgrün	15,00	100
4012	Laser-Color 70-A4 hellgrün	16,00	100
4158	Laser-Color 70-A4 weiß	15,00	100

14 Artikel

**Abbildung 29-5:** Nach Artikeln suchen

```
package query;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.AnchorPane;
```

```
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

import java.sql.*;

public class ArtikelQuery extends Application {
    private static final String DB_URL = "jdbc:sqlite:artikel.db";
    private TextField entry;
    private TableView<Artikel> tableView;
    private Label msg;
    private Connection con;
    private PreparedStatement ps;
    private String errorMessage;

    @Override
    public void init() {
        try {
            con = DriverManager.getConnection(DB_URL);
            String sql = """
                select id, name, preis, menge from artikel
                where name like ? order by id
                """;
            ps = con.prepareStatement(sql);
        } catch (Exception e) {
            errorMessage = e.getMessage();
        }
    }

    @Override
    public void start(Stage stage) {
        AnchorPane root = new AnchorPane();

        entry = new TextField();
        entry.setPrefWidth(400);

        Button search = new Button("Suchen");
        search.setOnAction(e -> search());

        tableView = new TableView<>();

        TableColumn<Artikel, Integer> id = new TableColumn<>("Id");
        TableColumn<Artikel, String> name = new TableColumn<>("Name");
        TableColumn<Artikel, Double> preis = new TableColumn<>("Preis");
        TableColumn<Artikel, Integer> menge = new TableColumn<>("Menge");

        id.setId("id");
        name.setId("name");
        preis.setId("preis");
        menge.setId("menge");

        tableView.getColumns().addAll(id, name, preis, menge);
        tableView.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY_FLEX_LAST_COLUMN);
        name.setMinWidth(200);

        id.setCellValueFactory(new PropertyValueFactory<>("id"));
        name.setCellValueFactory(new PropertyValueFactory<>("name"));
        preis.setCellValueFactory(new PropertyValueFactory<>("preis"));
        menge.setCellValueFactory(new PropertyValueFactory<>("menge"));
    }
}
```

```
preis.setCellFactory(column -> new TableCell<>() {
    protected void updateItem(Double item, boolean empty) {
        super.updateItem(item, empty);
        if (empty || item == null) {
            setText(null);
        } else {
            setText(String.format("%8.2f", item));
            setAlignment(Pos.CENTER_RIGHT);
        }
    }
});

menge.setCellFactory(column -> new TableCell<>() {
    protected void updateItem(Integer item, boolean empty) {
        super.updateItem(item, empty);
        if (empty || item == null) {
            setText(null);
        } else {
            setText(String.valueOf(item));
            setAlignment(Pos.CENTER_RIGHT);
        }
    }
});

msg = new Label();

HBox hbox = new HBox();
hbox.setSpacing(10);
hbox.getChildren().addAll(entry, search);

root.getChildren().addAll(hbox, tableView, msg);
root.setPrefWidth(600);
root.setPrefHeight(400);

AnchorPane.setTopAnchor(hbox, 12.);
AnchorPane.setLeftAnchor(hbox, 12.);

AnchorPane.setTopAnchor(tableView, 60.);
AnchorPane.setLeftAnchor(tableView, 12.);
AnchorPane.setRightAnchor(tableView, 12.);
AnchorPane.setBottomAnchor(tableView, 50.);

AnchorPane.setBottomAnchor(msg, 12.);
AnchorPane.setLeftAnchor(msg, 12.);

Scene scene = new Scene(root);
scene.getStylesheets().add(getClass()
    .getResource("ArtikelQuery.css").toExternalForm());

stage.setTitle("Artikel");
stage.setScene(scene);
stage.show();

if (errorMessage != null) {
    showDialog(errorMessage);
} else {
    search();
}
}
```

```
@Override
public void stop() {
    if (con != null) {
        try {
            ps.close();
            con.close();
        } catch (Exception e) {
            showDialog(e.getMessage());
        }
    }
}

private void search() {
    if (con == null)
        return;

    try {
        ObservableList<Artikel> list = FXCollections.observableArrayList();

        ps.setString(1, "%" + entry.getText() + "%");
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            list.add(new Artikel(rs.getInt(1), rs.getString(2),
                                 rs.getDouble(3), rs.getInt(4)));
        }
        rs.close();

        tableView.setItems(list);
        msg.setText(list.size() + " Artikel");
    } catch (Exception e) {
        showDialog(e.getMessage());
    }
}

private void showDialog(String text) {
    Alert alert = new Alert(AlertType.ERROR);
    alert.setTitle("Fehler");
    alert.setHeaderText("Exception");
    alert.setContentText(text);
    alert.showAndWait();
}

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

Die CSS-Datei *ArtikelQuery.css* ist wie in Kapitel 28.6 aufgebaut.

Sie können einen Fehler-Dialog provozieren und testen, indem Sie z. B. einen falschen Datenbanknamen im Quellcode eintragen.

## 29.7 Daten in einem Diagramm präsentieren

In einem Warenhaus werden die von Kunden getätigten Einkäufe in den verschiedenen Abteilungen gezählt und in der Tabelle *verkauf* mit den Spalten *id* und *abteilung* gespeichert.

Die folgende Abbildung zeigt einen Auszug aus der Tabelle.

id	▼	abteilung
Filtern	Filtern	
1		Elektrogeräte
2		Damenbekleidung
3		Elektrogeräte
4		Lebensmittel
5		Damenbekleidung
6		Lebensmittel
7		Herrenbekleidung

**Abbildung 29-6:** Die Tabelle *verkauf* (Auszug)

Die Tabelle *verkauf* wird mit der SQL-Anweisung

```
create table verkauf (
    id integer primary key,
    abteilung text
)
```

in der Datenbank *verkauf.db* erstellt.

Die Daten werden aus der Datei *verkauf.txt* in die Tabelle geladen.

Diese Datei hat den folgenden Aufbau:

```
Elektrogeräte
Damenbekleidung
Elektrogeräte
...
...
```

Sie befindet sich im Begleitmaterial zu diesem Buch.

```
package chart;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.sql.*;
import java.util.List;

public class Create {
    private static final String FILE = "verkauf.txt";
    private static final String DB_URL = "jdbc:sqlite:verkauf.db";

    private static void create(Connection con) throws SQLException {
        try (Statement stmt = con.createStatement()) {
            String sql = """
                create table verkauf (
                    id integer primary key,
                    abteilung text)
                """;
            stmt.executeUpdate(sql);
        }
    }

    private static List<String> loadData() throws IOException {
        Path in = Paths.get(FILE);
        return Files.readAllLines(in);
    }

    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection(DB_URL)) {
            create(con);

            String sql = "insert into verkauf (abteilung) values (?)";
            PreparedStatement stmt = con.prepareStatement(sql);

            List<String> list = loadData();
            con.setAutoCommit(false);
            for (String abt : list) {
                stmt.setString(1, abt);
                stmt.executeUpdate();
            }
            con.commit();
            stmt.close();
        } catch (SQLException | IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

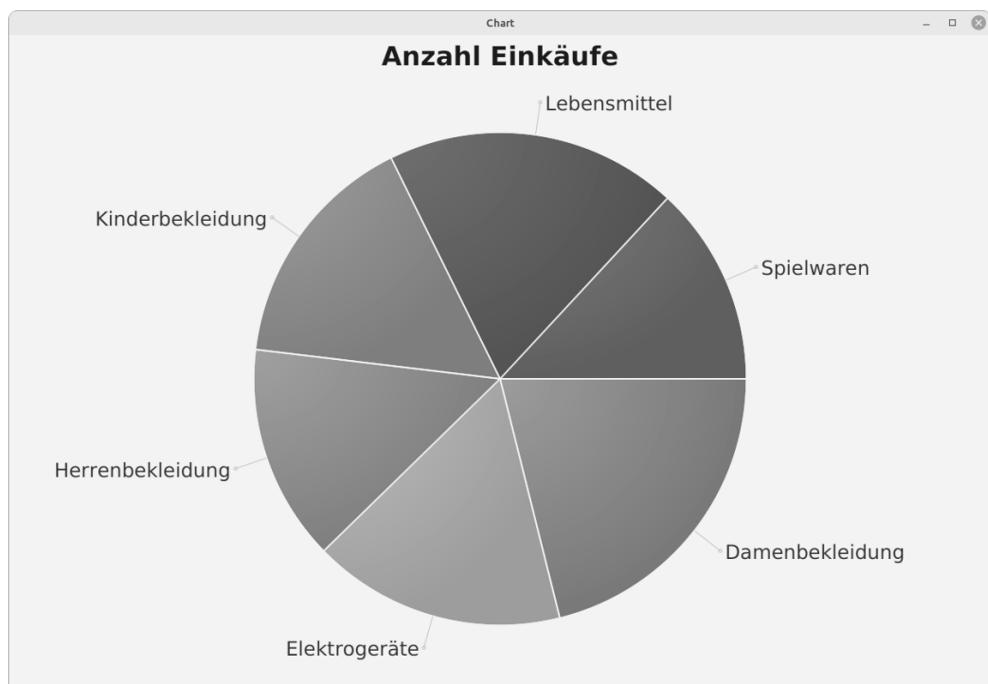
Die `insert`-Anweisung enthält keine Angabe zum Feld `id`. In diesem Fall setzt `SQLite` automatisch einen bisher noch nicht vergebenen Wert ein (*autoincrement*).

Eine *Datenbank-Transaktion* bildet eine logische Einheit, die einen Datenbestand nach fehlerfreier Ausführung in einem konsistenten Zustand hinterlässt.

Standardmäßig wird jede SQL-Anweisung als einzelne Transaktion ausgeführt. Mit `autoCommit(false)` kann dies aufgehoben werden. Alle nach diesem Aufruf bis zum Aufruf von `commit()` folgenden SQL-Anweisungen werden als eine einzige Transaktion ausgeführt.

Diese Maßnahme kann die Performance bei vielen `insert`-Anweisungen erheblich verbessern.

Ein Kreisdiagramm soll die Anzahl Einkäufe in den verschiedenen Abteilungen darstellen.



**Abbildung 29-7:** Aus der Datenbank aufbereitete Daten im Kreisdiagramm

Wir passen `Chart1` aus Kapitel 28.5 so an, dass nun die Daten direkt aus der Datenbank gelesen werden.

Hierzu müssen diese aber zuvor aggregiert werden. Es muss pro Abteilung die Anzahl der Einkäufe aufaddiert werden. Hierzu dient die folgende SQL-Anweisung:

```
select abteilung, count(*) from verkauf group by abteilung  
order by abteilung
```

Die Datensätze werden nach Abteilungen gruppiert.

Pro Abteilung wird die Anzahl mit der Funktion `count(*)` festgestellt und dann die Abteilung mit Anzahl als Ergebnis ausgegeben.

```
package chart;  
  
import javafx.application.Application;  
import javafx.collections.FXCollections;  
import javafx.collections.ObservableList;  
import javafx.scene.Node;  
import javafx.scene.Scene;  
import javafx.scene.chart.PieChart;  
import javafx.scene.control.Label;  
import javafx.scene.layout.AnchorPane;  
import javafx.stage.Stage;  
  
import java.sql.*;  
  
public class Chart extends Application {  
    private static final String DB_URL = "jdbc:sqlite:verkauf.db";  
    private ObservableList<PieChart.Data> dataList;  
  
    @Override  
    public void init() {  
        try (Connection con = DriverManager.getConnection(DB_URL);  
             Statement stmt = con.createStatement()) {  
            dataList = FXCollections.observableArrayList();  
            String sql = """  
                select abteilung, count(*) from verkauf  
                    group by abteilung order by abteilung  
            """;  
            ResultSet rs = stmt.executeQuery(sql);  
            while (rs.next()) {  
                System.out.println(rs.getString(1) + " " + rs.getInt(2));  
                dataList.add(new PieChart.Data(rs.getString(1), rs.getInt(2)));  
            }  
        } catch (SQLException e) {  
            System.err.println(e.getMessage());  
        }  
    }  
  
    @Override  
    public void start(Stage stage) {  
        AnchorPane pane = new AnchorPane();  
        PieChart chart = new PieChart();  
        chart.setData(dataList);  
        chart.setTitle("Anzahl Einkäufe");  
        chart.setPrefWidth(1200);  
        chart.setPrefHeight(800);  
    }  
}
```

```
Label label = new Label();
label.setLayoutX(50);
label.setLayoutY(50);
label.setId("value");

for (PieChart.Data data : dataList) {
    Node node = data.getNode();
    String text = String.format("%8.0f %8.2f %%", data.getPieValue(),
        data.getPieValue() * 100. / sum());
    node.setOnMouseEntered(e -> label.setText(text));
    node.setOnMouseExited(e -> label.setText(""));
}

pane.getChildren().addAll(chart, label);

Scene scene = new Scene(pane);
scene.getStylesheets().add(getClass()
    .getResource("chart.css").toExternalForm());
stage.setScene(scene);
stage.setTitle("Chart");
stage.show();
}

private double sum() {
    double sum = 0;
    for (PieChart.Data data : dataList) {
        sum += data.getPieValue();
    }
    return sum;
}

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

Die CSS-Datei *chart.css* ist wie in Kapitel 28.5 aufgebaut.

## 29.8 Aufgaben

### Aufgabe 1

Erstellen Sie mit Java und JDBC eine SQLite-Datenbank mit der Tabelle *kontakte*, die die Spalten *id*, *vorname*, *nachname* und *email* enthält. Überprüfen Sie das Ergebnis mit dem Programm *DB Browser for SQLite*.

Lösung: Paket *kontakte1*

### Aufgabe 2

Entwickeln Sie für die Datenbank aus Aufgabe 1 ein Programm, mit dem Sie Eingaben über Tastatur erfassen und in die Tabelle speichern können. Die *id* soll

von SQLite automatisch vergeben werden. Überprüfen Sie das Ergebnis mit dem Programm *DB Browser for SQLite*.

*Lösung:* Paket kontakte2

### Aufgabe 3

Erstellen Sie ein Programm, das eine komplette Liste mit den Kontakten der Datenbank aus Aufgabe 2 ausgibt.

*Lösung:* Paket kontakte3

### Aufgabe 4

Erstellen Sie ein Programm zum Löschen eines Kontakts aus der Datenbank in Aufgabe 3. Hierzu ist der Vor- und Nachname über Tastatur einzugeben. Beachten Sie, dass die SQL-Anweisung eine `where`-Klausel mit `and` enthalten muss. Hierdurch werden dann alle Sätze mit dem gleichen Vor- und Nachnamen gelöscht.

Überprüfen Sie das Ergebnis mit dem Programm *DB Browser for SQLite*.

*Lösung:* Paket kontakte4

### Aufgabe 5

Erstellen Sie ein Programm zur Abfrage der Datenbank aus Aufgabe 4. `select`-Anweisungen sollen über Tastatur in einer Schleife eingegeben werden können. Die Spaltennamen sollen vom Programm ermittelt werden.

Nutzen Sie hierzu `ResultSetMetaData`-Methoden.

Die `ResultSet`-Methode

```
ResultSetMetaData getMetaData() throws SQLException
```

liefert Informationen über die Ergebnismenge.

`ResultSetMetaData`-Methoden sind:

```
int getColumnCount() throws SQLException
```

liefert die Anzahl der Spalten.

```
String getColumnName(int col) throws SQLException
```

liefert den Namen der Spalte `col`.

Die Statement-Methode

```
Object getObject(int n) throws SQLException
```

liefert den Wert in der `n`-ten Spalte als Instanz vom Typ `Object`.

Beispiel:

```
> select vorname, nachname, email from kontakte order by nachname
```

```
#1
vorname      : Hugo
nachname     : Meier
email        : hugo.meier@web.de

#2
vorname      : Emil
nachname     : Nolde
email        : emil.nolde@gmx.de
...
...
```

*Lösung: Paket kontakte5*

### Aufgabe 6

Realisieren Sie eine Variante zu Aufgabe 1 in Kapitel 28. Die erfassten Datensätze sollen in der Tabelle *artikel* aus Kapitel 29.2 gespeichert werden. Überprüfen Sie das Ergebnis mit dem Programm *DB Browser for SQLite*.

*Lösung: Paket erfassung*

### Aufgabe 7

Eine große Anzahl von Zeilen mit den Spalten *id*, *name* und *description* soll in eine Tabelle eingefügt werden. Dazu sind zwei Programme zu entwickeln.

Das erste Programm nutzt *Statement* und fügt in einer Schleife 10.000.000 Sätze mit *insert* ein. Die Daten sollen mittels Laufindex generiert werden, also z. B.

```
1  Name #1  Decription #1
2  Name #2  Decription #2
```

usw.

Das zweite Programm nutzt hierzu *PreparedStatement* mit parametrisierter SQL-Anweisung.

Nutzen Sie in beiden Fällen eine Transaktion wie im Programm *Create* in Kapitel 29.7. Messen Sie in beiden Programmen mittels *System.currentTimeMillis()* die Laufzeit der Schleife und vergleichen Sie.

Das zweite Programm ist bedeutend schneller.

*Lösung: Paket performance*

### Aufgabe 8

#### *Ein digitales Tagebuch*

Das Programmfenster zeigt einen Kalender. Zu einem ausgewählten Tag kann ein Text erfasst werden. Texte des Tagebuchs sind normalerweise schreibgeschützt. Klickt man auf den Button "Ändern", kann ein Text erfasst bzw. geändert oder auch gelöscht werden. Klickt man auf den Button "HTML" kann ein Zeitraum (von - bis) ausgewählt werden. Für diesen werden alle Tagebucheinträge in einer HTML-Datei gespeichert und im Browser angezeigt.



Die Daten des Tagebuchs werden in einer SQLite-Datenbank im Projektverzeichnis mit dem Namen *diary.db*, die HTML-Seite in *diary.html* gespeichert.

Die Aufgabe besteht darin, den im Begleitmaterial vorhandenen Quellcode zu verstehen und auszuprobieren. Die verwendeten Eigenschaften und Methoden neuer, nicht im Kapitel 28 behandelter Klassen können in der API-Dokumentation zu JavaFX nachgeschlagen werden (vgl. Kapitel 28.1).

*Lösung: Paket diary*



## 30 Netzwerkkommunikation

Für die Kommunikation in Rechnernetzen auf der Basis des Protokolls *TCP/IP* (*Transmission Control Protocol/Internet Protocol*) stellt das Paket `java.net` die erforderlichen Klassen und Methoden zur Verfügung.

Bei *TCP/IP* handelt es sich um eine Gruppe von Protokollen, die die Grundlage für das Internet und andere Netzwerke (z. B. lokale Netzwerke, LAN = Local Area Network) bilden. Rechner, die an einem Netzwerk teilnehmen (auch Hosts genannt), werden über IP-Adressen identifiziert.

Beispiel: IPv4-Adresse 93.184.216.34

Rechnernamen, wie z. B. `www.example.com`, können über das weltweit verfügbare Domain Name System (DNS) in IP-Adressen übersetzt werden, ebenso umgekehrt.

### Lernziele

In diesem Kapitel lernen Sie

- wie Client und Server entwickelt werden können, die über das Protokoll TCP/IP im Netzwerk miteinander kommunizieren,
- wie das Web-Protokoll HTTP funktioniert,
- wie ein einfacher HTTP-Server und HTTP-Client entwickelt werden kann.

### 30.1 Dateien aus dem Netz laden

Mit einem *Uniform Resource Locator* (URL) können Ressourcen im Internet (Webseite, Text, Bild, Sound usw.) eindeutig adressiert werden. Diese Adresse hat die Form (hier vereinfacht dargestellt)

`protokoll://host[:port]/[path]`

- `protokoll` steht beispielsweise für `http`.
- `host` bezeichnet den Namen bzw. die IP-Nummer des Zielrechners.
- `port` (optional) steht für die Portnummer auf dem Zielrechner. Die Portnummer ist eine mehrstellige Zahl, mit der Dienste eines Rechners gekennzeichnet werden.
- `path` (optional) steht für den Pfad- und Dateinamen. Fehlt dieser oder ist nur der Name des Verzeichnisses angegeben, so wird in der Regel ein Standardname wie z. B. `index.html` als Dateiname unterstellt. `path` muss keine physische Datei bezeichnen. Der Name kann auch für eine Ressource stehen, die erst zur Laufzeit aus diversen Quellen (z. B. einer Datenbank) erzeugt wird.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_30](https://doi.org/10.1007/978-3-658-43574-5_30).

Beispiele:

```
http://www.firma.de/index.html  
https://de.wikipedia.org/wiki/Wiki
```

*https (HyperText Transfer Protocol Secure)* steht für ein Protokoll, das die Integrität und Vertraulichkeit des Datenverkehrs zwischen dem Rechner des Nutzers (Client) und dem Webserver schützt. HTTPS wird im Folgenden nicht behandelt.

## URL

Instanzen der Klasse `java.net.URL`, die Adressen der obigen Form repräsentieren, können aus einem String s wie folgt konstruiert werden:

```
URL url = URI.create(s).toURL();
```

Die URL-Methode

```
InputStream openStream() throws java.io.IOException
```

stellt eine Verbindung zur Ressource her und liefert einen `InputStream`, mit dem über diese Verbindung gelesen werden kann.

Mit dem Programm `Download` können Dateien auf den lokalen Rechner heruntergeladen werden.

```
package url;  
  
import java.io.BufferedReader;  
import java.io.BufferedInputStream;  
import java.io.BufferedOutputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.net.URI;  
import java.net.URL;  
  
public class Download {  
    public static void main(String[] args) {  
        String url = args[0];  
        String localFilename = args[1];  
  
        try {  
            URL _url = URI.create(url).toURL();  
            try (BufferedInputStream input = new BufferedInputStream(  
                _url.openStream());  
                BufferedOutputStream output = new BufferedOutputStream(  
                    new FileOutputStream(localFilename))) {  
  
                byte[] b = new byte[8192];  
                int c;  
                while ((c = input.read(b)) != -1) {  
                    output.write(b, 0, c);  
                }  
            }  
        }  
    }  
}
```

```
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Beispiele:

```
java -cp out/production/kap30 url.Download \
https://www.ietf.org/rfc/rfc1738.txt rfc1738.txt
```

Hier wird eine Textdatei über HTTPS heruntergeladen und lokal gespeichert.

```
java -cp out/production/kap30 url.Download \
file:///home/dietmar/tmp/test.pdf test.pdf
```

Die Datei `/home/dietmar/tmp/test.pdf` des lokalen Rechners wird über das Protokoll `file` gelesen. Hierzu ist kein Server nötig.

## 30.2 Eine einfache Client/Server-Anwendung

Eine *Client/Server-Anwendung* ist eine *geteilte Anwendung*, in der die Verarbeitung zu einem Teil vom *Client* und zum anderen Teil vom *Server* vorgenommen wird.

Das Client-Programm (z. B. ein Erfassungsformular) erstellt eine Anfrage (z. B. "Ermittle den Namen des Kunden mit der Kundennummer 4711") und schickt sie an das Server-Programm (z. B. ein Programm, das in einer Datenbank sucht).

Der Server nimmt diese Anfrage an, bearbeitet sie und schickt das Ergebnis als Antwort zurück.

Der Client kann sich auf demselben Rechner wie der Server oder auf einem anderen, über das Netz verbundenen Rechner befinden.

### Der Client

Im folgenden Beispiel wird ein *Client* implementiert, der eine Verbindung mit einem speziellen *Server* aufnehmen kann.

Direkt nach Aufnahme der Verbindung meldet sich der Server mit einer Textzeile zur Begrüßung. Eingaben für den Server werden über die Kommandozeile abgeschickt. Der Client schickt eine Textzeile und erwartet vom Server eine Textzeile als Antwort zurück.

### Socket

Sogenannte *Sockets* stellen die Endpunkte einer TCP/IP-Verbindung zwischen Client und Server dar. Sie stellen eine Schnittstelle für den Datenaustausch (Lesen und Schreiben) über das Netz zur Verfügung.

Die Klasse `java.net.Socket` implementiert das *clientseitige* Ende einer Netzwerkverbindung.

```
Socket(String host, int port) throws java.net.UnknownHostException,  
        java.io.IOException
```

erzeugt ein Socket-Objekt. `host` ist der Name bzw. die IP-Adresse des Rechners, auf dem der Server läuft. `port` ist die Nummer des Netzwerk-Ports auf dem Zielrechner. Ein Server bietet seinen Dienst immer über eine Portnummer (im Bereich von 0 bis 65535) an.

Einige Methoden der Klasse `Socket`:

```
InputStream getInputStream() throws IOException
```

liefert den Eingabedatenstrom für diesen Socket. Mit dem Schließen des Eingabedatenstroms wird auch der zugehörige Socket geschlossen.

```
OutputStream getOutputStream() throws IOException
```

liefert den Ausgabedatenstrom für diesen Socket. Mit dem Schließen des Ausgabedatenstroms wird auch der zugehörige Socket geschlossen.

```
void close() throws IOException
```

schließt diesen Socket. Mit dem Schließen des Sockets werden auch die zugehörigen Ein- und Ausgabedatenströme geschlossen.

`Socket` implementiert `AutoCloseable`.

```
package cs;  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.PrintWriter;  
import java.net.Socket;  
import java.util.Scanner;  
  
public class EchoClient {  
    public static void main(String[] args) {  
        String host = args[0];  
        int port = Integer.parseInt(args[1]);  
  
        try (Socket socket = new Socket(host, port);  
             BufferedReader in = new BufferedReader(new InputStreamReader(  
                     socket.getInputStream()));  
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);  
             Scanner sc = new Scanner(System.in)) {  
  
            System.out.println(in.readLine());  
  
            while (true) {  
                System.out.print("> ");  
                String line = sc.nextLine();  
                if (line.isEmpty())  
                    break;  
                out.println(line);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        System.out.println(in.readLine());
    }
} catch (IOException e) {
    System.err.println(e.getMessage());
}
}
```

## Der Server

Der *Server* schickt eine vom Client empfangene Textzeile als Echo ergänzt um einen Zeitstempel an den Client zurück.

### ServerSocket

Über ein Objekt der Klasse `java.net.ServerSocket` nimmt der Server die Verbindung zum Client auf.

```
ServerSocket(int port) throws IOException  
erzeugt ein ServerSocket-Objekt für den Port port.
```

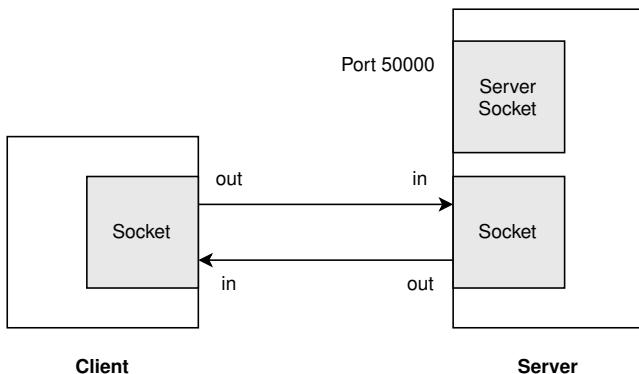
Die `ServerSocket`-Methode

```
Socket accept() throws IOException
```

wartet auf eine Verbindungsanforderung durch einen Client. Die Methode blockiert so lange, bis eine Verbindung hergestellt ist. Sie liefert dann ein `Socket`-Objekt, über das Daten vom Client empfangen bzw. an den Client gesendet werden können.

```
void close() throws IOException  
schließt den Server-Socket.
```

`ServerSocket` implementiert `AutoCloseable`.



**Abbildung 30-1:** Kommunikation über Sockets

Damit der Server mehrere Clients gleichzeitig bedienen kann, wird die eigentliche *Bearbeitung der Client-Anfrage in einem Thread* realisiert.

```
package cs;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.time.LocalDateTime;

public class EchoServer implements Runnable {
    private final Socket client;

    public EchoServer(Socket client) {
        this.client = client;
    }

    @Override
    public void run() {
        try (BufferedReader in = new BufferedReader(
                new InputStreamReader(client.getInputStream())));
            PrintWriter out = new PrintWriter(client.getOutputStream(), true)) {

            out.println("Hallo, ich bin der EchoServer.");

            String input;
            while ((input = in.readLine()) != null) {
                out.println("[" + LocalDateTime.now() + "] " + input);
            }
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}

public static void main(String[] args) {
    int port = Integer.parseInt(args[0]);

    try (ServerSocket server = new ServerSocket(port)) {
        System.out.println("EchoServer auf " + port + " gestartet ...");
        while (true) {
            Socket client = server.accept();
            new Thread(new EchoServer(client)).start();
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Zum Testen der beiden Programme muss eine freie Portnummer gewählt werden. Nummern kleiner als 1024 sind für Server des Systems reserviert und haben eine festgelegte Bedeutung.

Aufruf des Servers:

```
java -cp out/production/kap30 cs.EchoServer 50000
```

Der Server meldet:

```
EchoServer auf 50000 gestartet ...
```

Aufruf des Clients (in einem zweiten Terminal-Fenster):

```
java -cp out/production/kap30 cs.EchoClient localhost 50000
```

Der Name `localhost` bzw. die IP-Adresse `127.0.0.1` identifiziert den eigenen Rechner, auf dem auch der Server läuft.

Der Client gibt aus:

```
Hallo, ich bin der EchoServer.  
>
```

Eingabe beim Client:

```
Das ist ein Test.
```

Der Client meldet:

```
[2023-11-02T11:17:35.933344974] Das ist ein Test.  
>
```

Die Eingabetaste (Return) beendet den Client.

Strg + C beendet den Server.

### 30.3 HTTP

Webbrowser kommunizieren mit einem Webserver im Internet über das anwendungsbezogene Protokoll **HTTP** (HyperText Transfer Protocol).

Der Webbrowser ermittelt aus dem *URL*, z. B.

```
http://www.firma.de/produkte/index.html
```

den Webserver (`www.firma.de`) und stellt eine TCP-Verbindung zum Server über Port 80 her. 80 wird gewählt, falls die Portnummer nicht explizit angegeben ist.

Über diese Verbindung sendet der Webbrowser dann eine Anfrage, z. B. die Aufforderung, eine HTML-Seite zu übermitteln:

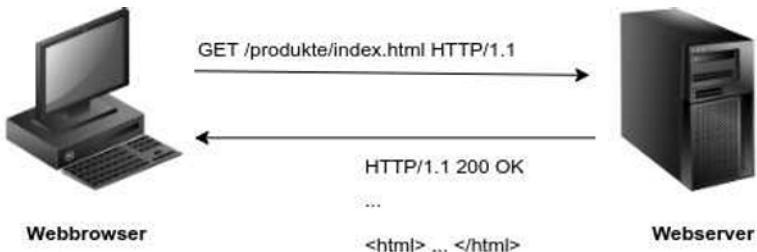
```
GET /produkte/index.html HTTP/1.1
```

Der Server beantwortet die Anfrage mit der Übertragung der verlangten HTML-Seite, die der Browser nun für die Darstellung am Bildschirm aufbereitet.

Viele HTML-Seiten sind *statisch*, d. h. sie sind in Dateien gespeichert, auf die der Webserver Zugriff hat. Diese Seiten enthalten für alle Benutzer dieselben Informationen.

Eine *dynamische* Seite wird erst dann erzeugt, wenn der Browser diese Seite anfordert. So können dann Informationen aus Datenquellen (z. B. einer Datenbank) erzeugt werden, die für jeden Client andere sind je nach der Client-spezifischen Anfrage.

Die Seitenbeschreibungssprache *HTML* bietet die Möglichkeit, Formulare zu erzeugen, mit denen z. B. interaktiv Daten aus einer Datenbank abgefragt werden können. Die Eingabedaten des Formulars werden vom Browser in spezieller Weise codiert und zum Server geschickt. Bei einer Datenbankanbindung greift der Server zum Zeitpunkt der Anfrage auf die Datenbank zu und generiert aus dem Abfrageergebnis einen in HTML codierten Ausgabestrom.



**Abbildung 30-2:** HTTP-Anfrage und -Antwort

### Formulardaten senden

## Testformular

Eingabe

**Abbildung 30-3:** Formular.html

In diesem Abschnitt wird ein spezieller HTTP-Server entwickelt, der den eingegeben Text empfängt und mit einem Zeitstempel versehen als Antwort an den Client zurücksendet.

Dabei muss der Server die Anfrage (*HTTP-Request*) interpretieren und die Antwort gemäß den HTTP-Konventionen aufbereiten (*HTTP-Response*).

Zunächst muss aber der hier relevante Teil des HTTP-Protokolls verstanden werden.

### Analyse des HTTP-Request

Das Programm `TestServer1` zeigt, was genau der Browser zum Server schickt, wenn Daten im Formular eingegeben und gesendet werden. Zum Testen wird die HTML-Datei `Formular.html` benutzt, die im Browser lokal geöffnet wird.

`Formular.html` hat den folgenden Inhalt:

```
<html lang="de">
<head>
    <meta charset="UTF-8">
    <title>Test</title>
</head>
<body>
<h1>Testformular</h1>
<form action="http://localhost:50000/test" method="GET">
    <label>
        <b>Eingabe</b>
        <input type="text" name="text" size="60">
    </label>
    <input type="submit" value="Senden">
</form>
</body>
</html>
```

Das Attribut `action` gibt den URL des Programms an, das die Eingabedaten verarbeiten soll. `method` bestimmt die Methode, mit der das Formular verarbeitet werden soll. Bei der `GET`-Methode werden die Eingabedaten direkt an den mit `action` spezifizierten URL angehängt. Das Formular enthält ein Textfeld (`input`) mit dem Namen `text` und einen Submit-Button, der URL und Eingabedaten zum Server sendet.

```
package http;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

public class TestServer1 {
    public static void main(String[] args) {
        int port = 50000;

        try (ServerSocket server = new ServerSocket(port)) {
            Socket client = server.accept();
            try (BufferedReader in = new BufferedReader(
                    new InputStreamReader(client.getInputStream()))) {
                String line;
                while ((line = in.readLine()) != null) {
                    System.out.println(line);
                }
            }
        }
    }
}
```

```

        if (line.isEmpty())
            break;
    }
}

} catch (IOException e) {
    System.err.println(e.getMessage());
}
}
}
}
```

Testablauf:

1. Aufruf des Servers
2. Öffnen des Formulars *Formular.html* im Browser (Strg + O)
3. Eingabe z. B.: Ein schöner Tag.
4. Senden

Eine evtl. Fehlermeldung des Browsers kann ignoriert werden.

Der Server gibt folgende Daten aus und ist dann beendet:

```

GET /test?text=Ein+sch%C3%BCner+Tag. HTTP/1.1
Host: localhost:50000
Accept: text/html,application/xhtml+xml,application/xml;q=0.9, ...
Accept-Encoding: gzip, deflate, br
Accept-Language: de,de-DE;q=0.9,en;q=0.8,en-US;q=0.7,1b;q=0.6
...

```

Alle Zeilen des HTTP-Request enden jeweils mit *Carriage Return* und *Linefeed*:

\r\n

Die erste Zeile enthält die HTTP-Methode (**GET**) und die Nutzdaten. Daten aus einem Formular mit mehreren Eingabefeldern werden allgemein nach dem Fragezeichen ? wie folgt codiert:

Name1=Wert1&Name2=Wert2&...

Alle Zeichen, die keine ASCII-Zeichen sind, und einige als Sonderzeichen verwendete Zeichen werden durch % gefolgt von ihrem Hexadezimalcode dargestellt. Leerzeichen werden als + codiert.

Diese Zeichenkette (*Query String*) muss nun vom Server interpretiert werden.

Das Programm **TestServer2** extrahiert den *Query String* aus der ersten Zeile des Request und decodiert ihn mit `decode` der Klasse `java.net.URLDecoder`:

```
static String decode(String str, java.nio.charset.Charset charset)
```

`charset` ist der zu verwendende Zeichensatz, hier `StandardCharsets.UTF_8`.

Nach der Decodierung des Query Strings stimmt er mit der Eingabe im Formular überein.

```
package http;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;

public class TestServer2 {
    public static void main(String[] args) {
        int port = 50000;

        try (ServerSocket server = new ServerSocket(port)) {
            Socket client = server.accept();
            try (BufferedReader in = new BufferedReader(
                    new InputStreamReader(client.getInputStream()))) {
                String line = in.readLine();
                if (line == null) {
                    return;
                }

                int x1 = line.indexOf('=');
                int x2 = line.indexOf(' ', x1);
                String query = line.substring(x1 + 1, x2);
                System.out.println(query);
                String decodedQuery = URLDecoder.decode(query,
                        StandardCharsets.UTF_8);
                System.out.println(decodedQuery);
            }
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Beispiel:

```
Ein+sch%C3%BCner+Tag.  
Ein schöner Tag.
```

### Ein spezieller HTTP-Server

Das folgende Programm implementiert das oben skizzierte Anwendungsbeispiel.

Nach Eingabe des URL

```
http://localhost:8080
```

im Browser schickt der Server zunächst das von ihm erzeugte "leere" Eingabe-Formular. Hier fehlt das Attribut `action` im Tag `<form>`. Der Browser setzt automatisch die Adresse des Servers ein, der das Formular geschickt hat.

Anstelle von `localhost` kann auch der Name (IP-Adresse) des Rechners stehen, falls er ans Netz (z. B. über WLAN) angeschlossen ist.

Die Antwort an den Browser (*HTTP-Response*) setzt sich aus dem *HTTP-Header* und dem *HTTP-Body* zusammen. Header und Body sind durch eine Leerzeile (`\r\n\r\n`) getrennt.

Der Header beginnt mit der HTTP-Versionsnummer und dem Statuscode (im Beispiel: `200 OK`) gefolgt von einer Information zum Typ der im Body enthaltenen Daten (im Beispiel: `Content-Type: text/html`).

Weitere Kopfzeilen (z. B. die Länge des HTML-Dokuments: `Content-Length`) können folgen.

Alle Kopfzeilen des Request enden jeweils mit *Carriage Return* und *Linefeed*:

`\r\n`

Die vom Server generierten HTML-Seiten sind in UTF-8 codiert.

```
package http;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.time.LocalDateTime;

public class EchoServer implements Runnable {
    private final Socket client;

    public EchoServer(Socket client) {
        this.client = client;
    }

    @Override
    public void run() {
        try (BufferedReader in = new BufferedReader(
                new InputStreamReader(client.getInputStream()));
             PrintWriter out = new PrintWriter(client.getOutputStream(), true)) {
            String text = readRequest(in);
            writeResponse(out, text);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }

    private String readRequest(BufferedReader in) throws IOException {
        String line = in.readLine();
        if (line == null)
            throw new IOException("Request ist null");
        int x1 = line.indexOf('=');
        if (x1 < 0)
```

```
        return "";
    int x2 = line.indexOf(' ', x1);
    String text = line.substring(x1 + 1, x2);
    return URLDecoder.decode(text, StandardCharsets.UTF_8).trim();
}

private void writeResponse(PrintWriter out, String text) {
    StringBuilder sb = new StringBuilder();
    sb.append("HTTP/1.1 200 OK\r\n")
        .append("Content-Type: text/html\r\n\r\n")
        .append("")
        .append("<html lang=\"de\">\n<head>\n    <meta charset=\"UTF-8\">\n    <title>Test</title>\n</head>\n<body>\n    <h1>Testformular</h1>\n    <form method=\"GET\">\n        <label>\n            <b>Eingabe</b>\n            <input type=\"text\" name=\"text\" size=\"60\">\n        </label>\n        <input type=\"submit\" value=\"Senden\">\n    </form>\n    \"");\n\n    if (!text.isEmpty()) {\n        sb.append("[")\n            .append(LocalDateTime.now())\n            .append("]<br>")\n            .append(text);\n    }\n\n    sb.append("</body>").append("</html>");\n\n    out.println(sb);
}

public static void main(String[] args) {
    int port = 8080;

    try (ServerSocket server = new ServerSocket(port)) {
        System.out.println("EchoServer auf " + port + " gestartet ...");
        while (true) {
            Socket client = server.accept();
            new Thread(new EchoServer(client)).start();
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

## 30.4 Simple Web Server

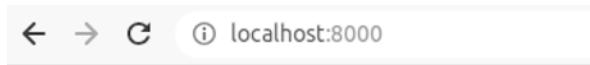
Ab Java 18 gehört ein einfacher Webserver zum JDK, mit dem statische Seiten bereitgestellt werden können. Er eignet sich besonders für die Erstellung von Prototypen und kann für Tests genutzt werden.

Mit dem Kommandozeilen-Tool `jwebserver` kann der *Simple Web Server* gestartet werden.

Beispiel (Start im Projekt-Verzeichnis `kap30`):

```
jwebserver
```

```
Binding an Loopback als Standard. Verwenden Sie für alle Schnittstellen  
"-b 0.0.0.0" oder "-b ::".  
Bedient /home/dietmar/Projekte/OPEN/java-buch/GKJava12/Kap30-Net/kap30 und  
Unterverzeichnisse auf 127.0.0.1 Port 8000  
URL http://127.0.0.1:8000/
```



### Verzeichnisliste für /

- [lib/](#)
- [src/](#)
- [out/](#)
- [Formular.html](#)
- [kap30.iml](#)

**Abbildung 30-4:** Anzeige des Verzeichnisinhalts

Beispiel (Aufruf mit Optionen):

```
jwebserver -b 192.168.2.90 -p 8080 -d /home/dietmar -o verbose
```

Optionen:

- b IP-Adresse
- p Portnummer
- d Root-Verzeichnis
- o Ausgabeformat (Konsole)

Der Webserver ist hier von Rechnern im Netz über `http://192.168.2.90:8080` erreichbar.

Der *Simple Web Server* hat folgende Einschränkungen:

- Das einzig unterstützte Protokoll ist HTTP/1.1.
- HTTPS wird nicht unterstützt.
- Erlaubt sind nur die HTTP-Methoden GET und HEAD.

Mit der HEAD-Methode sendet der Server die gleichen HTTP-Header wie bei GET, allerdings ohne die eigentlichen Nutzdaten. HEAD eignet sich insbesondere dazu, die Größe von Ressourcen, Inhaltstyp oder andere Meta-Daten zu analysieren.

## 30.5 Das HTTP-Client-API

Das mit Java-Version 11 eingeführte *HTTP-Client-API* unterstützt die Weiterentwicklung des HTTP-Protokolls (HTTP/2).<sup>1</sup>

Zur Illustration entwickeln wir die Client/Server-Anwendung aus Kapitel 30.2 auf dieser Basis. Hierzu passen wir den HTTP-Server aus Kapitel 30.3 so an, dass der empfange Text wiederum als einfacher Text mit Zeitstempel zurückgeschickt wird.

Im Programm EchoServer2 ist gegenüber EchoServer nur die folgende Methode geändert:

```
private void writeResponse(PrintWriter out, String text) {  
    out.print("HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n" +  
        "[" + LocalDateTime.now() + "] " + text);  
}
```

### Der HTTP-Client

Eine Instanz vom Typ `java.net.http.HttpClient` wird mit der statischen Methode `newHttpClient` erzeugt.

Die HTTP-Anfrage vom Typ `java.net.http.HttpRequest` wird mit

```
HttpRequest request = HttpRequest.newBuilder()  
    .uri(URI.create(uri))  
    .GET()  
    .build();
```

aufgebaut.

Im obigen Code-Ausschnitt wird ein *Uniform Resource Identifier (URI)* verwendet. Hierbei handelt es sich um eine Zeichenfolge, die eine abstrakte oder physische Ressource identifiziert.

---

<sup>1</sup> [https://de.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol#HTTP/2](https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol#HTTP/2)

Der Begriff *Uniform Resource Locator (URL)* aus Kapitel 30.1 bezieht sich nur auf eine Teilmenge der URIs. Ein URL gibt neben der eigentlichen Identifizierung auch ein Mittel zur Lokalisierung der Ressource an (physischer Standort).

Die Kommunikation wird durch Aufruf von `send` gestartet.

Für die Verarbeitung der HTTP-Antwort ist eine Instanz vom Typ `java.net.http.HttpResponse.BodyHandlers` zuständig, die hier mit `ofString` erzeugt wird. Die `java.net.http.HttpResponse`-Methode `body` liefert also einen String.

```
package http;

import java.io.IOException;
import java.net.URI;
import java.net.URLEncoder;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

public class EchoClient {
    public static void main(String[] args) {
        String url = "http://localhost:8080";

        try (HttpClient client = HttpClient.newHttpClient();
            Scanner sc = new Scanner(System.in)) {
            while (true) {
                System.out.print("> ");
                String line = sc.nextLine();
                if (line.isEmpty())
                    break;

                String uri = url + "?text=" + URLEncoder
                    .encode(line, StandardCharsets.UTF_8);
                HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(uri))
                    .GET()
                    .build();

                HttpResponse<String> response = client.send(
                    request, HttpResponse.BodyHandlers.ofString());
                System.out.println(response.body());
            }
        } catch (IOException | InterruptedException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

### Wetterdaten abrufen

*OpenWeatherMap* ist ein Online-Dienst, der eine Programmierschnittstelle (API) für Wetterdaten anbietet.<sup>2</sup>

Um das folgende Beispielprogramm auszuführen, kann man sich kostenlos registrieren und einen API-Key generieren.

Beispiel zum Abruf der Wetterdaten im Browser:

```
https://api.openweathermap.org/data/2.5/weather?lat=51.2964148&lon=6.8401844&units=metric&appid=<Hier steht der API-Key>
```

lat und long geben den Breiten- und Längengrad des Ortes an.

Ausgabe im JSON-Format (Beispiel):

```
{  
    ...  
    "main": {  
        "temp": 33.43,  
        "feels_like": 32.87,  
        "temp_min": 32.06,  
        "temp_max": 34.75,  
        "pressure": 1016,  
        "humidity": 32  
    },  
    ...  
    "name": "Ratingen",  
    ...  
}
```

Hier wurden die nicht benötigten Informationen weggelassen.

Zur Auswertung der Daten benutzen wir die Bibliothek *Gson* aus Kapitel 25.

```
package wetter;  
  
import com.google.gson.Gson;  
import com.google.gson.annotations.SerializedName;  
  
import java.io.IOException;  
import java.net.URI;  
import java.net.http.HttpClient;  
import java.net.http.HttpRequest;  
import java.net.http.HttpResponse;  
  
class Wetter {  
    String name;  
    @SerializedName("main")  
    Detail detail;  
}
```

---

2 <https://openweathermap.org>

```

class Detail {
    double temp;
    double feels_like;
    double temp_min;
    double temp_max;
    double pressure;
    double humidity;
}

public class Main {
    public static void main(String[] args) {
        try (HttpClient client = HttpClient.newHttpClient()) {
            String url = """
                https://api.openweathermap.org/data/2.5/weather?\n
                lat=51.2964148&lon=6.8401844&units=metric\n
                &appid=<Hier steht der API-Key>""";
            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(url))
                .GET()
                .build();
            HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());
            String jsonString = response.body();

            Gson gson = new Gson();
            Wetter wetter = gson.fromJson(jsonString, Wetter.class);
            System.out.println("name: " + wetter.name);
            System.out.println("temp: " + wetter.detail.temp);
            System.out.println("feels_like: " + wetter.detail.feels_like);
            System.out.println("temp_min: " + wetter.detail.temp_min);
            System.out.println("temp_max: " + wetter.detail.temp_max);
            System.out.println("pressure: " + wetter.detail.pressure);
            System.out.println("humidity: " + wetter.detail.humidity);
        } catch (IOException | InterruptedException e) {
            System.err.println(e.getMessage());
        }
    }
}

```

Die Klassen `Wetter` und `Detail` repräsentieren die gewünschten Informationen aus der JSON-Ausgabe. Da `main` als Attributname nicht erlaubt ist, erfolgt ein Mapping des Schlüssels "main" aus der JSON-Ausgabe mit der Annotation `@SerializedName` auf einen in Java gültigen Namen.

Ausgabe des Programms:

```

name: Ratingen
temp: 33.43
feels_like: 32.87
temp_min: 32.06
temp_max: 34.75
pressure: 1016.0
humidity: 32.0

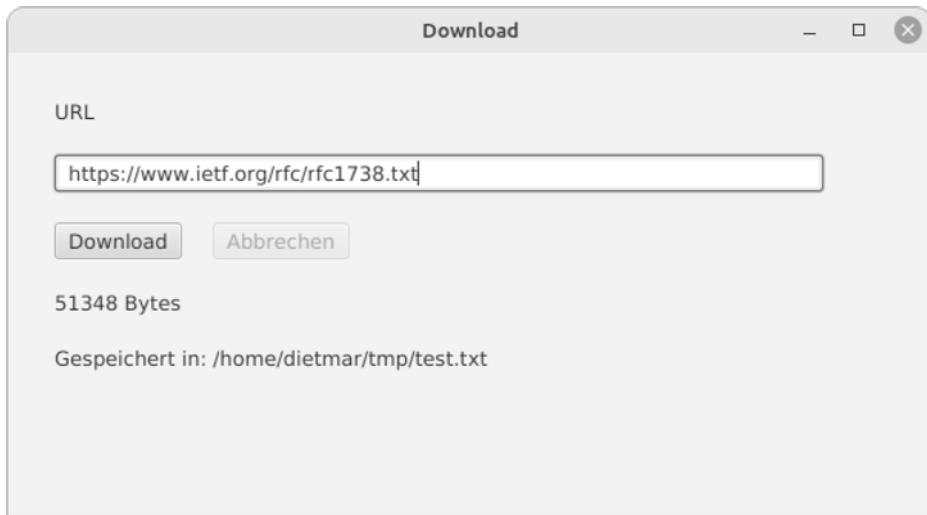
```

## 30.6 Aufgaben

### Aufgabe 1

Entwickeln Sie mit JavaFX eine grafische Oberfläche zum Download-Programm in Kapitel 30.1. Der lokale Speicherort soll über einen Dateiauswahldialog bestimmt werden.

Hinweis: Verwenden Sie die `FileChooser`-Methode `showSaveDialog` (vgl. auch Aufgabe 3 in Kapitel 28). Die Dateiübertragung soll in einer Task (siehe Kapitel 28.4) erfolgen.



*Lösung: Paket download1*

### Aufgabe 2

Entwickeln Sie einen Server, der nach Aufnahme einer Verbindung die aktuelle Systemzeit an den Client sendet. Implementieren Sie auch den passenden Client dazu.

*Lösung: Paket time*

### Aufgabe 3

Entwickeln Sie eine Variante zum EchoServer in Kapitel 30.3, bei der statt der GET- die POST-Methode im Formular verwendet wird. Der gesendete Text wird dann nicht an den URL angehängt und ist demnach auch nicht im Adressfeld des Browsers zu sehen.

Der HTTP-Request hat den folgenden Aufbau:

```
POST ... HTTP/1.1
...
Content-Length: nnn
...
<Hier steht eine Leerzeile>
Name1=Wert1&Name2=Wert2&...
```

Die Zahl `nnn` gibt die Länge der nach der Leerzeile folgenden Daten in Bytes an.

*Lösung: Paket post1*

#### Aufgabe 4

Entwickeln Sie analog zu den Programmen in Kapitel 30.5 einen HTTP-Client und einen HTTP-Server unter Verwendung der Methode `POST` (vgl. Aufgabe 3). Beim Client ist statt `GET()` der Aufruf von `POST(...)` zu benutzen:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://localhost:8080"))
    .POST(HttpRequest.BodyPublishers.ofString(text))
    .build();
```

*Lösung: Paket post2*

#### Aufgabe 5

Entwickeln Sie eine Variante zum Download-Programm in Kapitel 30.1, indem Sie das `HTTP-Client-API` verwenden.

Nutzen Sie in der Methode `send`:

```
HttpResponse.BodyHandlers.ofFile(Paths.get(localFilename))
```

*Lösung: Paket download2*

#### Aufgabe 6

Stellen Sie mit dem Simple Web Server `jwebserver` einen Server zur Verfügung, der PDF-Dokumente in einem Verzeichnis über HTTP für Rechner im lokalen Netz zur Verfügung stellt.

*Lösung: Paket fileservlet*



## 31 Fallbeispiel

In diesem Kapitel wird ein etwas umfangreicheres Programm vorgestellt. Es zeigt, dass eine gut durchdachte Architektur für das zu entwickelnde Programm eine wesentliche Grundlage der Entwicklung ist.

Zur Implementierung werden Klassen und Methoden der vorhergehenden Kapitel zu den Themen JavaFX, JDBC und Netzwerkkommunikation benutzt.

### Lernziele

In diesem Kapitel lernen Sie

- wie ein Programm in Schichten (GUI, Anwendungslogik und Datenhaltung) strukturiert werden kann,
- wie Klassen und Ressourcen zur Realisierung der grafischen Oberfläche, zur Umsetzung der fachlichen Logik und zur Verwaltung der Daten in einer Datenbank diesen Schichten zugeordnet werden können,
- wie aus einer lokalen Desktop-Anwendung eine Client/Server-Anwendung entwickelt werden kann.

### 31.1 Die Anwendung

Für dieses Praxisprojekt erstellen wir ein Programm zur Verwaltung des Sortiments eines Händlers. Es können neue Produkte zum Sortiment hinzugefügt, Produkte geändert und gelöscht und natürlich das Sortiment angezeigt werden. Ebenso ist ein Warenzugang bzw. -abgang möglich.

Die Produkte haben die folgenden Attribute:

- eindeutige Nummer,
- Bezeichnung,
- Preis und
- Menge.

Die Produkte werden in der Tabelle `product` einer *SQLite*-Datenbank gespeichert. Die Tabelle hat die Felder:

- `id` integer primary key
- `name` text
- `price` real
- `quantity` integer

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_31](https://doi.org/10.1007/978-3-658-43574-5_31).

Abbildung 31-1 zeigt das Hauptfenster nach Start des Programms. Die angezeigten Daten wurden speziell für diese Darstellung automatisch generiert.

Sortiment			
Nach Namen suchen		Suchen	Neu
<b>Id</b>	<b>Name</b>	<b>Preis</b>	<b>Menge</b>
1010	dapibus nulla	50,82	25
1020	ante ipsum	57,71	59
1030	malesuada	77,09	6
1040	blandit lacinia	23,46	65
1050	nulla sed	81,58	59
1060	ut suscipit	49,63	35
1070	rutrum rutrum	30,16	61
1080	ac tellus	21,95	74
1090	diam cras	14,60	9
1100	porta	18,92	69
1110	sed augue	15,10	42
1120	dolor	12,51	7
1130	feugiat	80,63	35
1140	est	45,72	79
1150	sit amet	50,39	17

Anzahl: 100

**Abbildung 31-1:** Das Hauptfenster

Man kann nach Produktnamen suchen. Dafür muss nur ein Teil des Namens eingegeben werden. Ein Produkt kann neu erfasst werden. Ein Produkt kann geändert oder gelöscht werden; hierzu muss die entsprechende Tabellenzeile ausgewählt werden. Abbildung 31-2 zeigt das Formular zur Datenerfassung.

Wir entwickeln die Anwendung in zwei Varianten.

#### *Variante 1*

Desktop-Anwendung mit Datenbank

#### *Variante 2*

Client/Server-Anwendung

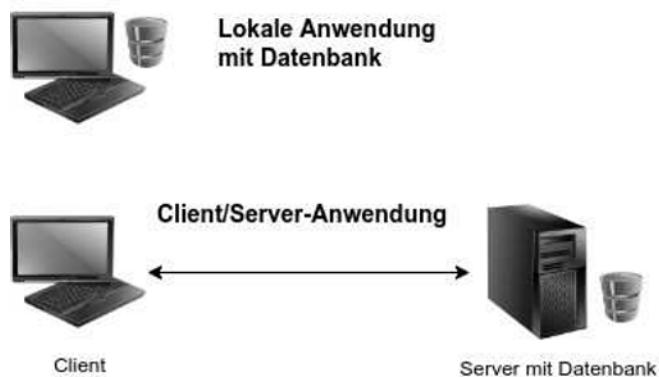
Der Client enthält die Oberfläche und die Anwendungs- bzw. Steuerungslogik. Der Server verwaltet die Daten in der Datenbank (Abfragen und Änderungen).

Produkt

<b>Id</b>	2010		
<b>Name</b>	Akku-Bohrer AB 2000		
<b>Preis</b>	35.99		
<b>Menge</b>	30	+/-	0

**Speichern** **Zurück**

**Abbildung 31-2:** Formular zur Datenerfassung



**Abbildung 31-3:** Zwei Varianten

## 31.2 Variante 1: Lokale Anwendung

Abbildung 31-4 zeigt eine Übersicht über die Klassen und ihre Beziehungen.

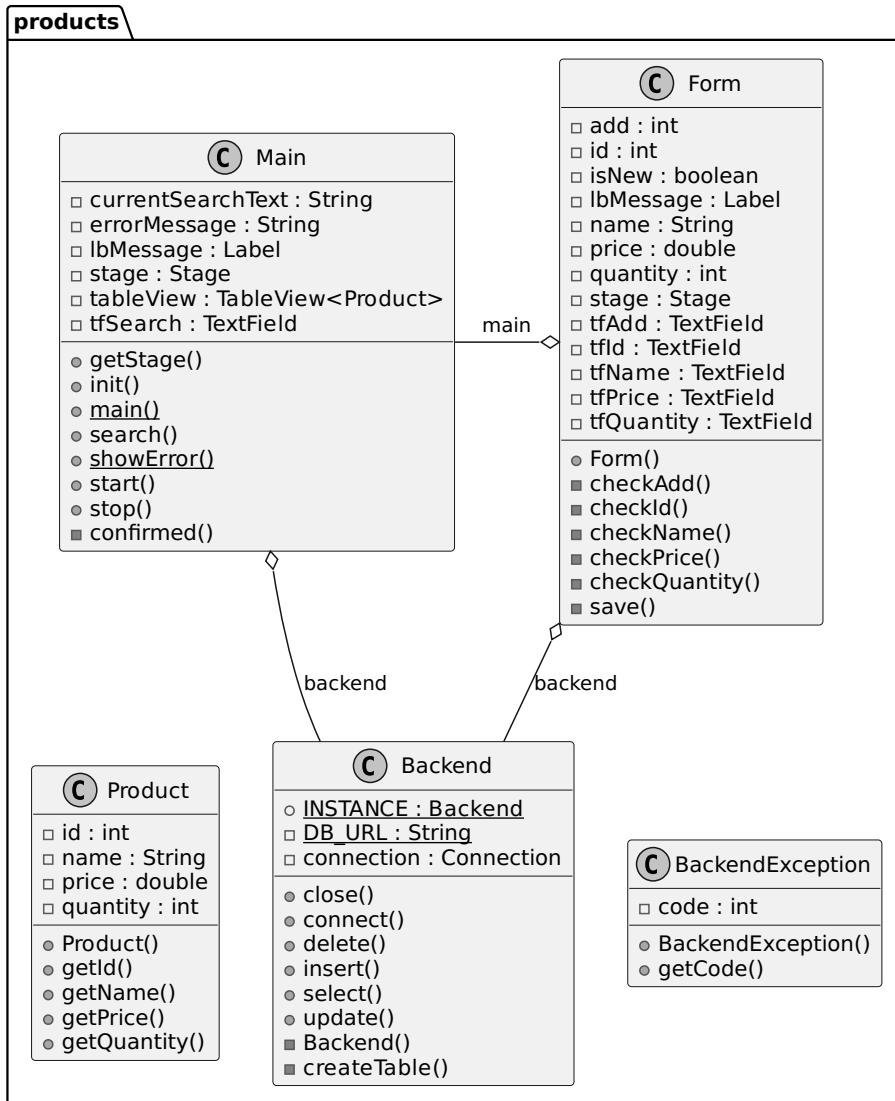


Abbildung 31-4: Klassendiagramm

## Backend

Wir beginnen mit der Datenhaltung. Alle Datenbankzugriffsmethoden sind in der Klasse Backend implementiert. Fehler werden mittels BackendException weitergeleitet. Die Klasse Backend ist als *Singleton* realisiert (siehe Kapitel 12.5), sodass eine Datenbankverbindung nur einmal zur Laufzeit erstellt wird. Bei Aufnahme der Verbindung wird die Tabelle product erzeugt, sofern sie noch nicht existiert. Für jeden SQL-Anweisungstyp (`select`, `insert`, `update`, `delete`) gibt es eine separate Methode.

```
package products;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class Backend {
    public static final Backend INSTANCE = new Backend();

    private static final String DB_URL = "jdbc:sqlite:products.db";
    private Connection connection;

    private Backend() {
    }

    public void connect() throws BackendException {
        if (connection == null) {
            try {
                connection = DriverManager.getConnection(DB_URL);
                createTable();
            } catch (SQLException e) {
                throw new BackendException(e.getMessage(), e.getErrorCode());
            }
        }
    }

    public void close() {
        try {
            if (connection != null) {
                connection.close();
                connection = null;
            }
        } catch (SQLException ignored) {
        }
    }

    public List<Product> select(String text) throws BackendException {
        List<Product> list = new ArrayList<>();
        String sql = """
                    select id, name, price, quantity from product
                    where name like ? order by id
                    """;
        try (PreparedStatement ps = connection.prepareStatement(sql)) {
            ps.setString(1, "%" + text + "%");
            ResultSet rs = ps.executeQuery();
```

```
        while (rs.next()) {
            list.add(new Product(rs.getInt(1), rs.getString(2),
                rs.getDouble(3), rs.getInt(4)));
        }
        return list;
    } catch (SQLException e) {
        throw new BackendException(e.getMessage(), e.getErrorCode());
    }
}

public void insert(Product product) throws BackendException {
    String sql =
        "insert into product (id, name, price, quantity) values(?, ?, ?, ?)";
    try (PreparedStatement ps = connection.prepareStatement(sql)) {
        ps.setInt(1, product.getId());
        ps.setString(2, product.getName());
        ps.setDouble(3, product.getPrice());
        ps.setInt(4, product.getQuantity());
        ps.executeUpdate();
    } catch (SQLException e) {
        throw new BackendException(e.getMessage(), e.getErrorCode());
    }
}

public void update(Product product) throws BackendException {
    String sql =
        "update product set name = ?, price = ?, quantity = ? where id = ?";
    try (PreparedStatement ps = connection.prepareStatement(sql)) {
        ps.setString(1, product.getName());
        ps.setDouble(2, product.getPrice());
        ps.setInt(3, product.getQuantity());
        ps.setInt(4, product.getId());
        ps.executeUpdate();
    } catch (SQLException e) {
        throw new BackendException(e.getMessage(), e.getErrorCode());
    }
}

public void delete(int id) throws BackendException {
    try (Statement stmt = connection.createStatement()) {
        String sql = "delete from product where id = " + id;
        stmt.executeUpdate(sql);
    } catch (SQLException e) {
        throw new BackendException(e.getMessage(), e.getErrorCode());
    }
}

private void createTable() throws SQLException {
    try (Statement stmt = connection.createStatement()) {
        String sql = """
            create table if not exists product (
                id integer primary key,
                name text,
                price real,
                quantity integer)
            """;
        stmt.executeUpdate(sql);
    }
}
```

```
        }
    }
}

package products;

public class BackendException extends Exception {
    private final int code;

    public BackendException(String message, int code) {
        super(message);
        this.code = code;
    }

    public int getCode() {
        return code;
    }
}
```

Die Klasse `Product`<sup>1</sup> stellt das fachliche Datenmodell dar.

```
package products;

public class Product {
    private final int id;
    private final String name;
    private final double price;
    private final int quantity;

    public Product(int id, String name, double price, int quantity) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    public int getQuantity() {
        return quantity;
    }
}
```

---

<sup>1</sup> Die Verwendung eines *Records* ist nicht möglich, da *JavaFX* und *Gson* get-Methoden benötigen.

## GUI und Anwendungslogik

Ein Großteil der Klasse `Main` widmet sich dem Aufbau des Hauptfensters. Die CSS-Datei `Main.css` sorgt für die Ausrichtung der Spaltenüberschriften der Tabelle.

Wir zeigen die komplette Klasse und erläutern im Anschluss die wesentlichen Aspekte.

```
package products;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.Event;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

import java.util.List;

public class Main extends Application {
    private Stage stage;
    private TextField tfSearch;
    private TableView<Product> tableView;
    private Label lbMessage;

    private Backend backend;
    private String errorMessage;
    private String currentSearchText = "";

    @Override
    public void init() {
        try {
            backend = Backend.INSTANCE;
            backend.connect();
        } catch (BackendException e) {
            errorMessage = e.getMessage();
        }
    }

    @Override
    public void start(Stage stage) {
        if (errorMessage != null) {
            showError(errorMessage);
            return;
        }

        this.stage = stage;
        AnchorPane root = new AnchorPane();

        tfSearch = new TextField();
        tfSearch.setPromptText("Nach Namen suchen");
    }
}
```

```
tfSearch.setPrefWidth(300);

Button btnSearch = new Button("Suchen");
Platform.runLater(btnSearch::requestFocus);
Button btnNew = new Button("Neu");
Button btnUpdate = new Button("Ändern");
Button btnDelete = new Button("Löschen");
Button btnExit = new Button("Beenden");

tableView = new TableView<>();

TableColumn<Product, Integer> id = new TableColumn<>("Id");
TableColumn<Product, String> name = new TableColumn<>("Name");
TableColumn<Product, Double> price = new TableColumn<>("Preis");
TableColumn<Product, Integer> quantity = new TableColumn<>("Menge");

id.setId("id");
name.setId("name");
price.setId("price");
quantity.setId("quantity");

tableView.getColumns().addAll(id, name, price, quantity);
tableView.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY_FLEX_LAST_COLUMN);
name.setMinWidth(200);

id.setCellValueFactory(new PropertyValueFactory<>("id"));
name.setCellValueFactory(new PropertyValueFactory<>("name"));
price.setCellValueFactory(new PropertyValueFactory<>("price"));
quantity.setCellValueFactory(new PropertyValueFactory<>("quantity"));

price.setCellFactory(column -> new TableCell<>() {
    protected void updateItem(Double item, boolean empty) {
        super.updateItem(item, empty);

        if (empty || item == null) {
            setText(null);
        } else {
            setText(String.format("%8.2f", item));
            setAlignment(Pos.CENTER_RIGHT);
        }
    }
});

quantity.setCellFactory(column -> new TableCell<>() {
    protected void updateItem(Integer item, boolean empty) {
        super.updateItem(item, empty);

        if (empty || item == null) {
            setText(null);
        } else {
            setText(String.valueOf(item));
            setAlignment(Pos.CENTER_RIGHT);
        }
    }
});

lbMessage = new Label();
```

```
HBox box1 = new HBox();
box1.setSpacing(10);
box1.getChildren().addAll(tfSearch, btnSearch);

HBox box2 = new HBox();
box2.setSpacing(10);
box2.getChildren().addAll(btnNew, btnUpdate, btnDelete, btnExit);

root.getChildren().addAll(box1, box2, tableView, lbMessage);

root.setPrefWidth(700);
root.setPrefHeight(500);

AnchorPane.setTopAnchor(box1, 12.);
AnchorPane.setLeftAnchor(box1, 12.);

AnchorPane.setTopAnchor(box2, 12.);
AnchorPane.setRightAnchor(box2, 12.);

AnchorPane.setTopAnchor(tableView, 60.);
AnchorPane.setLeftAnchor(tableView, 12.);
AnchorPane.setRightAnchor(tableView, 12.);
AnchorPane.setBottomAnchor(tableView, 50.);

AnchorPane.setBottomAnchor(lbMessage, 12.);
AnchorPane.setLeftAnchor(lbMessage, 12.);

btnSearch.setOnAction(e -> {
    currentSearchText = tfSearch.getText();
    search();
});

btnNew.setOnAction(e -> new Form(this, null));

btnUpdate.setOnAction(e -> {
    Product product = tableView.getSelectionModel().getSelectedItem();
    if (product != null)
        new Form(this, product);
});

btnDelete.setOnAction(e -> {
    Product product = tableView.getSelectionModel().getSelectedItem();
    try {
        if (product != null && confirmed()) {
            backend.delete(product.getId());
            search();
        }
    } catch (BackendException ex) {
        showError(ex.getMessage());
    }
});

btnExit.setOnAction(e -> stage.close());

search();

Scene scene = new Scene(root);
scene.getStylesheets().add(getClass().getResource("Main.css")
    .toExternalForm());
```

```
stage.setOnCloseRequest(Event::consume);
stage.setTitle("Sortiment");
stage.setScene(scene);
stage.show();
}

@Override
public void stop() {
    backend.close();
}

public Stage getStage() {
    return stage;
}

public void search() {
    try {
        List<Product> products = backend.select(currentSearchText);
        ObservableList<Product> list = FXCollections
            .observableArrayList(products);
        tableView.setItems(list);
        lbMessage.setText("Anzahl: " + list.size());
    } catch (BackendException e) {
        showError(e.getMessage());
    }
}

public static void showError(String text) {
    Alert alert = new Alert(Alert.AlertType.ERROR);
    alert.setContentText(text);
    alert.showAndWait();
}

private boolean confirmed() {
    Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
    alert.setContentText("Soll das Produkt wirklich gelöscht werden?");
    Optional<ButtonType> option = alert.showAndWait();
    return option.get() == ButtonType.OK;
}

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

`init()` stellt die Verbindung zur Datenbank her.

Fehlermeldungen werden in einem Alert-Dialog angezeigt. Ein ausgewähltes Produkt kann erst nach Bestätigung gelöscht werden.

Mit

```
Platform.runLater(btnSearch::requestFocus);
```

wird der *Eingabe-Fokus* auf den Suchen-Button gesetzt, sobald die Oberfläche sichtbar ist.

Mit

```
new Form(this, null)
```

wird das Formular zur *Neuerfassung* und mit

```
new Form(this, product)
```

das Formular zum Ändern eines Produkts aufgerufen.

```
tableView.getSelectionModel().getSelectedItem()
```

liefert ein ausgewähltes Produkt (siehe auch Aufgabe 5 in Kapitel 28).

`backend.delete(...), backend.select(...), backend.close()` rufen die entsprechenden Methoden aus Backend zum Löschen, Abfragen von Produkten bzw. zum Schließen der Datenbankverbindung auf.

Datei *Main.css*:

```
.table-view .column-header#id .label {  
    -fx-alignment: CENTER_LEFT;  
}  
  
.table-view .column-header#name .label {  
    -fx-alignment: CENTER_LEFT;  
}  
  
.table-view .column-header#price .label {  
    -fx-alignment: CENTER_RIGHT;  
}  
  
.table-view .column-header#quantity .label {  
    -fx-alignment: CENTER_RIGHT;  
}
```

Die Klasse `Form` repräsentiert das Formular zur Neuerfassung bzw. zum Ändern eines Produkts. Ob es sich beim Aufruf aus `Main` um eine Neuerfassung oder eine Änderung handelt, wird im Konstruktor entschieden: `product == null` oder `!= null`.

```
package products;  
  
import javafx.application.Platform;  
import javafx.event.Event;  
import javafx.scene.Scene;  
import javafx.scene.control.Alert;  
import javafx.scene.control.Button;  
import javafx.scene.control.Label;  
import javafx.scene.control.TextField;  
import javafx.scene.layout.AnchorPane;  
import javafx.scene.layout.GridPane;  
import javafx.scene.layout.HBox;  
import javafx.stage.Modality;  
import javafx.stage.Stage;
```

```
public class Form {
    private final Main main;
    private final Stage stage;
    private final TextField tfId;
    private final TextField tfName;
    private final TextField tfPrice;
    private final TextField tfQuantity;
    private final TextField tfAdd;
    private final Label lbMessage;

    private int id;
    private String name;
    private double price;
    private int quantity;
    private int add;

    private boolean isNew;
    private final Backend backend;

    public Form(Main main, Product product) {
        this.main = main;
        backend = Backend.INSTANCE;

        if (product == null) {
            isNew = true;
        }

        stage = new Stage();
        AnchorPane anchorPane = new AnchorPane();
        GridPane gridPane = new GridPane();

        tfId = new TextField();
        tfId.setMaxWidth(100);
        tfName = new TextField();
        tfName.setPrefWidth(400);
        tfPrice = new TextField();
        tfPrice.setMaxWidth(100);
        tfQuantity = new TextField();
        tfQuantity.setMaxWidth(100);
        tfAdd = new TextField();
        tfAdd.setText("0");
        tfAdd.setMaxWidth(100);

        if (isNew) {
            tfPrice.setText("0");
            tfQuantity.setText("0");
        } else {
            tfId.setText(String.valueOf(product.getId()));
            tfName.setText(product.getName());
            tfPrice.setText(String.valueOf(product.getPrice()));
            tfQuantity.setText(String.valueOf(product.getQuantity()));
            tfId.setEditable(false);
            Platform.runLater(tfName::requestFocus);
        }

        lbMessage = new Label();
        lbMessage.setId("message");
    }
}
```

```
Button btnSave = new Button();
btnSave.setText("Speichern");

Button btnBack = new Button();
btnBack.setText("Zurück");

HBox box1 = new HBox();
box1.setSpacing(10);
Label lbAdd = new Label(" +/-");
lbAdd.setId("add");
box1.getChildren().addAll(tfQuantity, lbAdd, tfAdd);

HBox box2 = new HBox();
box2.setSpacing(10);
box2.getChildren().addAll(btnSave, btnBack);

gridPane.setHgap(20);
gridPane.setVgap(20);
gridPane.add(new Label("Id"), 0, 0);
gridPane.add(new Label("Name"), 0, 1);
gridPane.add(new Label("Preis"), 0, 2);
gridPane.add(new Label("Menge"), 0, 3);
gridPane.add(tfId, 1, 0);
gridPane.add(tfName, 1, 1);
gridPane.add(tfPrice, 1, 2);
gridPane.add(box1, 1, 3);
gridPane.add(lbMessage, 1, 4);
gridPane.add(box2, 1, 5);

anchorPane.getChildren().add(gridPane);
anchorPane.setPrefWidth(600);
anchorPane.setPrefHeight(400);
AnchorPane.setLeftAnchor(gridPane, 30.);
AnchorPane.setTopAnchor(gridPane, 30.);

btnSave.setOnAction(e -> save());
btnBack.setOnAction(e -> stage.close());

tfId.textProperty().addListener(
    (observableValue, oldValue, newValue) -> checkId());
tfName.textProperty().addListener(
    (observableValue, oldValue, newValue) -> checkName());
tfPrice.textProperty().addListener(
    (observableValue, oldValue, newValue) -> checkPrice());
tfQuantity.textProperty().addListener(
    (observableValue, oldValue, newValue) -> checkQuantity());
tfAdd.textProperty().addListener(
    (observableValue, oldValue, newValue) -> checkAdd());

Scene scene = new Scene(anchorPane);

scene.getStylesheets().add(getClass().getResource("Form.css")
    .toExternalForm());

stage.setOnCloseRequest(Event::consume);
stage.setScene(scene);
stage.setTitle("Produkt");
stage.setX(main.getStage().getX() + 50);
```

```
stage.setY(main.getStage().getY() + 50);

stage.initOwner(main.getStage());
stage.initModality(Modality.APPLICATION_MODAL);
stage.showAndWait();
}

private boolean checkId() {
    lbMessage.setText("");

    try {
        id = Integer.parseInt(tfId.getText());
    } catch (NumberFormatException e) {
        lbMessage.setText("Id muss eine ganze Zahl sein");
        tfId.requestFocus();
        return false;
    }
    return true;
}

private boolean checkName() {
    lbMessage.setText("");
    name = tfName.getText().trim();
    if (name.isEmpty()) {
        lbMessage.setText("Name fehlt");
        tfName.requestFocus();
        return false;
    }
    return true;
}

private boolean checkPrice() {
    lbMessage.setText("");
    try {
        price = Double.parseDouble(tfPrice.getText());
        if (price < 0) {
            lbMessage.setText("Preis ist negativ");
            tfPrice.requestFocus();
            return false;
        }
    } catch (NumberFormatException e) {
        lbMessage.setText("Preis ist keine Zahl");
        tfPrice.requestFocus();
        return false;
    }
    return true;
}

private boolean checkQuantity() {
    lbMessage.setText("");
    try {
        quantity = Integer.parseInt(tfQuantity.getText());

        if (quantity < 0) {
            lbMessage.setText("Menge ist < 0");
            return false;
        }
    }
}
```

```
        } catch (NumberFormatException e) {
            lbMessage.setText("Menge ist keine ganze Zahl");
            tfQuantity.requestFocus();
            return false;
        }
        return true;
    }

private boolean checkAdd() {
    lbMessage.setText("");

    try {
        add = Integer.parseInt(tfAdd.getText());
    } catch (NumberFormatException e) {
        lbMessage.setText("Zugang/Abgang ist keine ganze Zahl");
        tfAdd.requestFocus();
        return false;
    }
    return true;
}

private void save() {
    if (!checkId()) return;
    if (!checkName()) return;
    if (!checkPrice()) return;
    if (!checkQuantity()) return;
    if (!checkAdd()) return;

    quantity += add;
    if (quantity < 0) {
        lbMessage.setText("Neue Menge: " + quantity);
        tfAdd.requestFocus();
        return;
    }

    try {
        Product product = new Product(id, name, price, quantity);
        if (isNew) backend.insert(product);
        else backend.update(product);
        main.search();
        stage.close();
    } catch (BackendException e) {
        if (e.getCode() == 19) { // vendor-specific exception code
            Alert alert = new Alert(Alert.AlertType.ERROR);
            alert.setHeaderText("Id ist bereits vorhanden");
            alert.showAndWait();
        } else {
            Main.showError(e.getMessage());
            stage.close();
        }
    }
}
```

Handelt es sich um eine Änderung (`isNew == false`), ist das Feld *Id* nicht änderbar: `setEditable(false)`. Der *Eingabe-Fokus* wird auf das Feld *Name* gesetzt.

Die Methoden `checkXXX` prüfen die Eingaben auf Gültigkeit: Eingabe nicht leer, numerischer Inhalt, Wert nicht negativ.

Diese Prüfungen werden direkt bei der Eingabe und beim Speichern durchgeführt. Für die direkte Prüfung wird bei der *Text-Property* eines Feldes ein *ChangeListener* registriert:

```
field.textProperty().addListener(  
    (observableValue, oldValue, newValue) -> checkXXX());
```

Die letzten drei Codezeilen im Konstruktor bewirken, dass das Fenster *modal* ist. Solange es angezeigt wird, ist das Hauptfenster nicht zugänglich.

Nach erfolgreicher Neuerfassung (`backend.insert(...)`) bzw. Änderung (`backend.update(...)`) werden die im Hauptfenster angezeigten Tabellendaten aktualisiert: `main.search()`.

Datei *Form.css*:

```
.label {  
    -fx-font-weight: bold;  
}  
  
.add {  
    -fx-font-size: 14;  
}  
  
.message {  
    -fx-text-fill: red;  
}
```

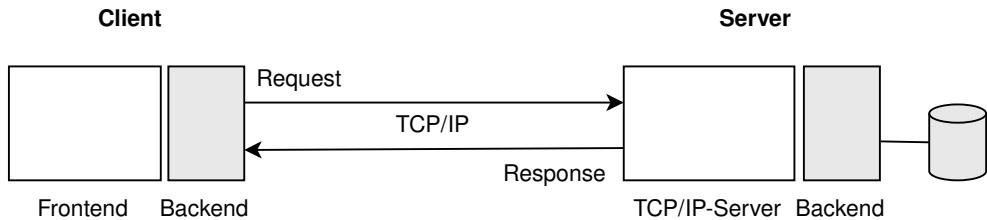
### 31.3 Variante 2: Client/Server-Anwendung

In diesem Abschnitt entwickeln wir – ausgehend von Variante 1 – eine geteilte Anwendung. GUI und Anwendungslogik bilden den Client, die Datenhaltung (Datenbankzugriff mit JDBC) ist Aufgabe des Servers.

Da Client und Server unterschiedliche Prozesse auf evtl. verschiedenen Rechnern sind, müssen die Anfragen an den Server sowie die Ergebnisse über das Netz übertragen werden. Hierbei orientieren wir uns am Beispiel in Kapitel 30.2.

Abbildung 31-5 zeigt die Architektur der Anwendung. Der Client sendet Anfragen an den Server und empfängt Antworten. Hierfür ist die Klasse `Backend` zuständig.

Der Server nimmt Anfragen entgegen und sendet Antworten an den Client. Die Klasse `Backend` des Servers implementiert die Datenbankzugriffe.



**Abbildung 31-5:** Client/Server-Anwendung

## Übertragungsformat

Für die Übertragung von Daten zwischen Client und Server verwenden wir das Datenformat *JSON (JavaScript Object Notation)*.<sup>2</sup> Hierbei handelt es sich um ein schlankes Datenaustauschformat, das für Menschen einfach zu lesen und für Maschinen einfach zu analysieren ist.

Für die Nutzung von JSON in Java gibt es eine Reihe von Bibliotheken. Wir nutzen hier *Gson* von Google.<sup>3</sup> Das Begleitmaterial zu diesem Buch enthält die jar-Datei *gson-n.n.n.jar*.

Zunächst zeigen wir den Umgang mit JSON anhand eines kleinen Beispiels. Dabei werden Objekte in einen JSON-String konvertiert und umgekehrt.

```

package gsontest;

import com.google.gson.Gson;
import java.util.Arrays;

public class GsonTest {
    static class Person {
        int id;
        String name;
        String[] skills;

        public Person(int id, String name, String[] skills) {
            this.id = id;
            this.name = name;
            this.skills = skills;
        }
    }
}

```

2 [https://de.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](https://de.wikipedia.org/wiki/JavaScript_Object_Notation)

3 <https://github.com/google/gson>

```

public static void main(String[] args) {
    Gson gson = new Gson();

    Person p1 = new Person(1, "Hugo", new String[]{"Java", "Python"});
    Person p2 = new Person(2, "Emil", null);
    String json = gson.toJson(new Person[]{p1, p2});
    System.out.println(json);

    Person[] persons = gson.fromJson(json, Person[].class);
    for (Person p : persons) {
        System.out.println(p.id + " " + p.name + " " + Arrays.toString(p.skills));
    }
}
}

```

Ausgabe:

```
[{"id":1,"name":"Hugo","skills":["Java","Python"]}, {"id":2,"name":"Emil"}]
1 Hugo [Java, Python]
2 Emil null
```

Schöner aufbereitet sieht der JSON-String wie folgt aus:

```
[
  {
    "id": 1,
    "name": "Hugo",
    "skills": [
      "Java",
      "Python"
    ]
  },
  {
    "id": 2,
    "name": "Emil"
  }
]
```

Ein JSON-Objekt ist eine Menge von Paaren aus Schlüssel und Wert, geklammert mit { und }. Arrays beginnen mit [ und enden mit ]. Man sieht hier auch, dass Paare mit null-Werten nicht serialisiert werden.

Der Client muss dem Server mitteilen, welche Operation ausgeführt werden soll (*select, insert, update oder delete*). Zudem müssen die nötigen Nutzdaten sowie Fehlermeldungen übertragen werden.

Die Klassen `Request` und `Response` repräsentieren Anfrage und Antwort (siehe Abbildung 31-5). Beide Klassen sind jeweils in der Klasse `Backend` des Clients und der Klasse `ProductServer` des Servers als statische Klassen eingefügt.

```
static class Request {  
    String op;  
    String text;  
    Product product;  
}  
  
static class Response {  
    boolean exception;  
    String message;  
    int code;  
    Product[] products;  
}
```

Request enthält die auszuführende Operation, einen optionalen Text (Suchbegriff für *select*, Primärschlüssel für *delete*) und optionale Produktdaten für *insert* und *update*.

Response enthält Text und Fehler-Code für eine Fehlermeldung, falls exception den Wert true hat. Im Fall von *select* werden Produktdaten geliefert.

## Der Server

Der Server ist analog zum EchoServer aus Kapitel 30.2 aufgebaut.

Die Klassen `Product`, `Backend` und `BackendException` werden unverändert aus *Variante 1* übernommen.

Sofort nach dem Start des Servers wird die Verbindung zur Datenbank hergestellt. Diese Verbindung wird für alle Threads verwendet.

Es wird ein Shutdown-Hook erstellt, der die Datenbank bei Terminierung (durch Strg + C) schließt.

In der `while`-Schleife der Methode `run` wird die Operation ermittelt und dann die zuständige Methode aufgerufen.

```
package products;  
  
import com.google.gson.Gson;  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.PrintWriter;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.net.SocketException;  
import java.util.List;  
  
public class ProductServer implements Runnable {  
    private final Socket client;  
    private final Backend backend = Backend.INSTANCE;  
    private final Gson gson;
```

```
public ProductServer(Socket client) {
    this.client = client;
    gson = new Gson();
    System.out.println(client + " connected");
}

@Override
public void run() {
    try (BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream())));
        PrintWriter out = new PrintWriter(client.getOutputStream(), true)) {

        String input;
        while ((input = in.readLine()) != null) {
            Request request = gson.fromJson(input, Request.class);
            switch (request.op) {
                case "select" -> {
                    System.out.println("select");
                    select(request.text, out);
                }
                case "insert" -> {
                    System.out.println("insert");
                    insert(request.product, out);
                }
                case "update" -> {
                    System.out.println("update");
                    update(request.product, out);
                }
                case "delete" -> {
                    System.out.println("delete");
                    delete(request.text, out);
                }
                default -> {
                    System.out.println("Ungültige Operation");
                    Response response = new Response();
                    response.exception = true;
                    response.message = "Ungültige Operation";
                    response.code = -1;
                    out.println(gson.toJson(response));
                }
            }
        }
    } catch (SocketException ignored) {
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}

System.out.println(client + " closed");
}

private void select(String text, PrintWriter out) {
    Response response = new Response();
    try {
        text = (text == null) ? "" : text;
        List<Product> list = backend.select(text);
        response.products = list.toArray(new Product[0]);
    }
}
```

```
        } catch (BackendException e) {
            response.exception = true;
            response.message = e.getMessage();
            response.code = e.getCode();
        }
        out.println(gson.toJson(response));
    }

private void insert(Product product, PrintWriter out) {
    Response response = new Response();
    try {
        backend.insert(product);
    } catch (BackendException e) {
        response.exception = true;
        response.message = e.getMessage();
        response.code = e.getCode();
    }
    out.println(gson.toJson(response));
}

private void update(Product product, PrintWriter out) {
    Response response = new Response();
    try {
        backend.update(product);
    } catch (BackendException e) {
        response.exception = true;
        response.message = e.getMessage();
        response.code = e.getCode();
    }
    out.println(gson.toJson(response));
}

private void delete(String id, PrintWriter out) {
    Response response = new Response();
    try {
        backend.delete(Integer.parseInt(id));
    } catch (BackendException e) {
        response.exception = true;
        response.message = e.getMessage();
        response.code = e.getCode();
    }
    out.println(gson.toJson(response));
}

static class Request {
    String op;
    String text;
    Product product;
}

static class Response {
    boolean exception;
    String message;
    int code;
    Product[] products;
}
```

```
public static void main(String[] args) {
    int port = 50000;
    if (args.length == 1) {
        port = Integer.parseInt(args[0]);
    }

    Backend backend = Backend.INSTANCE;
    try {
        // dieselbe Verbindung für alle Threads
        backend.connect();
    } catch (BackendException e) {
        System.err.println(e.getMessage());
        return;
    }

    Runtime.getRuntime().addShutdownHook(new Thread(backend::close));

    try (ServerSocket server = new ServerSocket(port)) {
        System.out.println("ProductServer auf " + port + " gestartet ...");
        while (true) {
            Socket client = server.accept();
            new Thread(new ProductServer(client)).start();
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Aufruf des Servers im Terminal (Beispiel):

```
java -cp out/production/server/server/lib/gson-2.10.1.jar:\server/lib/sqlite-jdbc-3.42.0.0.jar products.ProductServer
```

Optional kann eine andere Portnummer als Parameter angegeben werden.

## Der Client

Um die Funktionsfähigkeit des Servers zu prüfen, wird zunächst ein `TestClient` ohne grafische Oberfläche erstellt.

```
package products;

import com.google.gson.Gson;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class TestClient {
    private static final String host = "localhost";
    private static final int port = 50000;
```

```
private final Socket socket;
private final BufferedReader in;
private final PrintWriter out;
private final Gson gson;

public TestClient() throws IOException {
    socket = new Socket(host, port);
    in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    out = new PrintWriter(socket.getOutputStream(), true);
    gson = new Gson();
}

public void close() throws IOException {
    if (socket != null) {
        socket.close();
    }
}

public static void main(String[] args) {
    try {
        TestClient client = new TestClient();

        client.select("");
        client.insert(new Product(4711, "Hammer", 12.99, 100));
        client.insert(new Product(4712, "Zange", 10, 50));
        client.insert(new Product(4713, "Bohrer", 30, 10));
        client.update(new Product(4711, "Hammer", 12.99, 120));
        client.select("");
        client.delete(4712);
        client.select("");
        client.select("Boh");

        client.close();
    } catch (IOException e) {
        System.err.println(e.getMessage());
    } catch (BackendException e) {
        System.out.println(e.getCode() + " " + e.getMessage());
    }
}

public void select(String text) throws BackendException, IOException {
    Request request = new Request();
    request.op = "select";
    request.text = text;
    String json = gson.toJson(request);
    out.println(json);

    String line = in.readLine();
    if (line != null) {
        Response response = gson.fromJson(line, Response.class);
        if (response.exception) {
            throw new BackendException(response.message, response.code);
        }
        for (Product p : response.products) {
            System.out.println(p.getId() + " " + p.getName() + " " +
                p.getPrice() + " " + p.getQuantity());
        }
    }
}
```

```
public void insert(Product product) throws BackendException, IOException {
    Request request = new Request();
    request.op = "insert";
    request.product = product;
    String json = gson.toJson(request);
    out.println(json);

    String line = in.readLine();
    if (line != null) {
        Response response = gson.fromJson(line, Response.class);
        if (response.exception) {
            throw new BackendException(response.message, response.code);
        }
    }
}

public void update(Product product) throws BackendException, IOException {
    Request request = new Request();
    request.op = "update";
    request.product = product;
    String json = gson.toJson(request);
    out.println(json);

    String line = in.readLine();
    if (line != null) {
        Response response = gson.fromJson(line, Response.class);
        if (response.exception) {
            throw new BackendException(response.message, response.code);
        }
    }
}

public void delete(int id) throws BackendException, IOException {
    Request request = new Request();
    request.op = "delete";
    request.text = String.valueOf(id);
    String json = gson.toJson(request);
    out.println(json);

    String line = in.readLine();
    if (line != null) {
        Response response = gson.fromJson(line, Response.class);
        if (response.exception) {
            throw new BackendException(response.message, response.code);
        }
    }
}

static class Request {
    String op;
    String text;
    Product product;
}

static class Response {
    boolean exception;
    String message;
```

```

        int code;
        Product[] products;
    }
}

```

**Test**

1. Server starten
2. TestClient starten

Ausgabe des Clients (wenn die Datenbank beim Start noch nicht existiert oder leer ist):

```

4711 Hammer 12.99 120
4712 Zange 10.0 50
4713 Bohrer 30.0 10
4711 Hammer 12.99 120
4713 Bohrer 30.0 10
4713 Bohrer 30.0 10

```

Ausgabe des Servers (Beispiel):

```

ProductServer auf 50000 gestartet ...
Socket[addr=/127.0.0.1,port=46774,localport=50000] connected
select
insert
insert
insert
update
select
delete
select
select
Socket[addr=/127.0.0.1,port=46774,localport=50000] closed

```

Die Klassen `Main` und `Form` aus Variante 1 müssen nur an wenigen Stellen für die Client/Server-Anwendung angepasst werden.

Die neue Klasse `Backend` hat hier die Aufgabe, Daten zum Server zu schicken und Antworten entgegenzunehmen.

Die Methodensignaturen entsprechen denen aus der Klasse `Backend` in Variante 1 bis auf die um `IOException` ergänzte `throws`-Klausel.

Hier zunächst die Klasse `Backend` des Clients:

```

package products;

import com.google.gson.Gson;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

```

```
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Arrays;
import java.util.List;

public class Backend {
    public static final Backend INSTANCE = new Backend();
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    private final Gson gson;

    private Backend() {
        gson = new Gson();
    }

    public void connect(String host, int port) throws IOException {
        socket = new Socket(host, port);
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
    }

    public void close() {
        if (socket != null) {
            try {
                socket.close();
            } catch (IOException ignored) {
            }
        }
    }

    public List<Product> select(String text) throws BackendException, IOException {
        Request request = new Request();
        request.op = "select";
        request.text = text;
        String json = gson.toJson(request);
        out.println(json);
        String line = in.readLine();
        if (line != null) {
            Response response = gson.fromJson(line, Response.class);
            if (response.exception) {
                throw new BackendException(response.message, response.code);
            }
            return Arrays.asList(response.products);
        }
        return List.of();
    }

    public void insert(Product product) throws BackendException, IOException {
        Request request = new Request();
        request.op = "insert";
        request.product = product;
        String json = gson.toJson(request);
        out.println(json);
        String line = in.readLine();
        if (line != null) {
            Response response = gson.fromJson(line, Response.class);
```

```
        if (response.exception) {
            throw new BackendException(response.message, response.code);
        }
    }

public void update(Product product) throws BackendException, IOException {
    Request request = new Request();
    request.op = "update";
    request.product = product;
    String json = gson.toJson(request);
    out.println(json);
    String line = in.readLine();
    if (line != null) {
        Response response = gson.fromJson(line, Response.class);
        if (response.exception) {
            throw new BackendException(response.message, response.code);
        }
    }
}

public void delete(int id) throws BackendException, IOException {
    Request request = new Request();
    request.op = "delete";
    request.text = String.valueOf(id);
    String json = gson.toJson(request);
    out.println(json);
    String line = in.readLine();
    if (line != null) {
        Response response = gson.fromJson(line, Response.class);
        if (response.exception) {
            throw new BackendException(response.message, response.code);
        }
    }
}

static class Request {
    String op;
    String text;
    Product product;
}

static class Response {
    boolean exception;
    String message;
    int code;
    Product[] products;
}
}
```

Befindet sich der Server auf einem anderen über Netz verbundenen Rechner, muss localhost durch die IP-Nummer dieses Rechners ersetzt werden. Rechnername und Portnummer können als Aufrufparameter angegeben werden.

Änderungen in Main gegenüber Main aus Variante 1:

```
public void init() {
    String host = "localhost";
    int port = 50000;

    List<String> params = getParameters().getRaw();

    if (params.size() == 2) {
        host = params.get(0);
        port = Integer.parseInt(params.get(1));
    }

    try {
        backend = Backend.INSTANCE;
        backend.connect(host, port);
    } catch (IOException e) {
        errorMessage = e.getMessage();
    }
}

public void start(Stage stage) {
    ...

    btnDelete.setOnAction(e -> {
        Product product = tableView.getSelectionModel().getSelectedItem();
        try {
            if (product != null && confirmed()) {
                backend.delete(product.getId());
                search();
            }
        } catch (BackendException | IOException ex) {
            showError(ex.getMessage());
        }
    });
}

public void search() {
    try {
        ...
    } catch (BackendException | IOException e) {
        showError(e.getMessage());
    }
}
```

Änderungen in Form gegenüber Form aus Variante 1:

Zusätzlicher catch-Zweig in der Methode save der Klasse Form:

```
        catch (IOException e) {  
            Main.showError(e.getMessage());  
            stage.close();  
        }
```

## 31.4 Aufgaben

### Aufgabe 1

Testen Sie Variante 2 in den folgenden Fällen:

- a) Client und Server auf demselben Rechner,
- b) Client und Server auf verschiedenen Rechnern,
- c) Wie verhält sich der Client, wenn der Server nicht gestartet ist oder nicht im Netz erreichbar ist?



## 32 Prinzipien objektorientierten Designs

Hat ein Softwareprojekt eine bestimmte Größe (ausgedrückt als Anzahl von Programmen, Modulen, Klassen usw.) erreicht, kann es zunehmend aufwändiger werden, Programme zu warten, Fehler zu finden, Änderungen und Erweiterungen einzubringen.

Es erfordert viel Erfahrung und Wissen, schon zu Beginn der Entwicklung die richtigen Entwurfsentscheidungen zu fällen, die die Verständlichkeit des Quellcodes, die Wartbarkeit und Erweiterbarkeit ohne Hindernisse ermöglichen.

Unter *Clean Code* versteht man Prinzipien und Maßnahmen für "sauberen", d. h. übersichtlichen und verständlichen Quellcode (erstmalig im gleichnamigen Buch beschrieben von Robert C. Martin). Ein Teil dieser Prinzipien wurde von Martin als *SOLID-Prinzipien* beschrieben.

In diesem Kapitel werden die SOLID-Prinzipien anhand von einfachen Beispielen erläutert. Zu jedem Prinzip werden ein Negativbeispiel sowie eine Verbesserung unter Anwendung des jeweiligen SOLID-Prinzips vorgestellt.

Das sind die fünf SOLID-Prinzipien:

- **Single-Responsibility-Prinzip (SRP)**
- **Open-Closed-Prinzip (OCP)**
- **Liskovsches Substitutionsprinzip (LSP)**
- **Interface-Segregation-Prinzip (ISP)**
- **Dependency-Inversion-Prinzip (DIP)**

### Lernziele

In diesem Kapitel lernen Sie

- welche fünf Prinzipien hinter SOLID stehen und
- wie die Anwendung dieser Prinzipien zu besserem Code führen kann.

### 32.1 Single-Responsibility-Prinzip

Das *Single-Responsibility-Prinzip* (SRP) besagt, dass eine Klasse nur eine Zuständigkeit haben soll.

"*Es sollte nie mehr als einen Grund dafür geben, eine Klasse zu ändern.*"<sup>1</sup>

---

<sup>1</sup> Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002, S. 149–153

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_32](https://doi.org/10.1007/978-3-658-43574-5_32).

### Negativbeispiel

Die folgende Klasse `Contact` hat die Zuständigkeit, das Datenmodell für Kontakte zu repräsentieren. Die Klasse bietet aber auch die Methode `save` zum Speichern in einem einfachen Textformat (CSV = Comma-separated values) an.

```
package bad;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Contact {
    private String name;
    private String email;

    public Contact() {
    }

    public Contact(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void save(String filename) throws IOException {
        Path path = Paths.get(filename);
        Files.writeString(path, name + "," + email);
    }
}
```

Wenn sich die Art des Speicherns ändert oder eine neue Art hinzugefügt werden soll, muss die Klasse angepasst werden, obwohl sich das Datenmodell selbst nicht geändert hat.

### Verbesserung

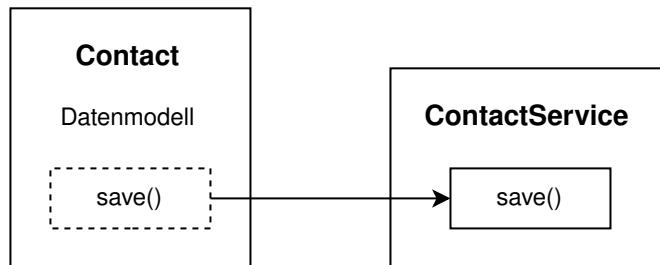
Die Methode zum Speichern wird in eine eigene Klasse als statische Methode ausgelagert. Hier kann dann die Implementierung geändert oder eine neue Speichermethode hinzugefügt werden, ohne die Klasse `Contact` zu berühren.

```
package good;

import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class ContactService {
    public static void save(Contact contact, String filename) throws IOException {
        Path path = Paths.get(filename);
        Files.writeString(path, contact.getName() + "," + contact.getEmail());
    }

    public static void saveAsXml(Contact contact, String filename)
        throws IOException {
        try (XMLEncoder encoder = new XMLEncoder(new BufferedOutputStream(
            new FileOutputStream(filename)))) {
            encoder.writeObject(contact);
        }
    }
}
```



**Abbildung 32-1:** Auslagern einer Methode

## 32.2 Open-Closed-Prinzip

Das *Open-Closed-Prinzip* (OCP) besagt, dass Klassen offen für Erweiterungen, aber geschlossen für Änderungen sein sollen.

Das heißt, der Quellcode und benutzte Interfaces sollen sich nicht ändern.

*"Module sollten sowohl offen (für Erweiterungen), als auch geschlossen (für Modifikationen) sein."<sup>2</sup>*

Eine Möglichkeit der Erweiterung ist die *Vererbung*. Sie verändert nicht das Verhalten der Basisklasse, erweitert diese aber in der abgeleiteten Klasse um zusätzliche Daten und Methoden. Eine andere Möglichkeit ist die *Delegation*, also die Übergabe der Verantwortung für eine bestimmte Aufgabe an eine andere Klasse.

### Negativbeispiel

Die Klasse `ContactService` implementiert hier die Methode `save`, die verschiedene Speicheroptionen anbietet, auswählbar über den ersten Aufrufparameter `type`.

```
package bad;

import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class ContactService {
    public static void save(String type, Contact contact, String filename)
        throws IOException {

        Path path = Paths.get(filename);
        switch (type) {
            case "txt" -> Files.writeString(path, contact.getName() + "," +
                contact.getEmail());
            case "xml" -> {
                try (XMLEncoder xmlEncoder = new XMLEncoder(
                    new FileOutputStream(filename))) {
                    xmlEncoder.writeObject(contact);
                }
            }
            default -> throw new IllegalArgumentException();
        }
    }
}
```

---

<sup>2</sup> Bertrand Meyer: Object Oriented Software Construction. Prentice Hall, 1988, S. 57–61

Soll eine weitere Option hinzugefügt werden, muss die Klasse `ContactService` geändert werden.

### Verbesserung

Wir wollen die Klasse `ContactService` stabil halten. Speicheroptionen sollen in eigenen Klassen implementiert werden.

Zu diesem Zweck führen wir das Interface `Save` ein:

```
package good;

import java.io.IOException;

public interface Save {
    void save(Contact contact, String filename) throws IOException;
}

package good;

import java.io.IOException;

public class ContactService {
    private Contact contact;
    private Save save;

    public void setContact(Contact contact) {
        this.contact = contact;
    }

    public void setSave(Save save) {
        this.save = save;
    }

    public void save(String filename) throws IOException {
        save.save(contact, filename);
    }
}
```

Die Klassen, die jeweils eine Speicheroption realisieren, implementieren alle das Interface `Save`. Eine Instanz wird über die `set`-Methode von `ContactService` gesetzt.

```
package good;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class SaveAsCsv implements Save {
    @Override
    public void save(Contact contact, String filename) throws IOException {
```

```

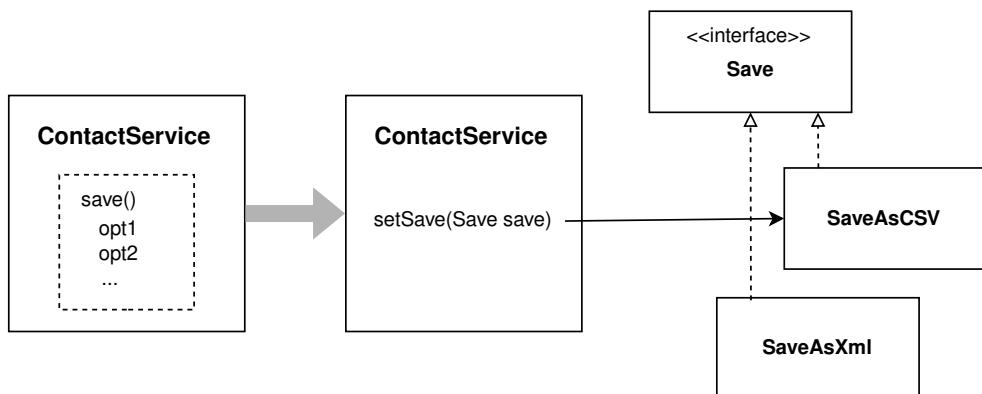
        Path path = Paths.get(filename);
        Files.writeString(path, contact.getName() + "," + contact.getEmail());
    }
}

package good;

import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.io.IOException;

public class SaveAsXml implements Save {
    @Override
    public void save(Contact contact, String filename) throws IOException {
        try (XMLEncoder xmlEncoder = new XMLEncoder(
                new FileOutputStream(filename))) {
            xmlEncoder.writeObject(contact);
        }
    }
}

```



**Abbildung 32-2:** Speicheroptionen als eigene Klassen

```

package good;

public class Test {
    public static void main(String[] args) throws Exception {
        Contact contact = new Contact("Hugo Meier", "hugo.meier@web.de");
        ContactService contactService = new ContactService();
        contactService.setContact(contact);

        contactService.setSave(new SaveAsCsv());
        contactService.save("contact.txt");

        contactService.setSave(new SaveAsXml());
        contactService.save("contact.xml");
    }
}

```

## 32.3 Liskovsches Substitutionsprinzip

Das *Liskovsche Substitutionsprinzip* (LSP), auch *Ersetzbarkeitsprinzip* genannt, fordert, dass überall, wo eine Basisklasse eingesetzt wird, auch eine von ihr abgeleitete Subklasse verwendet werden kann, ohne dass sich das von der Basisklasse erwartete Verhalten hierdurch ändert.

"Sei  $q(x)$  eine Eigenschaft des Objektes  $x$  vom Typ  $T$ , dann sollte  $q(y)$  für alle Objekte  $y$  des Typs  $S$  gelten, wobei  $S$  ein Subtyp von  $T$  ist."<sup>3</sup>

Es muss also darauf geachtet werden, dass die Basisfunktionalität der vererbenden Klasse nicht geändert wird.

### Negativbeispiel

Ein Quadrat ist ein spezielles Rechteck, also entwerfen wir die Klasse `Square` als Subklasse von `Rectangle`. Dabei müssen wir sicherstellen, dass Quadrate immer gleich lange Seiten haben: die Breite muss mit der Höhe übereinstimmen.

```
package bad;

public class Rectangle {
    private int width;
    private int height;

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}
```

---

<sup>3</sup> Barbara H. Liskov, Jeannette M. Wing: Behavioral Subtyping Using Invariants and Constraints. Hrsg.: MIT Lab. for Computer Science, School of Computer Science, Carnegie Mellon University. Prentice Hall, Pittsburgh Juli 1999

```

package bad;

public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}

package bad;

public class Test {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        rectangle.setWidth(2);
        rectangle.setHeight(5);
        System.out.println(rectangle.getArea());

        Rectangle rect = new Square();
        rect.setWidth(2);
        rect.setHeight(5);
        System.out.println(rect.getArea());
    }
}

```

Die Fläche für das zweite Rechteck `rect` beträgt 25, obwohl 10 erwartet wird. Das Quadrat verhält sich also nicht wie ein Rechteck.

Mit

```
rect.setHeight(5);
```

wird die Methode `setHeight` in `Square` ausgeführt und damit Höhe und Breite auf 5 gesetzt.

### Verbesserung

Quadrat verhalten sich anders als Rechtecke. Deshalb soll `Square` keine Subklasse von `Rectangle` sein.

```

package good;

public class Rectangle {
    private int width;
    private int height;
}

```

```
public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
}

public int getWidth() {
    return width;
}

public void setWidth(int width) {
    this.width = width;
}

public int getHeight() {
    return height;
}

public void setHeight(int height) {
    this.height = height;
}

public int getArea() {
    return width * height;
}
}

package good;

public class Square {
    private int length;

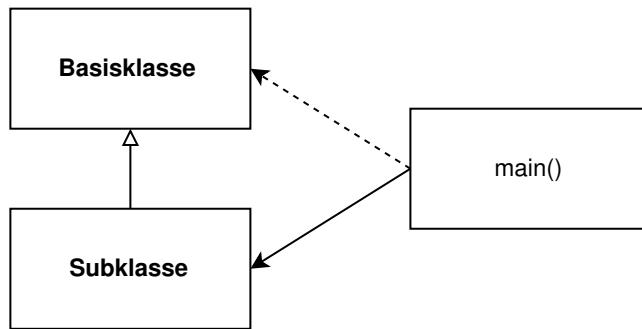
    public Square(int length) {
        this.length = length;
    }

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public int getArea() {
        return length * length;
    }
}
```

Beide Klassen können natürlich eine gemeinsame Basisklasse erweitern oder ein gemeinsames Interface implementieren, um zu dokumentieren, dass sie geometrische Formen darstellen.



**Abbildung 32-3:** Eine Instanz der Basisklasse kann durch eine Instanz der Subklasse ersetzt werden.

## 32.4 Interface-Segregation-Prinzip

Das *Interface-Segregation-Prinzip* (ISP) besagt, dass Klassen nicht von Interfaces abhängen sollen, von denen einige Methoden nicht benötigt werden. Zu große Interfaces sollen in kleinere aufgeteilt werden.

*"Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden."*<sup>4</sup>

### Negativbeispiel

Ein Sportler tritt zum Wettkampf an, schwimmt und/oder läuft. Hierzu gibt es das folgende Interface:

```

package bad;

public interface Athlete {
    void compete();
    void swim();
    void run();
}
  
```

Die Klasse `Swimmer` implementiert dieses Interface:

```

package bad;

public class Swimmer implements Athlete {
    private final String name;
  
```

---

<sup>4</sup> Robert C. Martin: The Interface Segregation Principle. Object Mentor, 1996

```
public Swimmer(String name) {  
    this.name = name;  
}  
  
@Override  
public void compete() {  
    System.out.println(name + " startet");  
}  
  
@Override  
public void swim() {  
    System.out.println(name + " schwimmt");  
}  
  
@Override  
public void run() {  
    throw new UnsupportedOperationException("Methode nicht implementiert");  
}  
}
```

Die Methode `run` wird nicht benötigt, muss aber formal implementiert werden. Ihr Aufruf führt zu einem Laufzeitfehler.

### Verbesserung

Das Interface `Athlete` wird wie folgt aufgeteilt:

```
package good;  
  
public interface Athlete {  
    void compete();  
}  
  
package good;  
  
public interface Runner extends Athlete {  
    void run();  
}  
  
package good;  
  
public interface Swimmer extends Athlete {  
    void swim();  
}
```

Läufer und Schwimmer implementieren jeweils die benötigten Interfaces.

```

package good;

public class RunnerImpl implements Runner {
    private final String name;

    public RunnerImpl(String name) {
        this.name = name;
    }

    @Override
    public void compete() {
        System.out.println(name + " startet");
    }

    @Override
    public void run() {
        System.out.println(name + " läuft");
    }
}

package good;

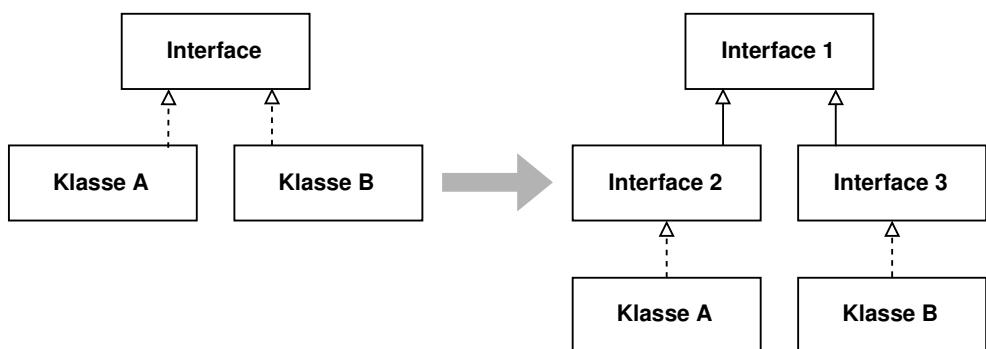
public class SwimmerImpl implements Swimmer {
    private final String name;

    public SwimmerImpl(String name) {
        this.name = name;
    }

    @Override
    public void compete() {
        System.out.println(name + " startet");
    }

    @Override
    public void swim() {
        System.out.println(name + " schwimmt");
    }
}

```



**Abbildung 32-4:** Aufteilung von Interfaces

## 32.5 Dependency-Inversion-Prinzip

Die Anwendung des *Dependency-Inversion-Prinzips* (DIP) soll die allzu enge Kopplung zwischen Klassen reduzieren. Klassen höherer Ebenen sollen nicht von Klassen niedrigerer Ebenen abhängen, sie sollen nur von Abstraktionen abhängen.

*"Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen."*<sup>5</sup>

Was hiermit konkret gemeint ist, erläutert das folgende Beispiel.

### Negativbeispiel

Ein Projekt hat einen Leiter und beschäftigt mehrere Java-Programmierer. Das kann wir folgt modelliert werden.

```
package bad;

public class Manager {
    private final String name;

    public Manager(String name) {
        this.name = name;
    }

    public void manage() {
        System.out.println(name + " leitet das Projekt");
    }

    @Override
    public String toString() {
        return "Manager: " + name;
    }
}

package bad;

public class JavaProgrammer {
    private final String name;

    public JavaProgrammer(String name) {
        this.name = name;
    }

    public void develop() {
        System.out.println(name + " programmiert in Java");
    }
}
```

---

5 Robert C. Martin: The Dependency Inversion Principle. Object Mentor, Mai 1996

```
@Override
public String toString() {
    return "JavaProgrammer: " + name;
}
}

package bad;

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

public class Project {
    private List<JavaProgrammer> javaProgrammers = new ArrayList<>();
    private Manager manager;

    public Manager getManager() {
        return manager;
    }

    public void setManager(Manager manager) {
        this.manager = manager;
    }

    public List<JavaProgrammer> getJavaProgrammers() {
        return javaProgrammers;
    }

    public void setJavaProgrammers(List<JavaProgrammer> javaProgrammers) {
        this.javaProgrammers = javaProgrammers;
    }

    public void add(JavaProgrammer developer) {
        javaProgrammers.add(developer);
    }

    @Override
    public String toString() {
        return "Project: " + manager + ", " + javaProgrammers;
    }
}
```

Man sieht, dass die Klasse Project von JavaProgrammer abhängig ist. Es ist hier nicht möglich, Personen mit anderen Fachgebieten in die Liste aufzunehmen.

## Verbesserung

Die Abhängigkeit eines Projekts vom konkreten "Detail" Java-Programmierer soll umgekehrt werden: Ein Projekt soll von einer "höheren" Abstraktion abhängig sein, hier vom Interface Developer. Das erlaubt dann, dass in die Liste verschiedene Arten von Entwicklern eingetragen werden können, also alle Instanzen, deren Klassen das Interface Developer implementieren. Es besteht damit keine Abhängigkeit mehr von einem konkreten Detail (siehe Abbildung 32-5).

```
package good;

public interface Developer {
    void develop();
}

package good;

public class JavaProgrammer implements Developer {
    private final String name;

    public JavaProgrammer(String name) {
        this.name = name;
    }

    @Override
    public void develop() {
        System.out.println(name + " programmiert in Java");
    }

    @Override
    public String toString() {
        return "JavaProgrammer: " + name;
    }
}

package good;

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

public class Project {
    private List<Developer> developers = new ArrayList<>();
    private Manager manager;

    public Manager getManager() {
        return manager;
    }

    public void setManager(Manager manager) {
        this.manager = manager;
    }

    public List<Developer> getDevelopers() {
        return developers;
    }

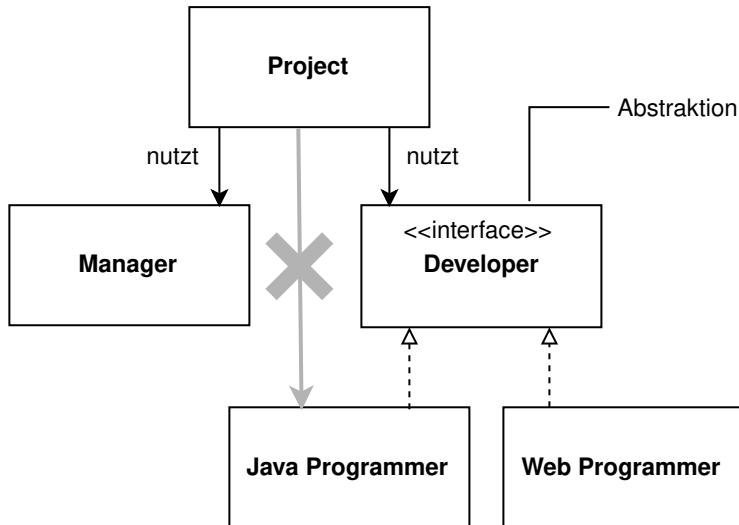
    public void setDevelopers(List<Developer> developers) {
        this.developers = developers;
    }

    public void add(Developer developer) {
        developers.add(developer);
    }
}
```

```

@Override
public String toString() {
    return "Project: " + manager + ", " + developers;
}
}

```



**Abbildung 32-5:** Dependency Inversion

## 32.6 Zusammenfassung

**Tabelle 32-1:** Die SOLID-Prinzipien

<b>Single Responsibility</b>	Eine Klasse soll nur eine Verantwortung haben.
<b>Open Closed</b>	Klassen sollen zur Erweiterung geöffnet, aber zur Änderung geschlossen sein.
<b>Liskov's Substitution</b>	Abgeleitete Klassen müssen ihre Basisklassen vollständig ersetzen können.
<b>Interface Segregation</b>	Klassen sollen nicht gezwungen werden, unnötige Methoden zu implementieren, die sie nicht verwenden.
<b>Dependency Inversion</b>	Klassen sollen von Abstraktionen abhängen, nicht von konkreten Details.

## 32.7 Aufgaben

### Aufgabe 1

Ein User-Service kann die Mail-Adresse eines Users ändern. Zuvor muss aber der User authentifiziert werden. Hierzu wurde eine Klasse wie folgt realisiert:

```
public class UserService {  
    public void changeEmail(User user) {  
        if (checkAccess(user)) {  
            //Grant option to change  
        }  
    }  
  
    public boolean checkAccess(User user) {  
        // check the access  
        return false;  
    }  
}
```

Die Klasse verstößt gegen das *Single-Responsibility-Prinzip*. Verbessern Sie den Entwurf.

*Lösung: Unterprojekt SRP*

### Aufgabe 2

Eine Methode soll den Gesamtflächeninhalt von Rechtecken und Kreisen berechnen:

```
public static double getArea(Object[] shapes) {  
    double area = 0;  
    for (Object shape : shapes) {  
        if (shape instanceof Rectangle rectangle) {  
            area += rectangle.width * rectangle.height;  
        } else if (shape instanceof Circle circle) {  
            area += circle.radius * circle.radius * Math.PI;  
        } else {  
            throw new IllegalArgumentException();  
        }  
    }  
    return area;  
}
```

Die Klasse `Area`, die diese Methode anbietet, verstößt gegen das *Open-Closed-Prinzip*. Verbessern Sie den Entwurf. Tipp: Führen Sie das Interface `Shape` mit der Methode `getArea()` ein, das vom Rechteck und vom Kreis implementiert wird.

*Lösung: Unterprojekt OCP*

### Aufgabe 3

Betrachten Sie die folgende Klassenhierarchie. Jede abgeleitete Klasse besitzt alle Methoden der Klasse `Vogel`.

Da ein Strauß nicht fliegen kann, wird die Methode `fliegen` überschrieben und wirft eine Ausnahme.

```
public abstract class Vogel {
    public void essen() {
    }

    public void fliegen() {
    }
}

public class Ente extends Vogel {
}

public class Strauss extends Vogel {
    @Override
    public void fliegen() {
        throw new UnsupportedOperationException("nicht implementiert");
    }
}
```

Die Klasse `Strauss` verstößt gegen das *Liskovsches Substitutionsprinzip*, da sie nicht ohne Weiteres den Typ `Vogel` in einem Programm ersetzen kann, weil sie dann ein unerwartetes Verhalten zeigen würde. Verbessern Sie den Entwurf.

Tipp: Fügen Sie die abstrakte Klasse `FliegenderVogel` ein.

*Lösung:* Unterprojekt LSP

#### Aufgabe 4

Es soll mit Swing ein Fenster realisiert werden, das beim Schließen einen Text ausgibt. Hierzu wird der `WindowListener` implementiert. Es müssen alle sieben Listener-Methoden bis auf `windowClosing` mit leerem Rumpf implementiert werden. Die Klasse verstößt also gegen das *Interface-Segregation-Prinzip*.

Verbessern Sie den Code, indem Sie den `WindowAdapter` einsetzen.

*Lösung:* Unterprojekt ISP

#### Aufgabe 5

Ein E-Book-Reader kann ein PDF-Dokument lesen:

```
public class PDFBook {
    public void read() {
        System.out.println("reading a pdf book");
    }
}

public class EBookReader {
    private PDFBook pdfBook;

    public EBookReader(PDFBook pdfBook) {
        this.pdfBook = pdfBook;
    }
}
```

```
public void read() {  
    pdfBook.read();  
}  
}
```

Dieser Reader ist eng an das PDF-Format gekoppelt. Modellieren Sie einen "generischen" Reader, der mehrere Formate lesen kann: PDF, EPUB, MOBI. Wenden Sie hierzu das *Dependency-Inversion-Prinzip* an.

*Lösung: Unterprojekt DIP*



## 33 Einführung in das Modulsystem

Mit der Java-Version 9 wurde die Modularisierung als Spracherweiterung eingeführt. Sie ermöglicht es, Programme in Module zu unterteilen (eine weitere Gliederungsmöglichkeit oberhalb der package-Ebene) und Abhängigkeiten zwischen den Modulen zu definieren.

### Lernziele

In diesem Kapitel lernen Sie

- die wesentlichen Konzepte des Modulsystems kennen,
- wie Abhängigkeiten zwischen Modulen definiert werden,
- welche Vorteile die Modularisierung hat,
- wie modularisierte Anwendungen compiliert und ausgeführt werden.

### 33.1 Motivation

Welche Unzulänglichkeiten bestehen ohne Nutzung des Modulsystems?

In einem Paket sind `public` definierte Typen (Klassen, Interfaces) aus *allen* anderen Paketen zugreifbar. Die Möglichkeit eines differenzierten Zugriffsschutzes, wie er innerhalb von Klassen und Paketen mittels `public`, `protected`, `private` besteht, ist zwischen Paketen nicht mehr gegeben.

Aus einer Klasse kann damit auf alle `public`-Elemente der im Klassenpfad eingebundenen jar-Dateien zugegriffen werden.

Häufig nutzt man Fremdbibliotheken, deren Klassen und Interfaces nur im Bytecode vorliegen und eine gewünschte Funktionalität zur Verfügung stellen (z. B. einen JDBC-Treiber oder die Implementierung eines Frameworks). Um eine Anwendung compilieren zu können und zum Laufen zu bringen, müssen alle benötigten jar-Dateien über den `CLASSPATH` bekannt gegeben werden.

Somit bestehen diverse Abhängigkeiten der eigenen Anwendung von fremden Typen. Befindet sich die gleiche zu nutzende Klasse mit abweichendem Stand in unterschiedlichen jar-Dateien, so bestimmt deren Reihenfolge im `CLASSPATH`, welche Klasse geladen wird.

Oft ist es schwierig herauszufinden, welche Typen, verstreut über verschiedene jar-Dateien, zum Compilieren oder gar nur zur Laufzeit benötigt werden. Man spricht in diesem Zusammenhang von der *jar-Hölle*. Um sicher zu gehen, werden dann oft alle jar-Dateien eines Tools mit ausgeliefert, obwohl man vielleicht nur einige wenige braucht.

---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_33](https://doi.org/10.1007/978-3-658-43574-5_33).

## Fazit

- Abhängigkeiten zwischen Klassen und Interfaces in jar-Dateien sind oft undurchsichtig und schwierig zu analysieren.
- Abhängigkeiten sind nicht explizit definiert.
- Es fehlt eine Abhängigkeitskontrolle, um eine bedarfsgerechte Auslieferung von Softwarekomponenten zu ermöglichen.

## 33.2 Grundlagen

### Der Modulbegriff

Unter einem *Modul* wird hier eine Komponente verstanden,

- die über einen Namen verfügt,
- aus einem oder mehreren Paketen besteht,
- ihre Abhängigkeiten von anderen Modulen beschreibt und
- spezifiziert, welche eigenen Pakete für andere Module sichtbar sind und genutzt werden können.

### Vorteile des Modulsystems

- *Strenge Kapselung*  
Ein Modul bestimmt, welche seiner Pakete von außen sichtbar sein sollen.
- *Zuverlässige Konfiguration*  
Die definierten Abhängigkeiten werden beim Compilieren und zur Laufzeit der Anwendung geprüft. Im Kontext der Compilierung und Ausführung darf ein Paket immer nur in genau einem Modul liegen. Sogenannte *Split-Packages* sind also nicht erlaubt.
- *Skalierbare Plattform*  
Anwendungen können zusammen mit betriebssystemspezifischen, minimalen Runtime-Images ausgeliefert werden, sodass keine installierte JRE vorausgesetzt werden muss. Hierzu dient das Tool *jlink*.

Diese Skalierung ist möglich, weil das JDK selbst in eine Vielzahl von Modulen aufgeteilt ist. Das Kommando

```
java --list-modules
```

zeigt die Module des JDK.

### Modulnamen

Der Name eines Moduls muss im Ausführungskontext eindeutig sein. Hierzu sollten Namen nach dem *Reverse Domain Name Pattern* gebildet werden (als z. B. `de.firma.xyz`).

#### Wichtiger Hinweis:

Zur besseren Übersicht und um eine Verwechslung zwischen Modulen und Paketen zu vermeiden, werden in den folgenden Beispielen abweichend von dieser Namenskonvention nur einfache Namen für Module verwendet.

### Verzeichnisstruktur eines Projekts

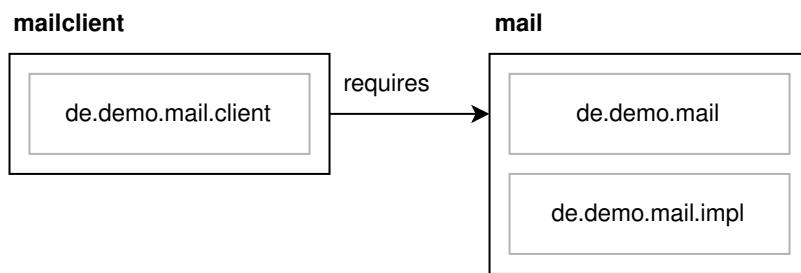
Wir orientieren uns an der von der IDE *IntelliJ IDEA* vorgegebenen Verzeichnisstruktur für die Ablage der diversen Artefakte. Alle Beispiele können auch auf der Konsole (Terminal) nachvollzogen werden.

### Kompatibilitätsmodus

Klassische, nicht modularisierte Java-Anwendungen können wie vor Java-Version 9 unter Verwendung von CLASSPATH compiliert und ausgeführt werden.

## 33.3 Abhängigkeiten und Zugriffsschutz

Die Beispielanwendung besteht aus den beiden Modulen `mail` und `mailclient`.



```
module-info.java:  
module mailclient {  
    requires mail;  
}
```

```
module-info.java:  
module mail {  
    exports de.demo.mail;  
}
```

**Abbildung 33-1:** Modul `mailclient` nutzt Modul `mail`

Das Modul `mail` enthält

- das Interface `de.demo.mail.MailSender`,
- die Implementierung `de.demo.mail.impl.MailSenderImpl` und
- die sogenannte Factory `de.demo.mail.MailSenderFactory` zur Erzeugung einer Instanz vom Typ `MailSender`.

Die Klasse `de.demo.mail.client.MailClient` im Modul `mailclient` verwendet das Interface `MailSender` und die Factory `MailSenderFactory` im Modul `mail`. Der Client hat keinen Zugriff auf die Implementierungsklasse.

Die Definition eines Moduls (*Moduldeskriptor* genannt) wird in der Datei

```
module-info.java
```

bereit gestellt (siehe Abbildung 33-1).

Das Schlüsselwort `requires` beschreibt die Abhängigkeit von einem anderen Modul, `exports` nennt das Paket, das für andere Module sichtbar ist.

Ein Paket kann auch gezielt nur für bestimmte Module mit Hilfe des Zusatzes `to` freigegeben werden:

```
exports paket to modulA, modulB, ...;
```

## Verzeichnisstruktur

### Projekt

```
+---mail
|   \---src
|       |   module-info.java
|       |
|       \---de
|           \---demo
|               \---mail
|                   |   MailSender.java
|                   |   MailSenderFactory.java
|                   |
|                   \---impl
|                           MailSenderImpl.java
\---mailclient
    \---src
        |   module-info.java
        |
        \---de
            \---demo
                \---mail
                    \---client
                        MailClient.java
```

Der Moduldeskriptor liegt im Quellcode-Verzeichnis des jeweiligen Moduls.

Im Folgenden ist der Quellcode abgedruckt (Projekt *kap33-1*).

```
package de.demo.mail;

public interface MailSender {
    boolean sendMail(String to, String text);
}

package de.demo.mail.impl;

import de.demo.mail.MailSender;

public class MailSenderImpl implements MailSender {
    public boolean sendMail(String to, String text) {
        System.out.println("Mail an " + to + ":" + text);
        return true;
    }
}

package de.demo.mail;

import de.demo.mail.impl.MailSenderImpl;

public class MailSenderFactory {
    public static MailSender create() {
        return new MailSenderImpl();
    }
}

package de.demo.mail.client;

import de.demo.mail.MailSender;
import de.demo.mail.MailSenderFactory;

public class MailClient {
    public static void main(String[] args) {
        MailSender mailSender = MailSenderFactory.create();
        boolean ok = mailSender.sendMail("hugo.meier@abc.de", "Das ist ein Test.");
        if (ok) {
            System.out.println("Mail wurde versandt.");
        } else {
            System.out.println("Fehler beim Senden.");
        }
    }
}
```

Die folgenden Kommandos sind jeweils in einer einzigen Zeile anzugeben und auf Projektebene, also im Verzeichnis *kap33-1* auszuführen.

Möchte man das Kommando in mehrere Zeilen aufteilen, muss der Zeilenumbruch bei Windows mit dem Zeichen ^ und bei Linux mit \ maskiert werden.

## Übersetzen

```
javac -d out/production/mail mail/src/*.java \
mail/src/de/demo/mail/*.java mail/src/de/demo/mail/impl/*.java
```

```
javac -d out/production/mailclient -p out/production \
mailclient/src/*.java mailclient/src/de/demo/mail/client/*.java
```

Die Option -p legt die Modulpfade (die Verzeichnisse, die Module enthalten) fest .

Die Bytecodes werden im Verzeichnis

```
out/production/mail bzw. out/production/mailclient  
gespeichert.
```

## Ausführen

```
java -p out/production -m mailclient/de.demo.mail.client.MailClient
```

Die Option -m legt die Main-Klasse fest.

Ausgabe:

```
Mail an hugo.meier@abc.de:  
Das ist ein Test.  
Mail wurde versandt.
```

Es können auch jar-Dateien erzeugt werden:

```
jar --create --file lib/mailclient.jar -C out/production/mailclient .
jar --create --file lib/mail.jar -C out/production/mail .
```

Das Verzeichnis lib liegt im Verzeichnis *kap33-1*. Damit kann das Programm wie folgt gestartet werden:

```
java -p lib -m mailclient/de.demo.mail.client.MailClient
```

Die Main-Klasse kann auch in der jar-Datei festgelegt werden:

```
jar --create --file lib/mailclient.jar \
--main-class de.demo.mail.client.MailClient -C out/production/mailclient .
```

Das Programm kann damit wie folgt gestartet werden:

```
java -p lib -m mailclient
```

### Abhängigkeiten analysieren

Mit dem Kommando

```
jdeps -s lib/*.jar
```

werden die Abhängigkeiten zwischen den Modulen ermittelt:

```
mail -> java.base  
mailclient -> java.base  
mailclient -> mail
```

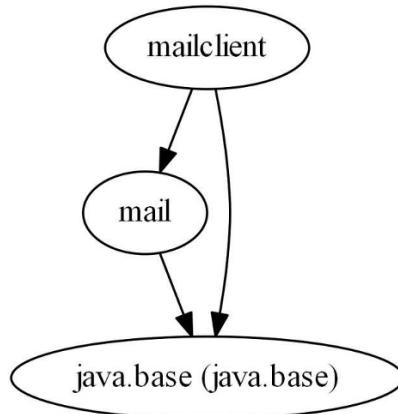
Mit dem Tool *graphviz*<sup>1</sup> können diese Informationen grafisch aufbereitet werden.

Zunächst wird die Datei `summary.dot` im Verzeichnis `graphs` (Unterverzeichnis von `kap33-1`) erzeugt:

```
jdeps -s -dotoutput graphs lib/*.jar
```

Nach der Installation von *graphviz* steht das Programm `dot` zur Verfügung, das den Abhängigkeitsgraphen im Bild `summary.dot.png` erzeugt:

```
dot -Tpng -Gdpi=300 graphs/summary.dot -O
```



**Abbildung 33-2:** Abhängigkeitsgraph (`summary.dot.png`)<sup>2</sup>

1 Siehe <https://graphviz.gitlab.io>

2 Das JDK-Modul `java.base` mit grundlegenden Paketen (`java.lang`, `java.io` usw.) hängt von keinem Modul ab, alle anderen Module sind von ihm abhängig. `java.base` muss nicht explizit per `requires` angefordert werden.

## Zusammenfassung

Ein Modul hat einen Modulnamen und besteht aus Paketen, die Interfaces und Implementierungen enthalten können.

Es können verschiedene Fähigkeiten spezifiziert werden:

### exports

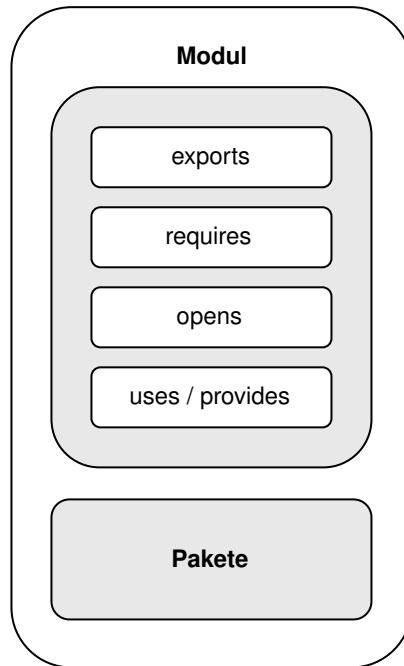
Ein Paket wird für alle oder nur für bestimmte Module zur Verfügung gestellt:

```
exports paket;  
exports paket to modulA, modulB, ...;
```

### requires

Ein anderes Modul benötigt:

```
requires module;
```



**Abbildung 33-3:** Bestandteile eines Moduls

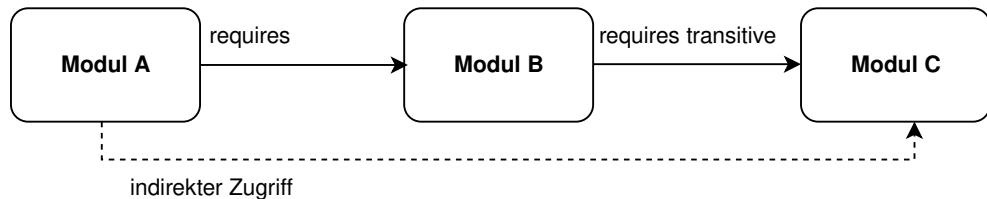
Die weiteren Schlüsselwörter in Abbildung 33-3 werden in den folgenden Kapiteln erklärt.

## 33.4 Transitive Abhangigkeiten

Das Modul B kann dem Modul A das Modul C indirekt zur Nutzung über

requires transitive

zur Verfügung stellen.



**Abbildung 33-4:** Transitivitat von Abhangigkeiten

Die Moduldeskriptoren sehen z. B. wie folgt aus:

```
module A {
    requires B;
}

module B {
    requires transitive C;
    exports de.demo.b;
}

module C {
    exports de.demo.c;
}
```

Würde in Modul B "requires C" statt "requires transitive C" stehen, dann müsste Modul A explizit noch "requires C" enthalten.

### 33.5 Abhangigkeit von JDK-Modulen und anderen Modulen

Während `java.base` nicht explizit per `requires` im Moduldeskriptor angefordert werden muss, ist das jedoch bei anderen Modulen des JDK erforderlich.

Im Projekt *kap33-2* hat der Mailclient eine grafische Oberfläche. Diese ist mit Swing realisiert. Das Modul `mail` ändert sich nicht.

```
package de.demo.mail.client;

import de.demo.mail.MailSender;
import de.demo.mail.MailSenderFactory;

import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.FlowLayout;

public class MailClient {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        JLabel label = new JLabel();

        MailSender mailSender = MailSenderFactory.create();
        boolean ok = mailSender.sendMail("hugo.meier@abc.de",
                                         "Das ist ein Test.");
        if (ok) {
            label.setText("Mail wurde versandt.");
        } else {
            label.setText("Fehler beim Senden.");
        }

        frame.setLayout(new FlowLayout());
        frame.getContentPane().add(label);
        frame.setSize(400, 100);
        frame.setTitle("MailClient (Swing)");
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Die beiden verwendeten Pakete `javax.swing` und `java.awt` liegen im Modul `java.desktop`. Deshalb muss der Moduldeskriptor neben `requires mail` auch `requires java.desktop` enthalten:

```
module mailclient {
    requires mail;
    requires java.desktop;
}
```

Um das für eine Klasse oder ein Interface zugehörige Modul zu ermitteln, kann die API-Dokumentation von Java (Javadoc) benutzt werden. Ein Eingabefeld ermöglicht die Suche.

Im Modul `mailclient.javafx` des Projekts *kap33-2* ist die Oberfläche mit *JavaFX* realisiert. JavaFX ist ab Version 11 nicht mehr Bestandteil von Java und muss deshalb als Fremdbibliothek eingebunden werden. Die jar-Dateien von JavaFX sind modular, d. h. sie enthalten Moduldeskriptoren.

```
package de.demo.mail.client2;

import de.demo.mail.MailSender;
import de.demo.mail.MailSenderFactory;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class MailClient extends Application {
    @Override
    public void start(Stage stage) {
        Label label = new Label();

        MailSender mailSender = MailSenderFactory.create();
        boolean ok = mailSender.sendMail("hugo.meier@abc.de", "Das ist ein Test.");
        if (ok) {
            label.setText("Mail wurde versandt.");
        } else {
            label.setText("Fehler beim Senden.");
        }

        StackPane pane = new StackPane();
        pane.getChildren().add(label);
        stage.setScene(new Scene(pane, 400, 100));
        stage.setTitle("MailClient (JavaFX)");
        stage.setResizable(false);
        stage.show();
    }

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

Anstelle von `requires java.desktop` enthält der zugehörige Moduldeskriptor:

```
    requires javafx.controls
```

Es wird fehlerfrei compiliert.

Allerdings kann die Anwendung nicht gestartet werden. Fehlermeldung:

```
class com.sun.javafx.application.LauncherImpl (in module javafx.graphics) cannot
access class de.demo.mail.client2.MailClient
```

Offensichtlich muss die Klasse `LauncherImpl` zur Laufzeit per *Reflection*<sup>3</sup> auf Elemente von `MailClient` zugreifen.

---

3 Mit *Reflection* wird eine Technologie beschrieben, die es ermöglicht, zur Laufzeit unbekannte Objekte zu untersuchen und zu manipulieren.

Im Moduldeskriptor können nach dem Schlüsselwort `opens` aufgeführte Pakete für den Zugriff per Reflection zur Laufzeit geöffnet werden.

Das kann in einer Variante auch nur für bestimmte Module geschehen. Wir ergänzen also den Deskriptor um:

```
opens de.demo.mail.client2 to javafx.graphics;
```

`opens` erlaubt keine Zugriffe zur Compilierzeit, sondern nur zur Laufzeit.

Nach erneuter Übersetzung kann das Programm nun fehlerfrei gestartet werden.

Da `javafx.controls` seine Abhängigkeit von `javafx.graphics` mittels `requires` transitive weiterreicht (siehe Kapitel 33.4), muss `javafx.graphics` nicht explizit aufgenommen werden:

```
module mailclient.javafx {
    requires mail;
    requires javafx.controls;
    opens de.demo.mail.client2 to javafx.graphics;
}
```

## Zusammenfassung

### opens

Ein Paket wird zur Laufzeit für den externen Zugriff per Reflection geöffnet.

```
opens paket;
opens paket to modulA, modulB, ...;
```

Alternative können mit dem Kommandozeilenparameter `--add-opens` auch bei der Ausführung Pakete geöffnet werden.

Passend zum obigen Beispiel:

```
java --add-opens mailclient.javafx/de.demo.mail.client2=javafx.graphics \
-p out/production:$PATH_TO_FX \
-m mailclient.javafx/de.demo.mail.client2.MailClient4
```

`mailclient.javafx/de.demo.mail.client2=javafx.graphics` bedeutet:

Das Modul `mailclient.javafx` öffnet sein Paket `de.demo.mail.client2` für das Modul `javafx.graphics`.

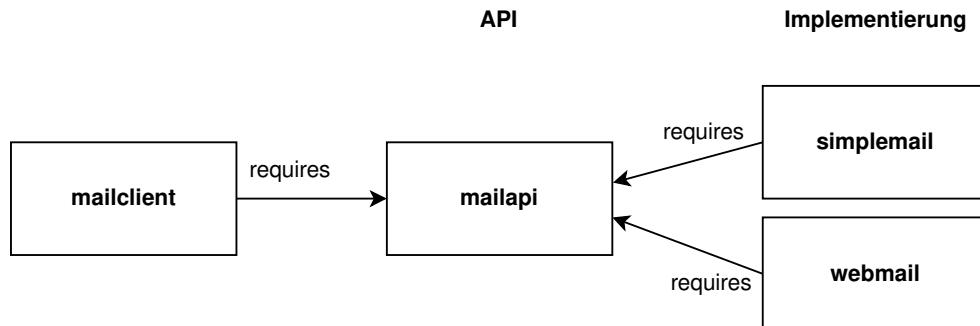
---

<sup>4</sup> `PATH_TO_FX` siehe Kapitel 28.1

## 33.6 Trennung von Schnittstelle und Implementierung

Im Projekt *kap33\_1* ist das Interface `MailSender` des Moduls `mail` eng an die Implementierung gekoppelt: Beide liegen im selben Modul.

Damit die Implementierung leicht gegen eine andere ausgetauscht werden kann, sollten Schnittstelle und Implementierung in separaten Modulen liegen (siehe Projekt *kap33-3*).



**Abbildung 33-5:** Trennung von Schnittstelle und Implementierung

Das Modul `mailapi` enthält das Interface `de.demo.mail.MailSender`.

```

package de.demo.mail;

public interface MailSender {
    boolean sendMail(String to, String text);
}
  
```

Die Klasse `de.demo.mail.simple.impl.SimpleMailSenderImpl` liegt im Modul `simplemail`, die Klasse `de.demo.mail.web.impl.WebMailSenderImpl` im Modul `webmail`.

```

package de.demo.mail.simple.impl;

import de.demo.mail.MailSender;

public class SimpleMailSenderImpl implements MailSender {
    @Override
    public boolean sendMail(String to, String text) {
        System.out.println("SimpleMail an " + to + ":" + text);
        return true;
    }
}
  
```

```
package de.demo.mail.web.impl;

import de.demo.mail.MailSender;

public class WebMailSenderImpl implements MailSender {
    @Override
    public boolean sendMail(String to, String text) {
        System.out.println("WebMail an " + to + ":\n" + text);
        return true;
    }
}
```

### Zyklische Abhängigkeit

Die jeweiligen Factories `SimpleMailSenderFactory` und `WebMailSenderFactory` sollten im Modul `mailapi` liegen, damit der Client diese nutzen kann.

Das führt allerdings zu einem Übersetzungsfehler, da beispielsweise die Factory `SimpleMailSenderFactory` dann eine Abhängigkeit zum Modul `simplemail` und dieses wiederum eine Abhängigkeit zum Modul `mailapi` hat.

Solche *zyklischen Abhängigkeiten* zwischen Modulen sind nicht erlaubt.

Wir lösen dieses Problem, indem wir die Factories in eigenen Paketen innerhalb von `simplemail` und `webmail` unterbringen.

```
package de.demo.mail.simple;

import de.demo.mail.MailSender;
import de.demo.mail.simple.impl.SimpleMailSenderImpl;

public class SimpleMailSenderFactory {
    public static MailSender create() {
        return new SimpleMailSenderImpl();
    }
}

package de.demo.mail.web;

import de.demo.mail.MailSender;
import de.demo.mail.web.impl.WebMailSenderImpl;

public class WebMailSenderFactory {
    public static MailSender create() {
        return new WebMailSenderImpl();
    }
}
```

`SimpleMailClient` und `WebMailClient` liegen im Paket `de.demo.mail.client` des Moduls `mailclient`.

```
package de.demo.mail.client;

import de.demo.mail.MailSender;
import de.demo.mail.simple.SimpleMailSenderFactory;

public class SimpleMailClient {
    public static void main(String[] args) {
        MailSender mailSender = SimpleMailSenderFactory.create();
        boolean ok = mailSender.sendMail("hugo.meier@abc.de", "Das ist ein Test.");
        if (ok) {
            System.out.println("Mail wurde versandt.");
        } else {
            System.out.println("Fehler beim Senden.");
        }
    }
}

package de.demo.mail.client;

import de.demo.mail.MailSender;
import de.demo.mail.web.WebMailSenderFactory;

public class WebMailClient {
    public static void main(String[] args) {
        MailSender mailSender = WebMailSenderFactory.create();
        boolean ok = mailSender.sendMail("hugo.meier@abc.de", "Das ist ein Test.");
        if (ok) {
            System.out.println("Mail wurde versandt.");
        } else {
            System.out.println("Fehler beim Senden.");
        }
    }
}
```

Es folgen die jeweiligen Moduldeskriptoren:

```
module mailclient {
    requires mailapi;
    requires simplemail;
    requires webmail;
}

module mailapi {
    exports de.demo.mail;
}

module simplemail {
    requires mailapi;
    exports de.demo.mail.simple;
}

module webmail {
    requires mailapi;
    exports de.demo.mail.web;
}
```

Beachten Sie, dass die Implementierungsklassen von außen nicht sichtbar sind.

Übersetzung und Ausführung erfolgen analog zu den früheren Beispielen. Das Begleitmaterial zu diesem Buch enthält die nötigen Anweisungen zur manuellen Übersetzung und Ausführung.

### 33.7 Modularisierung und Services

Wir zeigen hier im Projekt *kap33-4* eine elegantere Lösung für das Problem in Kapitel 33.6.



**Abbildung 33-6:** Bindung von Modulen zur Startzeit

Mit `uses` und `provides` werden Abhängigkeiten nicht statisch zur Übersetzungszeit, sondern erst zur Startzeit der Anwendung aufgelöst.

Gegenüber dem vorhergehenden Projekt werden keine Factories verwendet.

Die Service-Module `simplemail` und `webmail` enthalten nur das Paket `de.demo.mail.simple` bzw. `de.demo.mail.web` mit den Klassen `SimpleMailSenderImpl` bzw. `WebMailSenderImpl`.

Das Modul `mailapi` enthält nach wie vor das Interface `de.demo.mail.Mailsender`.

Statt der beiden Clients `SimpleMailClient` und `WebMailClient` reicht nur ein Client. Dieser verwendet den aus Kapitel 17 bekannten `ServiceLoader`.

Beim Aufruf des Programms wird durch Angabe des Modulpfads der gewünschte Service bestimmt.

```

package de.demo.mail.client;

import de.demo.mail.Mailsender;

import java.util.Optional;
import java.util.ServiceLoader;

public class MailClient {
    public static void main(String[] args) {
        Optional<Mailsender> optional = ServiceLoader
            .load(Mailsender.class)
            .findFirst();
    }
}
  
```

```

        if (optional.isPresent()) {
            boolean ok = optional.get().sendMail("hugo.meier@abc.de",
                "Das ist ein Test.");
            if (ok) {
                System.out.println("Mail wurde versandt.");
            } else {
                System.out.println("Fehler beim Senden.");
            }
        }
    }
}

```

### Die ServiceLoader-Methode

`Optional<S> findFirst()`

lädt den ersten verfügbaren *Service Provider* oder ein "leeres" optional, wenn kein Provider gefunden wurde.

### Deskriptor des Moduls mailapi:

```

module mailapi {
    exports de.demo.mail;
}

```

### Deskriptor des Moduls simplemail:

```

module simplemail {
    requires mailapi;
    provides de.demo.mail.MailSender with de.demo.mail.simple.SimpleMailSenderImpl;
}

```

Die Schlüsselwörter `provides` und `with` beschreiben, mit welcher Implementierung ein Service Provider das Interface realisiert. Es erfolgt jedoch kein Export des Pakets `de.demo.mail.simple`.

### Deskriptor des Moduls webmail:

```

module webmail {
    requires mailapi;
    provides de.demo.mail.MailSender with de.demo.mail.web.WebMailSenderImpl;
}

```

### Deskriptor des Moduls mailclient:

```

module mailclient {
    requires mailapi;
    uses de.demo.mail.MailSender;
}

```

`uses` drückt hier die Abhängigkeit vom Service-Interface aus.

**Übersetzung:**

```
javac -d out/production/mailapi \
mailapi/src/*.java mailapi/src/de/demo/mail/*.java

javac -d out/production/simplemail -p out/production \
simplemail/src/*.java simplemail/src/de/demo/mail/simple/*.java

javac -d out/production/webmail -p out/production \
webmail/src/*.java webmail/src/de/demo/mail/web/*.java

javac -d out/production/mailclient -p out/production \
mailclient/src/*.java mailclient/src/de/demo/mail/client/*.java
```

**Erzeugung der jar-Dateien (Ablage getrennt nach Modulen):**

```
jar --create --file mailclient/lib/mailclient.jar \
-C out/production/mailclient .

jar --create --file mailapi/lib/mailapi.jar \
-C out/production/mailapi .

jar --create --file simplemail/lib/simplemail.jar \
-C out/production/simplemail .

jar --create --file webmail/lib/webmail.jar \
-C out/production/webmail .
```

**Ausführung:**

```
java -p mailclient/lib:mailapi/lib:simplemail/lib \
-m mailclient/de.demo.mail.client.MailClient

java -p mailclient/lib:mailapi/lib:webmail/lib \
-m mailclient/de.demo.mail.client.MailClient
```

***Hinweis***

Im Client-Programmen wird der erste zum Service-Interface `MailSender` gefundene Service Provider geliefert. Deshalb liegen die jar-Dateien der Implementierungen in getrennten Verzeichnissen, sodass immer der passende Provider gefunden wird.

**Zusammenfassung****uses**

Es wird das zu nutzende Service-Interface angegeben:

```
uses interface;
```

**provides**

Hiermit wird angegeben, welche Implementierung das Service-Interface realisiert:

```
provides interface with klasse;  
provides interface with klasse1, klasse2 ...;
```

Mehrere Implementierungen werden durch Kommas getrennt.

## 33.8 Einbindung nicht-modularer Bibliotheken

Befindet sich eine jar-Datei ohne Moduldeskriptor (eine sogenannte nicht-modulare jar-Datei) im Modulpfad, wird zur Laufzeit ein Moduldeskriptor automatisch erzeugt.

Es liegt dann ein sogenanntes *Automatic Module* vor. Alle seine Pakete werden exportiert und es kann selbst alle exportierten Pakete der anderen expliziten Module benutzen.

Der Name eines solchen Moduls wird nach einer bestimmten Regel gebildet. Hat der Name der jar-Datei die Form `xxx-n.n.jar`, so verwendet das Modulsystem den Basisnamen `xxx` ohne Versionsnummer und Endung als Modulnamen. Sind Minuszeichen im Basisnamen vorhanden, müssen diese jeweils durch einen Punkt im Moduldeskriptor ersetzt werden.

Der Modulname kann aber auch über einen Eintrag in der Manifest-Datei der jar-Datei wie folgt festgelegt werden:

```
Automatic-Module-Name: Modulname
```

Im folgenden Beispiel (Projekt `kap33-5`) wird eine nicht-modulare Bibliothek nur zur Laufzeit benutzt, sodass diese in den Moduldeskriptoren nicht berücksichtigt werden muss.

Zur Demonstration greifen wir ein Beispiel aus Kapitel 29 auf und erzeugen eine SQLite-Datenbank. Der dazu benötigte JDBC-Treiber befindet sich in der nicht-modularen jar-Datei `lib/sqlite-jdbc-3.42.0.0.jar`<sup>5</sup>. Das Modul `db` enthält das Paket `de.demo.db` mit den Klassen `CreateDB` und `Query`.

```
package de.demo.db;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
import java.sql.Statement;
```

<sup>5</sup> <https://github.com/xerial/sqlite-jdbc/releases>

```
public class CreateDB {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        try (Connection con = DriverManager.getConnection(url);
             Statement stmt = con.createStatement()) {

            String sql = """
                create table if not exists artikel (
                    id integer primary key,
                    name text,
                    preis real,
                    menge integer)
                """;
            stmt.executeUpdate(sql);

            sql = """
                insert into artikel (id, name, preis, menge)
                values (4711, 'Hammer', 3.9, 10)
                """;
            stmt.executeUpdate(sql);
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}

package de.demo.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Query {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:mydb.db";
        try (Connection con = DriverManager.getConnection(url);
             Statement stmt = con.createStatement()) {

            String sql = """
                select id, name, menge, preis from artikel
                order by preis desc
                """;
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                System.out.printf("%4d %-15s %4d %8.2f%n",
                    rs.getInt(1),
                    rs.getString(2),
                    rs.getInt(3),
                    rs.getDouble(4));
            }
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Moduldeskriptor:

```
module db {  
    requires java.sql;  
}
```

Übersetzung und Ausführung:

```
javac -d out/production/db db/src/*.java db/src/de/demo/db/*.java  
jar --create --file lib/db.jar -C out/production/db .  
java -p lib -m db/de.demo.db.CreateDB  
java -p lib -m db/de.demo.db.Query
```

Im Modul `text` wird die jar-Datei `commons-text-1.11.0.jar`<sup>6</sup> benutzt. Die Manifest-Datei hat den Eintrag

```
Automatic-Module-Name: org.apache.commons.text
```

Im folgenden Programm wird ein Text codiert und wieder decodiert.

```
package de.demo.text;  
  
import org.apache.commons.text.AlphabetConverter;  
  
import java.io.UnsupportedEncodingException;  
  
public class Coding {  
    public static void main(String[] args) throws UnsupportedEncodingException {  
  
        Character[] originals = new Character[]  
            {'a', 'b', 'c', 'd', 'e', 'f', ' '};  
        Character[] encoding = new Character[]  
            {'1', '2', '3', '4', '5', '6', '-'};  
  
        AlphabetConverter converter = AlphabetConverter  
            .createConverterFromChars(originals, encoding, null);  
  
        String encoded = converter.encode("f e d c b a");  
        System.out.println(encoded);  
  
        String decoded = converter.decode(encoded);  
        System.out.println(decoded);  
    }  
}
```

---

6 <https://commons.apache.org/proper/commons-text>

Die jar-Datei selbst verwendet noch `commons-lang3-3.13.0.jar`, siehe  
<https://commons.apache.org/proper/commons-lang>

Moduldeskriptor:

```
module text {
    requires org.apache.commons.text;
}
```

Übersetzung und Ausführung:

```
javac -d out/production/text -p lib \
text/src/*.java text/src/de/demo/text/*.java

jar --create --file lib/text.jar -C out/production/text .

java -p lib -m text/de.demo.text.Coding
```

## Zusammenfassung

jar-Dateien, die keinen Moduldeskriptor besitzen, werden als sogenannte *Automatic Modules* behandelt, wenn sie in den Modulpfad gelegt werden.

Damit sind sie ein wichtiges Hilfsmittel für die Umstellung (Migration) auf modularisierte Anwendungen, wenn man fremde Bibliotheken nutzt, die nicht modularisiert sind.

Der Name eines *Automatic Modules* wird wie oben beschrieben bestimmt (automatische Namenszuweisung).

Der Modulname kann aber auch über einen Eintrag in der Manifest-Datei der jar-Datei festgelegt werden:

Automatic-Module-Name: *Modulname*

## 33.9 Modulkategorien

Dieser Abschnitt stellt die verschiedenen Modulararten zusammenfassend dar. Dabei werden auch die Beziehungen untereinander und evtl. Einschränkungen erwähnt.

### Named Modules

Hierzu gehören:

- *Explicit Modules*

Das sind die vollwertigen Module mit Moduldeskriptor (siehe Abbildung 33-3).

- *Open Modules*

Wie explizite Module. Zusätzlich werden zur Laufzeit *alle* Pakete für *Reflection* freigegeben. Im Moduldeskriptor beginnt die erste Zeile mit `open module`.

- *Automatic Modules*

jar-Dateien ohne Moduldeskriptor, die im Modulpfad liegen (siehe Kapitel 33.8). Ein *Automatic Module* kann die exportierten Pakete aller *Explicit Modules* und *Open Modules* benutzen. Es exportiert automatisch alle seine Pakete, auch für *Reflection*.

### Unnamed Modules

Alle Inhalte des CLASSPATH werden automatisch zu einem sogenannten *Unnamed Module* zusammengefasst.

- Ein *Unnamed Module* kann auf die exportierten Pakete aller Module im Modulpfad zugreifen.
- Es exportiert alle seine Pakete.
- *Explicit Modules* und *Open Modules* haben keinen Zugriff auf ein solches *Unnamed Module*.
- Nur *Automatic Modules* haben Zugriff auf Pakete des *Unnamed Module*.

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten geordnet nach Modulkategorie.

**Tabelle 33-1:** Modulkategorie und Zugriffsmöglichkeiten

Modulkategorie	Exportiert ...	Zugriff auf exportierte Pakete in ...
Explicit Module	per exports ...	Explicit, Open, Automatic Modules
Open Module	per exports ...	Explicit, Open, Automatic Modules
Automatic Module	alle Pakete	Explicit, Open, Automatic Modules, Unnamed Module
Unnamed Module	alle Pakete	Explicit, Open, Automatic Modules

## 33.10 Aufgaben

### Aufgabe 1

Gegeben sei das folgende Programm:

```
public class Quoter {
    public String quote(String text) {
        return "\"" + text + "\"";
    }

    public static void main(String[] args) {
        Quoter quoter = new Quoter();
        System.out.println(quoter.quote("Das ist ein Test."));
    }
}
```

- a) Teilen Sie das Programm in die beiden Module

quoterclient und quoterserver

auf mit den Klassen

de.quoter.app.QuoterApp bzw. de.quoter.services.QuoterUtils

und den zugehörigen Moduldeskriptoren.

- b) Übersetzen und starten Sie die Anwendung. Orientieren Sie sich hierzu an den Kommandos im Projekt *kap33-1*.
- c) Erzeugen Sie zwei modulare jar-Dateien und starten Sie die Anwendung mit Hilfe dieser jar-Dateien.
- d) Listen Sie die Abhängigkeiten mit Hilfe des Tools *jdeps* auf. Bereiten Sie diese auch grafisch auf (vgl. Kapitel 33.3).

*Lösung: Projekt kap33-Aufgabe-1*

### Aufgabe 2

Erweitern Sie die Klasse QuoterUtils aus Aufgabe 1 so, dass Methodenaufrufe protokolliert werden. Nutzen Sie dazu das *Logging-API*:

```
package de.quoter.services;

import java.util.logging.Logger;

public class QuoterUtils {
    private static final Logger LOGGER;

    static {
        System.setProperty("java.util.logging.config.file", "logging.properties");
        LOGGER = Logger.getLogger(QuoterUtils.class.getName());
    }

    public String quote(String text) {
```

```
    LOGGER.info("quote wird aufgerufen");
    return "\"" + text + "\"";
}
```

Die Datei *logging.properties* enthält die Einstellungen:

```
handlers = java.util.logging.FileHandler
java.util.logging.FileHandler.pattern = quoterserver.log
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.append = true
```

Beachten Sie, dass das JDK-Modul `java.logging` per `requires` angefordert werden muss.

*Lösung: Projekt kap33-Aufgabe-2*

### Aufgabe 3

Ausgehend von der Lösung in Aufgabe 2 soll das Modul `quoterclient` unabhängig vom Modul `quoterserver` werden. Führen Sie dazu ein neues Modul mit einem Interface ein und verwenden Sie das Service-Konzept aus Kapitel 33.7.

*Lösung: Projekt kap33-Aufgabe-3*

### Aufgabe 4

Ergänzen Sie Interface und Implementierung aus Aufgabe 3 um die Methode

```
String quoteHtml(String text),
```

die "kritische" Zeichen in entsprechende HTML-Entities umwandelt. So soll beispielsweise aus dem Text

```
bread & butter
```

als Ergebnis

```
&quot;bread & butter&quot;
```

erzeugt werden.

Für die Implementierung soll die Klassenmethode `escapeHtml4` der Klasse `org.apache.commons.text.StringEscapeUtils` aus der nicht-modularen jar-Datei `commons-text-1.11.0.jar` des *Apache Commons Text API* genutzt werden. Die erforderlichen jar-Dateien sind im Begleitmaterial vorhanden.

`escapeHtml4` wandelt eine als Aufrufparameter übergebene Zeichenkette entsprechend um.

Der Deskriptor des Moduls `quoterserver` muss die Zeile

```
    requires org.apache.commons.text;
```

enthalten.

*Lösung: Projekt kap33-Aufgabe-4*

**Aufgabe 5**

Realisieren Sie den Brutto-Rechner in der ersten Variante (programmatischer Zusammenbau) aus Kapitel 28 als modulare Anwendung.

*Lösung: Projekt kap33-Aufgabe-5*

**Aufgabe 6**

Demonstrieren Sie die *transitive Abhängigkeit* aus Kapitel 33.4 anhand eines einfachen Beispiels.

*Lösung: Projekt kap33-Aufgabe-6*



# 34 IntelliJ IDEA, Debugging und Testen mit JUnit

Dieses Kapitel enthält einige nützliche Hinweise zum Umgang mit der Entwicklungsumgebung IntelliJ IDEA, zum Debugging und zum Testen von Java-Code.

## 34.1 Hinweise zu IntelliJ IDEA

Diese Hinweise beziehen sich auf die Version *IntelliJ IDEA 2023.2 (Community Edition)*. In neueren Versionen können Abweichungen bzgl. der Darstellung (Layout der Oberfläche) vorkommen. Die Funktionalität bleibt jedoch erhalten.

Unter der Adresse

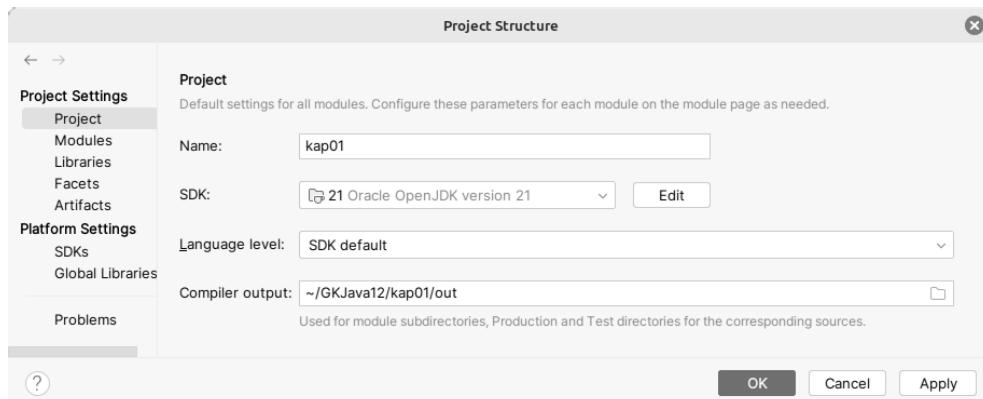
<https://www.jetbrains.com/help/idea/creating-running-and-packaging-your-first-java-application.html>

findet man ein kurzes Tutorial, das zeigt, wie ein einfaches Java-Projekt in *IntelliJ IDEA* erstellt und ausgeführt werden kann.

### Java-Version einstellen

Nach Installation einer neuen Java-Version muss diese zur Verwendung in *IntelliJ IDEA* bekannt gemacht werden. Für bestehende Projekte müssen dann Einstellungen bzgl. der neuen Java-Version vorgenommen werden.

*File > Project Structure*



- Unter *SDKs* wird der *JDK home path* eingestellt,
- unter *Project* das *SDK* und
- unter *Modules, Dependencies* das *Module SDK*.

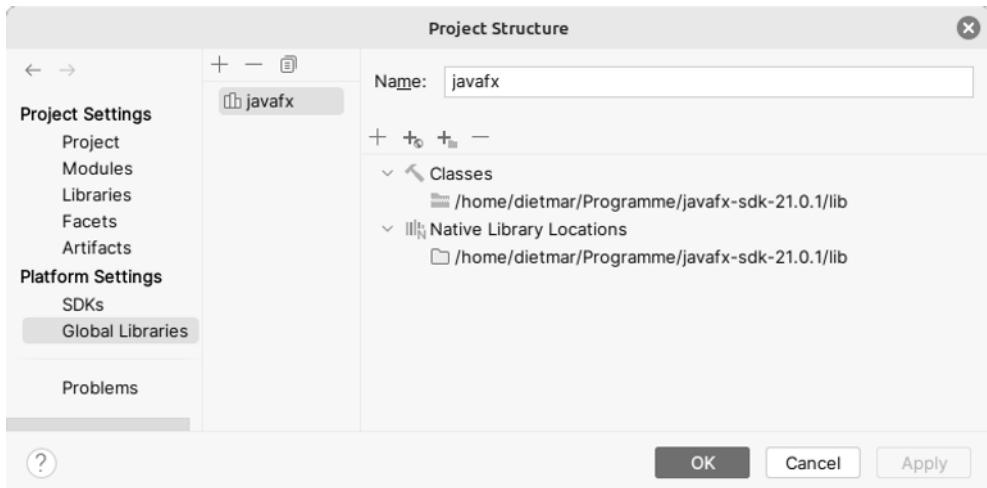
**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43574-5\\_34](https://doi.org/10.1007/978-3-658-43574-5_34).

## Globale Bibliothek erzeugen und verwenden

Mehrere jar-Dateien können zu einer Bibliothek zusammengefasst werden, die dann in verschiedenen Projekten verwendet werden kann.

Gezeigt wird, wie die jar-Dateien von *JavaFX* in der globalen Bibliothek `javafx` zusammengefasst und in einem Projekt eingebunden werden.

*File > Project Structure*

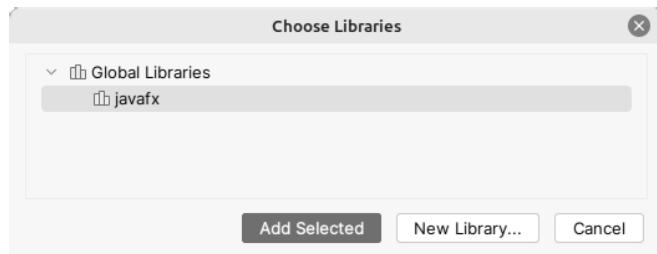


Diese globale Bibliothek kann in einem Projekt wie folgt eingebunden werden:

*Modules, Add (+)*



*Library... auswählen*

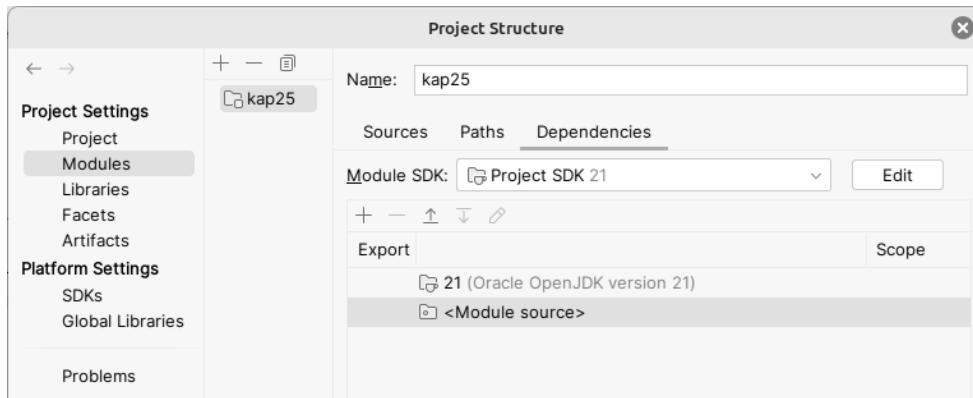


### jar-Dateien einbinden

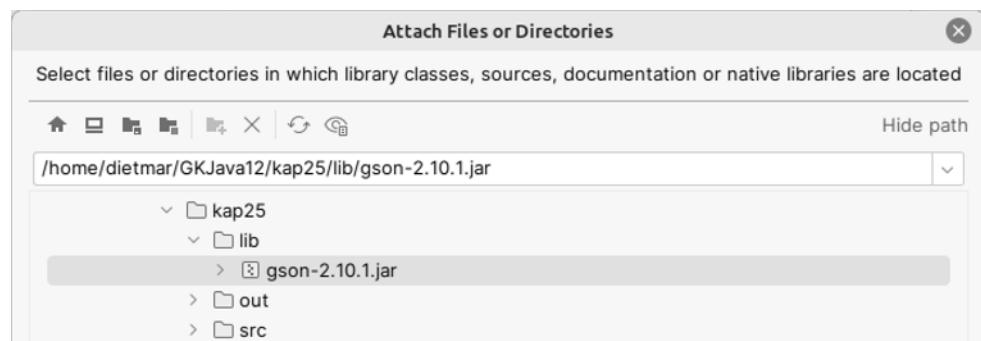
Beispiel: Im Projekt *kap25* wird die jar-Datei *gson-2.10.1.jar* benötigt.

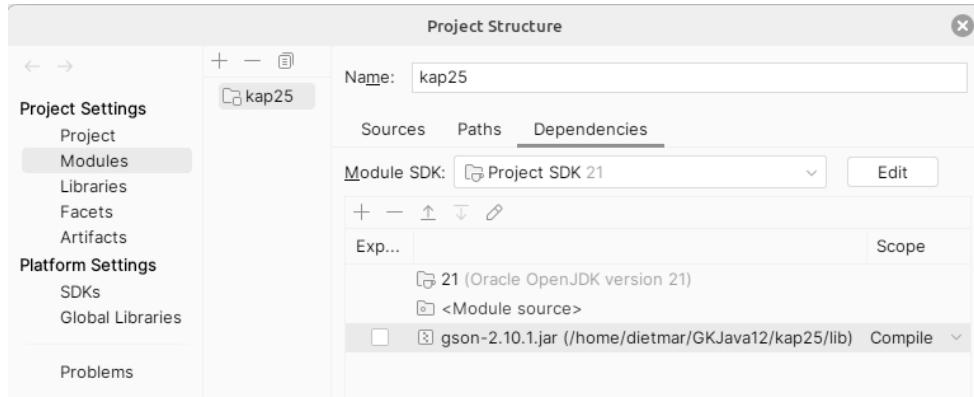
*File > Project Structure*

*Modules, Add (+)*



*JARs or Directories... auswählen*





## Modul erstellen

Ein Modul (z. B. *mail*) kann im Projektverzeichnis (hier *kap33-1*) mit

*File > New > Module...*

erstellt werden. Es entsteht das Unterverzeichnis *mail* mit *src*:

```
+---mail
|   \---src
```

Nach Auswahl von *src* kann dort mit *New module-info.java* der Deskriptor für das Modul *mail* erzeugt werden. Java-Module haben in IntelliJ IDEA die Struktur von Unterprojekten.

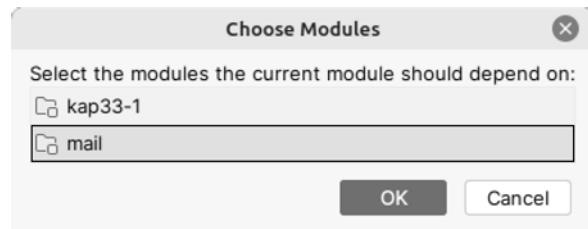
## Modulabhängigkeit einstellen

Hängt ein Modul von einem anderen Modul ab, so muss die Abhängigkeit eingestellt werden.

Beispiel: Im *kap33-1* hängt das Modul *mailclient* vom Modul *mail* ab.

*File > Project Structure*



*Add (+) Module Dependency...*

## 34.2 Debugging

Mit dem Debugger von *IntelliJ IDEA* lässt sich die Programmausführung schrittweise verfolgen. Das ist hilfreich, um die Ursache logischer Fehler eines Programms zu finden. Ein logischer Fehler liegt z. B. vor, wenn das Programm zwar compiliert und ausgeführt werden kann, aber Ergebnisse falsch berechnet werden.

Das folgende Programm soll den Durchschnitt von Zahlen berechnen:

```
public class Test {  
    public static double average(double ...n) {  
        double result = 0;  
        for (double i : n) {  
            result += i;  
        }  
        return result;  
    }  
  
    public static void main(String[] args) {  
        double[] numbers = {1, 2, 3};  
        double avg = average(numbers);  
        System.out.println(avg);  
    }  
}
```

Die Ausgabe des Programms ist jedoch 6 und nicht 2.

Wir wollen den Debugger nutzen, um die Fehlerursache zu finden.

Durch einen Mausklick auf die Zeilennummer 12 im Editor wird ein *Haltepunkt* (*Breakpoint*) gesetzt. Alternativ kann auch die Tastenkombination *Strg + F8* in dieser Zeile genutzt werden. (Übrigens: eine erneute Ausführung entfernt diesen Haltepunkt wieder.)

Der Debugger wird mit *Debug* (*Shift + F9*) gestartet. Das Programm pausiert am Haltepunkt. Die Anweisung in dieser Zeile wurde aber noch nicht ausgeführt.

Im Debugger-Fenster wird die Wertbelegung von Variablen angezeigt. Diese Werte werden auch im Quellcode gezeigt und ebenso, wenn man mit der Maus über eine Variable im Quellcode fährt (siehe Abbildung).

Mit *Step Into* (*F7*) wird die Anweisung ausgeführt und in die nächste Zeile gesprungen.

Mit *Step Over* (*F8*) geschieht Ähnliches. Methoden werden ausgeführt, aber die Ausführung einer Methode wird nicht schrittweise verfolgt.

Mit *Step Out* (*Shift + F8*) springt man aus der aktuellen Methode wieder heraus.

Mit *Resume* (*F9*) läuft das Programm bis zum nächsten Haltepunkt weiter.

Mit *Rerun* (*Strg + F5*) wird der Debugger neu gestartet.

```
④ Test.java x
1 ► public class Test {
2 @     public static double average(double ...n) {    n: [1.0, 2.0, 3.0]
3         double result = 0;    result: 0.0
4         for (double i : n) {    n: [1.0, 2.0, 3.0]    i: 1.0
5             result += i;    result: 0.0    i: 1.0
6         }
7         return result / n.length;
8     }
9
10 ►    public static void main(String[] args) {
11        double[] numbers = {1, 2, 3};
12        double avg = average(numbers);
13        System.out.println(avg);
14    }
15 }
```

Fhrt man das Programm schrittweise aus, stellt man fest, dass am Ende der Methode average die Division durch die Anzahl Zahlen im Array vergessen wurde:

```
return result / n.length;
```

Eine Dokumentation des Debuggers findet man unter:

<https://www.jetbrains.com/help/idea/debugging-code.html>

### 34.3 Testen mit JUnit

*JUnit* ist ein populäres Framework zum automatisierten Testen einzelner Klassen oder Methoden von Java-Programmen.

Um JUnit in einem bestehenden IntelliJ-Projekt nutzen zu können, muss eine zusätzliche Bibliothek wie folgt eingebunden werden:

*File > Project Structure*

Wählen Sie *Libraries*, klicken auf + und wählen dann *From Maven*.

Geben Sie im Suchfeld *org.junit.jupiter* ein und suchen Sie in der Auswahlliste nach der neuesten Version von

*org.junit.jupiter:junit-jupiter:5.x.x*,

kreuzen *Download to* an und tragen den Pfad zum Unterverzeichnis *lib* Ihres Projekts ein.

Nachdem Sie auf *OK* geklickt haben, werden die benötigten jar-Dateien in das Verzeichnis *lib* heruntergeladen. Unter *Modules, Dependencies* sehen Sie nun die eingebundene Testbibliothek.

Die weitere Vorgehensweise soll am Beispiel der folgenden Klasse demonstriert werden:

```
public class Calculator {  
    public int divide(int a, int b) {  
        if (b == 0) {  
            throw new IllegalArgumentException("Attempt to divide by zero");  
        }  
        return a / b;  
    }  
}
```

*Testklassen* enthalten Methoden zum Testen von Code und verwenden besondere Annotationen. Wir speichern diese Klassen im Quellcode-Verzeichnis *test*, das parallel zu *src* liegt:

*File > New > Directory: test*

Rechts-Klick auf *test*: *Make Directory as Test Sources Root*

Die zur Klasse *Calculator* passende Testklasse erhält den Namen *CalculatorTest*.

*Testmethoden* werden mit der Annotation *@Test* ausgezeichnet, sind *public*, besitzen keine Parameter und haben den Return-Typ *void*.

Testfälle sind nach dem *Triple-A-Muster*

*Arrange, Act, Assert*

aufgebaut.

- *Arrange*

Vorbereitung der Umgebung. In unserem Beispiel heißt das, dass eine Instanz von `Calculator` erzeugt werden muss.

- *Act*

Die zu testende Aktion wird ausgeführt, in unserem Fall die Methode `divide`.

- *Assert*

Hier wird geprüft, ob das Ergebnis den Erwartungen entspricht.

Neben `@Test` existieren noch weitere Annotationen, die zum Teil im Beispiel genutzt werden:

`@DisplayName` Kurze Beschreibung der Testmethode

`@BeforeEach` Diese Methode wird vor jeder Testmethode ausgeführt.

`@BeforeAll` Diese Methode wird genau einmal vor allen Testmethoden ausgeführt.

`@AfterEach` Diese Methode wird nach jeder Testmethode ausgeführt.

`@AfterAll` Diese Methode wird genau einmal nach allen Testmethoden ausgeführt.

`@Disable` Diese Testmethode wird nicht ausgeführt.

Methoden der Klasse `org.junit.jupiter.api.Assertions` ermöglichen die Überprüfung der Ergebnisse. Hier eine Auswahl:

```
assertEquals(expected, actual)
assertFalse(expression)
assertTrue(expression)
assertNull(actual)
assertNotNull(actual)
assertThrows()
```

Die Testklasse:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
public class CalculatorTest {
    Calculator calculator;
```

```
@BeforeEach
void setUp() {
    calculator = new Calculator();
}

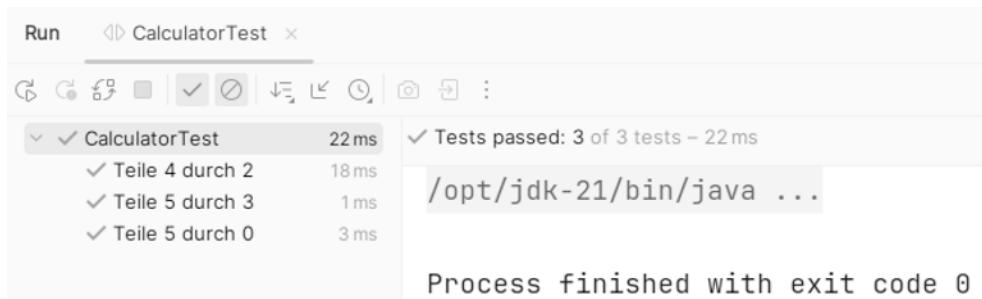
@Test
@DisplayName("Teile 4 durch 2")
public void divide1() {
    int result = calculator.divide(4, 2);
    assertEquals(2, result);
}

@Test
@DisplayName("Teile 5 durch 3")
public void divide2() {
    int result = calculator.divide(5, 3);
    assertEquals(1, result);
}

@Test
@DisplayName("Teile 5 durch 0")
public void divide3() {
    Throwable exception = assertThrows(IllegalArgumentException.class,
        () -> calculator.divide(5, 0));
    assertEquals("Attempt to divide by zero", exception.getMessage());
}
```

Die Testklasse enthält drei Testmethoden. Die ersten beiden testen die ganzzahlige Division. Mit der dritten Testmethode wird die ausgelöste Exception überprüft. Dazu werden der erwartete Typ der Exception und die zu testende Aktion übergeben. Vor jedem Test wird eine `Calculator`-Instanz erzeugt.

Mit den kleinen grünen Play-Knöpfen links neben dem Quellcode im Editor können einzelne oder auch alle Testfälle ausgeführt werden. Alternativ kann *Run* im Menü gewählt werden.



Das Projekt im Begleitmaterial zu diesem Anhang enthält im Verzeichnis *lib* eine jar-Datei<sup>1</sup>, mit der der Test im Terminal-Fenster ausgeführt werden kann:

```
java -jar lib/junit-platform-console-standalone-1.10.1.jar \
-cp out/production/kap34:out/test/kap34 \
-c CalculatorTest
```

---

<sup>1</sup> <https://mvnrepository.com/artifact/org.junit.platform/junit-platform-console-standalone/1.10.1>

# Literaturhinweise

Die folgenden Quellen sind für eine Vertiefung einzelner Themen dieses Buches gut geeignet.

Abts, D.: *Masterkurs Client/Server-Programmierung mit Java. Anwendungen entwickeln mit Standard-Technologien*. Springer Vieweg, 6. Auflage 2022

Baun, C.: *Computernetze kompakt*. Springer Vieweg, 6. Auflage 2022

Bloch, J.: *Effective Java. Best Practices für die Java-Plattform*. dpunkt.verlag 2018

Chin, S. et al.: *The Definitive Guide to Modern Java Clients with JavaFX 17*. Apress, 2nd Edition 2022

Hettel, J. u. a.: *Nebenläufige Programmierung mit Java*. dpunkt.verlag 2016

Inden, M.: *Java – die Neuerungen in Version 17 LTS, 18 und 19*. dpunkt.verlag 2022

*JavaFX – Dokumentation*: <https://openjfx.io>

*JDBC Database Access Tutorial*: <https://docs.oracle.com/javase/tutorial/jdbc>

Lampe, J.: *Clean Code für Dummies*. Wiley-VCH 2020

Laube, M.: *Einstieg in SQL*. Rheinwerk Computing, 3. Auflage 2022

Maurice, F.: *HTML & CSS für Dummies*. Wiley-VCH, 2. Auflage 2023

Martin, R. C.: *Clean Architecture*. mitp 2018

Oechsle, R.: *Parallele und verteilte Anwendungen in Java*. Hanser, 6. Auflage 2022

Oelmann, G.: *Modularisierung mit Java 9: Grundlagen und Techniken für langlebige Softwarearchitekturen*. dpunkt.verlag 2018

# Sachwortverzeichnis

- @  
@FunctionalInterface 276  
@FXML 518  
@Override 82, 100
- A**  
abgeleitete Klasse 81  
abstract 91  
AbstractTableModel 488  
Abstract Window Toolkit 435  
abstrakte Klasse 91  
abstrakte Methode 91  
ActionEvent 459, 460, 469, 473, 478  
ActionListener 459, 469  
Adapterklasse 441  
Alert 595  
AnchorPane 510  
Annotation 82  
anonyme Klasse 126  
Anweisung 31  
API-Evolution 109  
Application 507  
Argumente 56  
ArithmeticException 139  
arithmetischer Operator 19  
ARM 335  
Array 45  
ArrayList 262  
Arrays 196, 227  
Attribut 51  
Aufzählung 129  
Aufzählungstyp 131  
Ausdruck 19  
ausführbare Klasse 64  
Ausgabestrom 328  
Ausnahme 137  
Ausnahmebehandlung 137
- Ausnahmen-Verkettung 146  
Auswahlkomponente 472  
Autoboxing 172  
AutoCloseable 335, 543  
Automatic Module 653, 656, 657  
Automatic Resource Management 335  
Auto-Unboxing 172  
AWT 435, 505
- B**  
Balkendiagramm 532  
BarChart 532  
Basisklasse 81  
Bedingungsoperator 26  
Bestätigungsdialog 484  
Bezeichner 11  
Bibliothek 80  
BiFunction 277  
BigInteger 202  
BinaryOperator 277  
Bitoperator 24  
Bivarianz 250  
Block 31  
BlockingQueue 408  
boolean 14  
Boolean 167  
Border 457  
BorderLayout 450  
Box 452  
BoxLayout 452  
break 38  
BufferedInputStream 329, 336  
BufferedOutputStream 329, 336  
BufferedReader 332, 343  
BufferedWriter 332, 343  
Button 509, 510  
ButtonGroup 459

- byte 15  
Byte 167  
ByteArrayInputStream 329  
ByteArrayOutputStream 329  
Bytecode 3  
Bytestrom 328
- C**  
Calendar 207  
Callable 417  
call by value 56  
CamelCase 11  
Cascading Style Sheets 505, 513  
Cast-Operator 27  
catch 143  
catch or throw 140  
CategoryAxis 532  
ChangeEvent 466, 475  
ChangeListener 466  
char 14  
Character 167  
character encoding 326  
character set 325  
CharArrayReader 332  
CharArrayWriter 332  
Charset 326, 327  
Chart 527  
ChronoUnit 211  
Class 191  
CLASSPATH 76  
Clean Code 615  
Client 567  
Client/Server 567  
clone 177  
Cloneable 177  
Closure 282  
Codierungsschema 326  
Collator 227  
Collection 259
- Collections 270, 289  
Collections Framework 180, 259  
Color 439  
ComboBox 510, 511  
ComboBoxEditor 473  
Comparable 196, 245  
Comparator 279, 289  
CompletableFuture 418  
CompletionStage 419  
Component 435  
Connection 543, 544, 546  
Consumer 277  
Container 180, 435  
Content Pane 438  
continue 39  
Contravarianz 252  
Controller 436  
CountDownLatch 412  
Covarianz 251  
CSS 505, 513
- D**  
Daemon Thread 382  
DataInputStream 329, 337, 338  
DataOutputStream 329, 337  
Date 205  
Date and Time API 209, 225  
DateFormat 224  
Dateizeiger 349  
Datenbank 541  
Datenbankmanagementsystem 541  
Datenbanksystem 541  
Datenbank-Transaktion 559  
Datenkapselung 94  
Datenkomprimierung 352  
Datenstrom 328  
Datentyp 13  
DateTimeFormatter 210  
DayOfWeek 212

---

DBMS	541	EOFException	339
Deadlock	393	equals	173
DecimalFormat	200, 224	Ereignis	440
default	105	Ereignisempfänger	440
Default-Methode	105	Ereignismodell	440
Default-Paket	77	Ereignisquelle	440
Dekrementierungsoperator	20	err	188
Delegation	117	Error	137
Dependency-Inversion-Prinzip	615, 627	Ersetzbarkeitsprinzip	621
Deserialisierung	363	erweiterte Klasse	81
Diagramm	527	Escape-Sequenz	14
Diamond Operator	244	Event	440
Dimension	479	Event-Dispatcher	493
DIP	627	Event-Handler	529, 532
DirectoryStream	322	Event-Modell	440
Distribution	3	EventQueue	493, 495
DNS	565	Exception	137, 138, 139
do	36	exception chaining	146
Domain Name System	565	Exception Handling	137
double	15	Executor	416
Double	167	Executors	415, 423
Downcast	88, 101	ExecutorService	416
DriverManager	543	Explicit Module	656
Duration	211	exports	638, 642
Durchfallen	33	extends	81, 99
dynamisches Binden	93		
<b>F</b>			
Fall Through			
effectively final			
einfacher Datentyp			
Einfachvererbung			
Eingabedialog			
Eingabe-Fokus			
Eingabestrom			
EmptyBorder			
encoding scheme			
enum			
enum-Aufzählung			
Enumeration			
File			
FileChooser			
FileFilter			
FileInputStream			
FileNotFoundException			
FileOutputStream			
Filepointer			
FileReader			

- Files 321  
FileVisitor 323  
FileWriter 332, 343  
FilterInputStream 329  
FilterOutputStream 329  
FilterReader 332  
FilterWriter 332  
final 58, 94  
finally 145  
flache Kopie 178  
Fließkommotyp 15  
Fließkommazahl 15  
float 15  
Float 167  
FlowLayout 449  
Font 439  
for 36  
foreach 38, 47, 182, 260  
forEach 285  
formatierte Ausgabe 345  
Fortschrittsbalken 497  
Function 277  
funktionale Schnittstelle 276  
Funktionsinterface 276  
Future 417, 419  
FXCollections 529  
FXML 514  
FXMLLoader 518
- G**
- ganzzahliger Typ 15  
Garbage Collection 2  
Garbage Collector 62  
Generics 241, 255  
generische Klasse 242  
generische Methode 254  
generischer Typ 242  
generisches Interface 242  
Generizität 241
- geprüfte Ausnahme 138  
geschachtelte Klasse 121  
GET 573  
Gleitkommazahl 15  
Grafikkontext 438  
Graphical User Interface 435  
Graphics 438  
graphviz 641  
GregorianCalendar 206  
GridBagConstraints 454  
GridBagLayout 454  
GridLayout 451  
GridPane 510  
Gson 369  
GUI 435  
gzip 352  
GZIPInputStream 352  
GZIPOutputStream 352
- H**
- hashCode 174  
Hashcode 174  
HashMap 267  
HashSet 264  
Hashtable 183, 267  
HBox 522  
Heap 62  
Hintergrund-Thread 383  
Hotkey 478  
HTML 572  
HTTP 571  
HTTP-Body 576  
HttpClient 579  
HTTP-Client-API 579  
HTTP-Header 576  
HttpRequest 579  
HTTP-Request 572  
HttpResponse 580  
HTTP-Response 572, 576

- 
- Hüllklasse 167  
HyperText Transfer Protocol 571
- I**
- I18N 221  
Icon 458  
IDE 6  
Identität 54  
if 31  
ImageIcon 458  
implements 100  
import 76  
import static 77  
in 188  
Infinity 16  
Initialisierung 16, 46, 54, 62  
Initialisierungsblock 62  
Initializable 518  
Inkrementierungsoperator 20  
In-memory-Compilierung 6  
InputEvent 443, 478  
InputStream 329, 330  
InputStreamReader 332, 341  
Insets 454  
instanceof 88, 89, 101  
Instant 213  
Instanz 51, 54  
Instanziierung 54  
Instanzklasse 122  
Instanzmethode 64  
Instanzvariable 63  
int 15  
Integer 167  
Integrated Development Environment 6  
Interface 99  
Interface-Segregation-Prinzip 615, 624  
intermediäre Operation 295  
Internationalisierung 221  
InterruptedException 384, 403
- Invarianz 247  
invokeAndWait 496  
invokeLater 496  
IOException 330, 332, 338, 339, 349  
ISP 624  
ItemEvent 460, 473  
ItemListener 460  
Iterable 260  
Iteration 66  
iterativ 66  
Iterator 260
- J**
- jar 80  
java.base 641, 643  
Java-Applikation 49, 64  
Java-Bibliothek 80  
Java Database Connectivity 541  
Java Development Kit 1  
javadoc 12, 237  
JavaFX 505  
JavaFX-Application-Thread 519  
JavaFX Markup Language 514  
Java Runtime Environment 1  
Java-Shell 40  
Java-Systemeigenschaften 190  
Java-Technologie 1  
JButton 458  
JCheckBox 459  
JCheckBoxMenuItem 478  
JComboBox 472  
JComponent 438  
JDBC 541  
JDBC-Treiber 542  
jdeps 641  
JDialog 482  
JDK 1  
JFileChooser 485  
JFrame 437

JLabel 460  
JList 472, 473  
JMenu 477  
JMenuBar 476  
JMenuItem 477  
JOptionPane 484  
JPanel 438  
JPasswordField 468  
JPopupMenu 479  
JProgressBar 497  
JRadioButton 459  
JRadioButtonMenuItem 478  
JRE 1  
JScrollPane 463  
jshell 40  
JSlider 474  
JSON 369, 602  
JTabbedPane 465  
JTable 488  
JTextArea 469  
JTextComponent 467  
JTextField 468  
JToolBar 478  
JVM 3  
jwebserver 578

**K**

KeyAdapter 442  
KeyEvent 478  
KeyListener 441  
KeyStroke 478  
Klasse 51, 52  
Klassenliteral 192  
Klassenmethode 64  
Klassenvariable 63  
Kommandozeilen-Parameter 442  
Kommentar 12  
kompatibel 245  
Kompatibilitätsmodus 637

Kompressionsfaktor 353  
Konstante 58, 94, 129  
Konstruktor 51, 60, 84  
Kontextmenü 479  
kontrollierte Ausnahme 138, 140  
Kopierkonstruktor 61  
Kreisdiagramm 527

**L**

Label 510  
Lambda 275, 278  
Lambda-Ausdruck 275  
LAN 565  
Ländercode 221  
LayoutManager 447  
LESS-Prinzip 253  
LIFO 264  
LineNumberReader 332  
LinkedList 263  
Linksschieben 24  
Liskovsches Substitutionsprinzip 87, 615, 621  
List 180, 261  
Listener 440  
ListSelectionEvent 490  
ListSelectionListener 490  
ListSelectionModel 473, 490  
Literal 11, 14  
Local Area Network 565  
LocalDate 209, 212  
LocalDateTime 209, 213  
Locale 222  
localhost 571  
LocalTime 209  
logischer Operator 22  
logischer Typ 14  
lokale Klasse 124  
lokale Variable 57  
long 15

- 
- Long 167
  - Lower-Typebound 252
  - LSP 621
  - LTS-Version 3
  - M**
    - main 49, 64
    - main-Thread 379
    - Map 259, 265
    - Map.Entry 266
    - markierte Anweisung 39
    - Math 199
    - mathematische Funktion 199
    - Mausereignis 443
    - mehrdimensionales Array 47
    - Mehrfachzuweisung 25
    - Member 51
    - Memoization 291
    - Menü 476
    - Methode 51, 56
    - Methodenkopf 56
    - Methodenreferenz 282
    - Methodenrumpf 56
    - Mitteilungsdialog 485
    - modal 601
    - modales Fenster 482
    - Model 436
    - Model-View-Controller-Architektur 436
    - Modifizierer 93
    - Modul 80, 636
    - Modularisierung 635
    - Moduldeskriptor 638
    - Modulsystem 635
    - Month 212
    - MouseAdapter 444
    - MouseListener 443, 444
    - MouseMotionAdapter 444
    - MouseMotionListener 444
    - MouseWheelEvent 504
    - MouseListener 504
    - Multicatch 143
    - Multitasking 379
    - Multithreading 379
    - MVC-Architektur 436
  - N**
    - Namenskonvention 11
    - native 94
    - Nebenläufigkeit 379
    - new 45, 53
    - nicht geprüfte Ausnahme 138
    - nicht kontrollierte Ausnahme 138
    - Node 509
    - non-sealed 95
    - NoSuchElementException 164, 260
    - notify 403, 408
    - notifyAll 403, 408
    - null 54
    - Null-Layout 448
    - Number 226, 244
    - NumberAxis 532
    - NumberFormat 226
  - O**
    - Oberklasse 81
    - Object 83, 159, 173
    - ObjectInputStream 330, 363
    - ObjectOutputStream 330, 363
    - Objects 176
    - Objekt 51, 53
    - ObservableList 529
    - OCP 618
    - Open-Closed-Prinzip 615, 618
    - OpenJFX 505
    - Open Module 656
    - opens 646
    - OpenWeatherMap 580
    - Operand 19

Operator 19  
Optional 308  
out 188  
OutputStream 329, 331  
OutputStreamWriter 332, 341  
Overloading 59  
Overriding 81

**P**

package 75  
paintComponent 438  
Paket 75, 80  
Parameter 56  
parametrisierter Typ 243  
Path 321  
Paths 321  
Pattern Matching 89, 90  
PECS-Prinzip 253  
permits 95  
Persistenz 363  
PieChart 527  
Pipe 408  
PipedInputStream 330, 408  
PipedOutputStream 330, 408  
PipedReader 332  
PipedWriter 332  
Platform 509, 519  
Point 443  
Polymorphie 92  
port 568  
Portnummer 565, 568, 570  
Predicate 277  
PreparedStatement 546  
Preview 2  
Primärschlüssel 541, 543  
PrintStream 329, 346  
PrintWriter 332, 344  
private 94  
Process-API 421

ProcessHandle 421  
Producer/Consumer 403  
ProgressBar 522  
Properties 185  
Property Binding 523  
Property-Liste 185  
protected 94  
provides 650, 651, 653  
Prozess 379  
public 53, 93  
Punktnotation 54  
PushbackInputStream 329, 340  
PushbackReader 332

**Q**

Quelldatei 53  
Query String 574

**R**

race condition 388  
Random 201  
RandomAccessFile 349  
Raw Type 248  
Reader 332  
Rechtsschieben 24  
record 68, 90, 100, 127  
Record Patterns 90  
reentrant 389  
Referenztyp 54, 100  
Referenzvariable 54  
Reflection 645  
Rekursion 66  
rekursiv 66  
relationaler Operator 22  
relationales Datenbanksystem 541  
Releasezyklus 2  
repaint 439  
requires 638, 642  
ResourceBundle 228

- Ressourcenbündel 228  
Rest-Operator 21  
ResultSet 549  
return 56  
Reverse Domain Name Pattern 77, 637  
Rückgabetyp 56  
Rückgabewert 56  
Runnable 380  
Runtime 415  
RuntimeException 138
- S**
- Scanner 188  
Scene 509  
Scene Builder 506  
SceneGraph 509  
Schiebeoperator 24  
Schleife 35  
Schlüsseltabelle 265  
Schlüsselwort 11  
sealed 95  
Selektor 514  
SequencedCollection 269  
SequencedMap 269  
SequenceInputStream 330  
Serialisierung 363  
Serializable 364  
serialVersionUID 364  
Server 567  
ServerSocket 569  
Service Consumer 231  
ServiceLoader 232, 650  
Service Provider 231  
Set 264  
short 15  
Short 167  
short circuit 22  
Shutdown 415  
Signatur 58
- SimpleDateFormat 205  
SimpleFileVisitor 323  
Simple Web Server 578  
Single-Responsibility-Prinzip 615  
Singleton 134, 399  
skalierbare Plattform 636  
Socket 567, 568  
SOLID 615  
spätes Binden 93  
Split-Package 636  
Sprachcode 221  
Sprunganweisung 38  
SQL 541  
SQLite 541  
SRP 615  
Stack 62, 264  
StackPane 509  
Stage 509  
StandardCharsets 326  
Standarddatenstrom 188, 328  
Standardkonstruktor 60  
Standardverzeichnisse 317  
Statement 544  
static 63, 64, 110, 121  
statische import-Klausel 77  
statische Initialisierung 64  
statische Klasse 121  
statische Methode 64  
statischer Initialisierungsblock 64  
statische Variable 63  
Stream 295, 328  
strenge Kapselung 636  
String 16, 151  
StringBuffer 161  
StringBuilder 161  
StringReader 332  
StringTokenizer 163  
StringWriter 332

- Structured Query Language 541  
Style-Klasse 514  
Subklasse 81  
super 83, 84  
Superklasse 81  
Supplier 277  
Swing 435, 505  
switch 32, 33, 89  
switch-Ausdruck 33  
Symbolleiste 478  
Synchronisation 388, 403  
synchronized 93, 389, 391  
System 188  
System.err 328  
System.in 328  
System.out 328  
System Properties 190
- T**
- TableModel 488  
TableModelEvent 489  
TableModelListener 489  
TableView 534  
Task 522  
TCP/IP 565  
terminale Operation 295  
ternärer Operator 26  
Textblock 160  
TextField 510  
Textkomponente 467  
this 58, 62, 84  
Thread 379, 380, 381  
ThreadLocal 402  
Thread-Pool 415  
Thread-sicher 161, 397  
throw 140  
Throwable 137, 139  
throws 141  
tiefe Kopie 178
- TimeZone 206  
TitledBorder 457  
T-Konsument 253  
Tooltip 458  
Top-Level-Klasse 121  
toString 159  
T-Produzent 252  
transient 93  
transitive Abhängigkeit 643  
TreeMap 267  
true 14  
try 143  
try with resources 335  
Typargument 243  
Typebound 244  
Type-Erasure 248  
Type-Selektor 514  
Typ-Inferenz 254, 280  
Typparameter 241, 242  
typsicher 130  
Typumwandlung 20, 27
- Ü**
- Überladen 59  
Überschreiben 81
- U**
- UI-Thread 519  
Umgebungsvariable 190  
UnaryOperator 277  
Unicode 14  
Uniform Resource Identifier 579  
Uniform Resource Locator 194, 565, 579  
Unnamed Module 657  
Unterbrechungssignal 384  
Unterklasse 81  
Untermenü 477  
Upcast 87  
Upper-Typebound 251

- URI 579  
URL 194, 565, 566, 579  
URLDecoder 574  
User Thread 383  
uses 650, 651, 652  
UTF-8 152, 338, 342
- V**  
var 17  
Varargs 65  
Variable 13, 16, 51  
VBox 522  
Vector 180, 261  
Vererbung 81  
Verhalten 51  
versiegelte Klasse 95  
Verzweigung 31  
View 436  
virtuelle Maschine 3  
virtueller Thread 423  
void 56  
volatile 93, 386, 387  
Vordergrund-Thread 383
- W**  
Wahrheitswert 14  
wait 403  
Wettlaufbedingung 388  
when 89  
while 35  
white space 153  
Wildcard-Typ 249, 250  
WindowAdapter 442  
WindowListener 441  
Wrapper-Klasse 167  
Writer 332, 333
- X**  
XYChart.Data 532  
XYChart.Series 532
- Z**  
Zeichen 14  
Zeichencodierung 326  
Zeichenkette 16  
Zeichenstrom 331  
Zeichentyp 14  
ZIP-Dateiformat 354  
ZipEntry 354  
ZipFile 355  
ZipInputStream 356  
ZipOutputStream 354  
ZonedDateTime 212  
ZoneId 212  
Zufallszahl 200, 201  
Zugriffsrecht 93  
Zustand 51  
zuverlässige Konfiguration 636  
Zuweisungsoperator 25  
zyklische Abhängigkeit 648