

LEHRBUCH

Dietmar Abts

Grundkurs JAVA

Von den Grundlagen bis zu
Datenbank- und Netzanwendungen

8. Auflage



Springer Vieweg

Grundkurs JAVA

Dietmar Abts

Grundkurs JAVA

Von den Grundlagen bis zu
Datenbank- und Netzanwendungen

8., überarbeitete und erweiterte Auflage



Springer Vieweg

Dietmar Abts
HS Niederrhein
Mönchengladbach, Deutschland

ISBN 978-3-658-07967-3 ISBN 978-3-658-07968-0 (eBook)
DOI 10.1007/978-3-658-07968-0

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

Springer Vieweg

© Springer Fachmedien Wiesbaden 1999, 2000, 2002, 2004, 2008, 2010, 2013, 2015

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 Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. 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.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Fachmedien Wiesbaden ist Teil der Fachverlagsgruppe Springer Science+Business Media
(www.springer.com)

Vorwort zur achten Auflage

Das vorliegende Buch führt Sie schrittweise durch die wichtigsten Aspekte von Java, von den elementaren Grundlagen über objektorientierte Konzepte und grafische Benutzungsoberflächen bis zu Datenbankzugriffen und Kommunikation im Netzwerk.

Zudem schließt dieser Grundkurs mit zwei ausführlichen Einführungen zu aktuellen Themen: das Persistenz-Framework JPA zur Speicherung von Objekten in relationalen Datenbanken und die Objektdatenbank db4o.

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.

Die im März 2014 veröffentlichte *Java-Version 8* führt die seit Langem bedeutendste Spracherweiterung, die Kernideen aus der Welt der funktionalen Programmierung enthält, ein: *Lambda-Ausdrücke*. Was es damit auf sich hat, erfahren Sie im Kapitel 7. Dieses Kapitel enthält auch ein Beispiel zur Illustration des neu eingeführten *Stream-API*.

Im Kapitel 3 werden u. a. die dafür notwendigen Anpassungen des Interface-Konzepts berücksichtigt: *Default-Methoden*.

Gegenüber der vorhergehenden Auflage sind zahlreiche Verbesserungen und Ergänzungen vorgenommen worden. Die wichtigsten sind:

- Versionsnummern bei der Serialisierung von Objekten sowie von der Standard-Serialisierung abweichende Sonderbehandlungsmaßnahmen,
- Behandlung von Daemon-Threads,
- Aufruf von Applet-Methoden mit JavaScript,
- Ergänzungen zum Java Persistence API: eingebettete Klassen, Vererbungsstrategien, Lebenszyklusmethoden, optimistisches Sperren in Multi-User-Anwendungen,
- Callback-Methoden bei db4o.

Der *Quellcode von über 350 Programmbeispielen* (inkl. Lösungen zu den Aufgaben) ist im Internet auf der Website des Verlags bei den bibliographischen Angaben zu diesem Buch verfügbar:

Alle Beispiele und Lösungen wurden mit Java SE 8 unter Windows ausführlich getestet.

Danken möchte ich zum Schluss Frau Sybille Thelen vom Lektorat IT für die gute Zusammenarbeit sowie meinen Leserinnen und Lesern für die guten Vorschläge, die in dieser Auflage berücksichtigt wurden.

Ratingen, November 2014

Dietmar Abts

abts@hs-niederrhein.de

www.dietmar-abts.de

Inhaltsverzeichnis

1	Einleitung	1
1.1	Entwicklungsumgebung	1
1.2	Vom Quellcode zum ausführbaren Programm	3
1.3	Erste Beispiele	4
1.4	Wichtige Merkmale der Programmiersprache Java	6
1.5	Zielsetzung und Gliederung des Buches	7
1.6	Programm- und Aufgabensammlung	10
2	Imperative Sprachkonzepte	11
2.1	Kommentare und Bezeichner	11
2.2	Einfache Datentypen und Variablen	13
2.3	Ausdrücke und Operatoren	17
2.4	Kontrollstrukturen	25
2.5	Aufgaben	31
3	Objektorientierte Sprachkonzepte	35
3.1	Klassen und Objekte	35
3.2	Methoden	40
3.3	Konstruktoren	44
3.4	Klassenvariablen und Klassenmethoden	47
3.5	Vererbung	49
3.6	Abstrakte Klassen	55
3.7	Modifizierer	57
3.8	Interfaces	59
3.9	Innere Klassen	68
3.10	Arrays	74
3.11	Aufzählungen	80

3.12	Pakete	84
3.13	Aufgaben	89
4	Ausnahmebehandlung	97
4.1	Ausnahmetypen	98
4.2	Auslösung und Weitergabe von Ausnahmen	100
4.3	Abfangen von Ausnahmen	102
4.4	Verkettung von Ausnahmen	106
4.5	Aufgaben	108
5	Ausgewählte Standardklassen	111
5.1	Zeichenketten.....	111
5.1.1	Die Klasse String	111
5.1.2	Die Klassen StringBuffer und StringBuilder	116
5.1.3	Die Klasse StringTokenizer.....	118
5.2	Hüllklassen und Autoboxing	120
5.3	Die Klasse Object	126
5.3.1	Die Methoden equals und hashCode	126
5.3.2	Flache und tiefe Kopien	129
5.4	Container	132
5.4.1	Die Klasse Vector	133
5.4.2	Die Klasse Hashtable	135
5.4.3	Property-Listen.....	138
5.5	Die Klasse System	140
5.6	Die Klasse Class.....	142
5.7	Die Klasse Arrays	147
5.8	Mathematische Funktionen	149
5.9	Datum und Zeit	155
5.10	Internationalisierung	159
5.11	Aufgaben	165

6	Generische Typen	169
6.1	Motivation und Definition.....	169
6.2	Typparameter mit Einschränkungen	172
6.3	Raw Types.....	174
6.4	Wildcard-Typen	176
6.5	Generische Methoden.....	179
6.6	Grenzen des Generics-Konzepts	181
6.7	Generische Container	181
6.7.1	Listen	181
6.7.2	Schlüsseltabellen.....	184
6.8	Aufgaben	187
7	Lambda-Ausdrücke.....	189
7.1	Funktionsinterfaces.....	190
7.2	Lambda-Ausdrücke	192
7.3	Methodenreferenzen.....	196
7.4	Ein Beispiel zum Stream-API für Collection-Klassen	200
7.5	Aufgaben	202
8	Ein- und Ausgabe	205
8.1	Die Klasse File	205
8.2	Datenströme.....	209
8.2.1	Byteströme	210
8.2.2	Zeichenströme	212
8.3	Dateien byteweise kopieren	215
8.4	Daten im Binärformat lesen und schreiben	218
8.5	Pushback	220
8.6	Zeichencodierung	222
8.7	Zeichenweise Ein- und Ausgabe.....	223
8.8	Gefilterte Datenströme	227
8.9	Serialisierung von Objekten	229
8.10	Wahlfreier Dateizugriff	234

8.11	Datenkomprimierung	238
8.12	Aufgaben	243
9	Threads	247
9.1	Threads erzeugen und beenden	248
9.2	Synchronisation	255
9.3	Kommunikation zwischen Threads	264
9.4	Shutdown-Threads.....	273
9.5	Aufgaben	275
10	Grafische Benutzungsoberflächen	279
10.1	Übersicht.....	280
10.2	JFrame	281
10.3	JPanel und Methoden zum Zeichnen	284
10.4	Ereignisbehandlung	289
10.5	Layout-Manager	295
10.6	Buttons	307
10.7	Labels	314
10.8	Spezielle Container	316
10.9	Textkomponenten	321
10.10	Auswahlkomponenten	328
10.11	Menüs und Symbolleisten.....	335
10.12	Mausaktionen und Kontextmenüs.....	340
10.13	Dialogfenster.....	346
10.14	Tabellen	355
10.15	Aktualisierung der GUI-Oberfläche	361
10.16	Aufgaben	367
11	Applets.....	373
11.1	Der Lebenszyklus eines Applets	374
11.2	Die Appletumgebung	376
11.3	Hybridanwendungen	381

11.4	Wiedergabe von Bild- und Audiodaten.....	383
11.5	Zugriffsrechte für Applets	389
11.6	Aufgaben	392
12	Datenbankzugriffe mit JDBC	395
12.1	Konfiguration und Verbindungsaufbau	395
12.2	Daten suchen und anzeigen	399
12.3	Daten ändern	405
12.4	Aufgaben.....	408
13	Netzwerkkommunikation mit TCP/IP	411
13.1	Dateien aus dem Netz laden	411
13.2	Eine einfache Client/Server-Anwendung	413
13.3	HTTP-Transaktionen	417
13.3.1	Formulardaten über HTTP senden	418
13.3.2	Ein spezieller HTTP-Server für SQL-Anweisungen	422
13.4	Aufgaben.....	427
14	Fallbeispiel	431
14.1	Die Anwendung.....	431
14.2	Drei-Schichten-Architektur	433
14.3	Klassenentwurf und Architektur.....	435
14.4	Implementierung	436
14.4.1	Persistenzschicht.....	436
14.4.2	Anwendungsschicht.....	440
14.4.3	Präsentationsschicht	449
14.5	Bereitstellung der Anwendung.....	456
15	Exkurs: Das Java Persistence API	459
15.1	Einleitung.....	459
15.2	Einrichten der Entwicklungsumgebung.....	462
15.3	Entity-Klassen.....	462
15.4	Der Entity Manager	465

15.4.1 Persistenzeinheit	465
15.4.2 Persistenzkontext	466
15.4.3 Der Lebenszyklus der Entity-Objekte	467
15.4.4 Erzeugen eines Entity-Objekts	468
15.4.5 Lesen eines Entity-Objekts.....	469
15.4.6 Aktualisieren eines Entity-Objekts	470
15.4.7 Die Methode merge	471
15.4.8 Löschen eines Entity-Objekts	472
15.5 Entity-Beziehungen.....	473
15.5.1 OneToOne	474
15.5.2 OneToMany und ManyToOne	478
15.5.3 ManyToMany	486
15.6 Abfragen.....	492
15.7 Eingebettete Klassen	495
15.8 Vererbung.....	500
15.8.1 SINGLE_TABLE	501
15.8.2 TABLE_PER_CLASS.....	505
15.8.3 JOINED.....	506
15.9 Lebenszyklusmethoden.....	508
15.10 Optimistisches Sperren in Multi-User-Anwendungen	514
15.11 Aufgaben	522
16 Exkurs: Die Objektdatenbank db4o.....	527
16.1 Einleitung	527
16.2 CRUD-Operationen	532
16.3 Objektidentität	537
16.4 Native Abfragen	540
16.5 Tiefe Objektgraphen.....	543
16.6 Callbacks.....	545
16.7 Aufgaben	551

Quellen im Internet	555
Literaturhinweise	557
Sachwortverzeichnis	559

1 Einleitung

Java wird als universelle Programmiersprache für eine Vielzahl von Anwendungen in der industriellen Praxis auf der Client- und insbesondere auf der Serverseite eingesetzt. Sie dient als Standard für die Entwicklung von Unternehmenssoftware und Webanwendungen sowie in technische Systeme (Geräte der Unterhaltungselektronik, der Medizintechnik usw.) eingebettete und mobile Anwendungen (z. B. Apps auf der Basis des Betriebssystems Android).

Ein hervorstechendes Merkmal von Java ist die Plattformunabhängigkeit, d. h. in Java programmierte Anwendungen sind ohne Portierung auf nahezu allen Rechnersystemen lauffähig.

Java hat von den Erfahrungen mit anderen Programmiersprachen wie Smalltalk, C und C++ profitiert. Wesentliche Konzepte wurden übernommen. Auf allzu komplexe und fehleranfällige Eigenschaften wurde bewusst verzichtet, um die Sprache verhältnismäßig einfach und robust halten zu können.¹

Lernziele

Nach dem Durcharbeiten dieses Kapitels

- wissen Sie, welche Schritte zur Entwicklung eines Java-Programms nötig sind,
- kennen Sie die Hauptbestandteile der Java SE Plattform,
- wissen Sie, womit sich die folgenden Kapitel dieses Buches beschäftigen und wie die Programmbeispiele des Begleitmaterials (Online-Service) genutzt werden können.

1.1 Entwicklungsumgebung

Die Java-Technologie ist für den Einsatz auf unterschiedlichen Endgeräten in verschiedene Editionen eingeteilt, die sich im Umfang der mitgelieferten Werkzeuge und Bibliotheken unterscheiden:

- Java Standard Edition (*Java SE*) zur Entwicklung und Ausführung von Desktop-Anwendungen. Diese Edition ist Basis des vorliegenden Buches.

¹ Wie sich Java von den Anfängen bis heute entwickelt hat, kann man auf der Webseite <http://oracle.com.edgesuite.net/timeline/java/> erfahren.

- Java Enterprise Edition (*Java EE*) vereinigt eine Reihe von Technologien, um mehrschichtige verteilte Anwendungen (Unternehmensanwendungen, Webanwendungen) zu entwickeln.
- Java Micro Edition (*Java ME*) für eingebettete Systeme und mobile Geräte.
- Java Card zur Ausführung von Java-Anwendungen auf Chipkarten.

Java SE

Das kostenlos zur Verfügung gestellte *Java SE Development Kit* (JDK) der Standard Edition enthält die Klassenbibliotheken, API-Beschreibungen und Programme, die man zum Übersetzen, Ausführen und Testen von Java-Programmen braucht. Das JDK bietet allerdings keine komfortable grafische Oberfläche. Die Programme werden auf Kommandozeilenebene aufgerufen. Diese Entwicklungsumgebung liegt unter der im Quellenverzeichnis aufgeführten Webadresse zum Herunterladen bereit. Beachten Sie die betriebssystemspezifischen Installationshinweise.

Das JDK enthält die Java-Laufzeitumgebung JRE (*Java Runtime Environment*), die für die Ausführung der Programme verantwortlich ist. Abbildung 1-1 zeigt den Zusammenhang zwischen JDK und JRE.

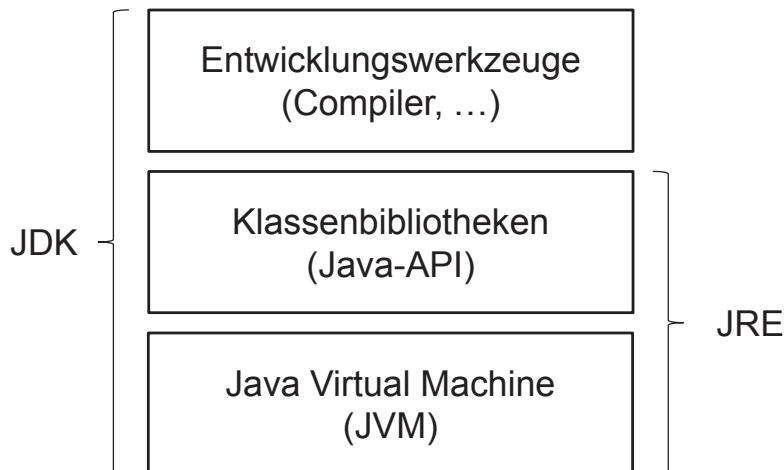


Abbildung 1-1: Bestandteile von Java SE

Weitere Entwicklungsumgebungen

Von verschiedenen Herstellern werden auf der Basis von Java SE integrierte Entwicklungsumgebungen (*Integrated Development Environment*, IDE) zur Unterstützung von komplexen Projekten mit teamfähigen Werkzeugen angeboten, beispielsweise *Eclipse*, *NetBeans*, *JDeveloper* von Oracle und *IntelliJ IDEA* von JetBrains.

Es gibt eine Reihe von einfach zu bedienenden Freeware-Editoren, aus denen heraus der erstellte Java-Code direkt kompiliert und gestartet werden kann.²

1.2 Vom Quellcode zum ausführbaren Programm

Plattformunabhängigkeit und Internet-Fähigkeit sind wichtige Eigenschaften von Java, zusammengefasst in dem Slogan: *Write once – run anywhere*. Das vom Java-Compiler aus dem Quellcode erzeugte Programm, der so genannte *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 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 performancekritische Teile des Bytecodes vor der Programmausführung in Prozessorbefehle übersetzt.

Schritte der Programmentwicklung

(vgl. Abbildung 1-2)

1. Quellcode mit einem Editor erstellen (`Demo1.java`).
2. Quellcode kompilieren (`javac Demo1.java`). Sofern kein Fehler gemeldet wird, liegt nun der Bytecode `Demo1.class` vor.
3. Bytecode ausführen (`java Demo1`).

² Das Quellenverzeichnis am Ende des Buches enthält die URLs.

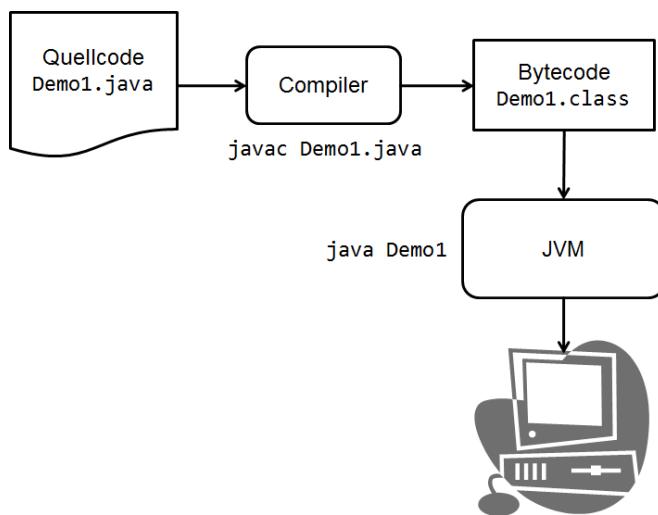


Abbildung 1-2: Übersetzung und Ausführung

1.3 Erste Beispiele

Um einen ersten Eindruck von Java und dem Umgang mit den Werkzeugen der Entwicklungsumgebung JDK zu vermitteln, wird ein einfaches Anzeigeprogramm in zwei Versionen vorgestellt:

- als zeilenorientierte Anwendung und
- als Anwendung mit grafischer Oberfläche.

Der Quellcode wird jeweils mit einem Texteditor erfasst und unter dem Dateinamen `Demo1.java` bzw. `Demo2.java` gespeichert.

Programm 1.1

Zeilenorientierte Anwendung

```

public class Demo1 {
    public static void main(String[] args) {
        System.out.println("Viel Erfolg mit dem Grundkurs Java");
    }
}
  
```

Die Datei `Demo1.java` enthält die Definition einer Klasse mit dem Namen `Demo1`. Jede ausführbare Java-Anwendung startet mit der `main`-Methode. Die Methode `System.out.println` gibt eine Zeichenkette am Bildschirm aus.

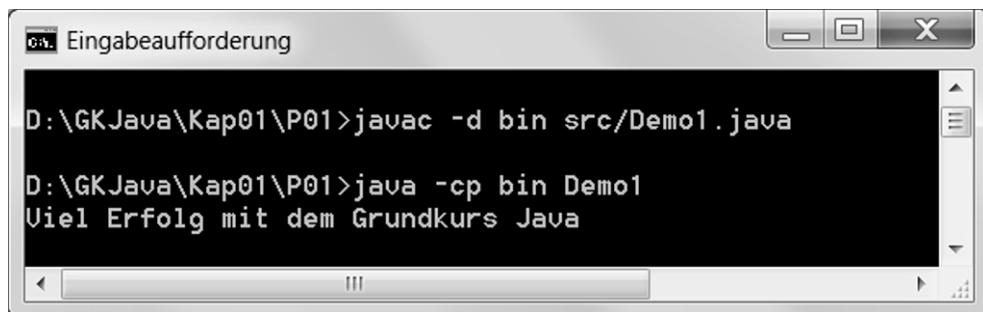


Abbildung 1-3: Übersetzen und ausführen auf Kommandozeilenebene

Wir gehen hier von der folgenden Annahme aus:

P01 enthält die beiden Unterverzeichnisse `src` und `bin`. `src` enthält die Datei `Demo1.java`, `bin` ist zurzeit noch leer. In `bin` soll der Bytecode abgelegt werden. Um das Beispielprogramm zu übersetzen, gibt man in der Kommandozeile im Verzeichnis P01 folgenden Befehl ein:

```
javac -d bin src/Demo1.java
```

Nach erfolgreicher Übersetzung des Quellcodes existiert im `bin`-Verzeichnis die Datei `Demo1.class`, die den Bytecode enthält. Zur Ausführung wird der Java-Interpreter wie folgt aufgerufen (mit der Option `cp` wird der so genannte Klassenpfad angegeben):

```
java -cp bin Demo13
```

Anwendung mit grafischer Oberfläche

```
import java.awt.Color;
import java.awt.Font;

import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Demo2 extends JFrame {
    public Demo2() {
        super("Ein erstes Beispiel");

        Icon icon = new ImageIcon(getClass().getResource("duke.gif"));
        JLabel label = new JLabel("Viel Erfolg mit dem Grundkurs Java", icon,
                               JLabel.CENTER);
```

³ Lage der Quellcode direkt im Verzeichnis P01, könnte wie in Abbildung 1-2 übersetzt und ausgeführt werden. Quellcode und Bytecode liegen dann im selben Verzeichnis.

```
add(label);

Font schrift = new Font("SansSerif", Font.BOLD, 20);
label.setFont(schrift);
label.setForeground(Color.red);
label.setBackground(Color.white);
label.setOpaque(true);

setDefaultCloseOperation(EXIT_ON_CLOSE);
setSize(500, 200);
setVisible(true);
}

public static void main(String[] args) {
    new Demo2();
}
}
```

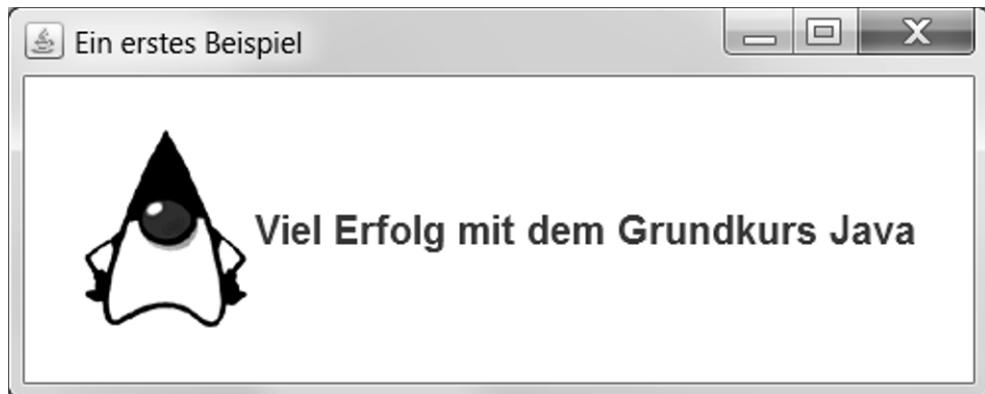


Abbildung 1-4: Grafische Ausgabe in einem Fenster

Hier wird der gleiche Text innerhalb eines Fensters mit einem kleinen Bild (Icon) ausgegeben. `duke.gif` muss im Verzeichnis `bin` liegen. Dieses Programm wird genauso wie das vorhergehende übersetzt und gestartet.

Die ausführliche Erläuterung der einzelnen Programmanweisungen ist den folgenden Kapiteln vorbehalten.

1.4 Wichtige Merkmale der Programmiersprache Java

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, Vererbung und Polymorphie.
- Die Sprache ist bewusst einfach gehalten. Im Unterschied zu C++ gibt es in Java z. B. keine expliziten Zeiger und keine Mehrfachvererbung.
- Java ist eine stark *typisierte Sprache*. Bereits bei der Übersetzung in den Bytecode werden Datentypüberprüfungen ausgeführt und Typinkonsistenzen erkannt.
- Zur Zeichendarstellung nutzt Java den internationalen *Unicode*-Standard.
- Die in einem Programm benutzten Klassen können als Dateien an unterschiedlichen Orten liegen. Sie werden erst zur Laufzeit des Programms bei Bedarf geladen.
- Das Speichermanagement in Java erfolgt automatisch. Während der Laufzeit eines Programms kann der Speicherplatz für nicht mehr benötigte Objekte vom Laufzeitsystem freigegeben werden (*Garbage Collection*).
- In Java gibt es eine strukturierte Behandlung von Laufzeit-Fehlern (*Exception-Handling*), 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-Klassenbibliothek bietet eine Reihe einfacher Möglichkeiten für die Netzwerkkommunikation auf Basis des Protokolls *TCP/IP*.
- Die Java-Klassenbibliotheken stellen darüber hinaus eine Vielzahl nützlicher APIs (*Application Programming Interfaces*) in Form von Klassen und Interfaces für die Anwendungsentwicklung zur Verfügung.

1.5 Zielsetzung und Gliederung des Buches

Dieser Grundkurs bietet eine strukturierte und anschauliche Einführung in grundlegende Aspekte der Java-Programmierung auf der Basis der Standard-Edition Java SE 8. Kennzeichnende Eigenschaften der Objektorientierung, wie Klassendefinition, Vererbung und Polymorphie, werden ausführlich dargestellt. Das Buch kann nicht die gesamte Java-Klassenbibliothek, die mehrere tausend Klassen umfasst, vorstellen. Nur 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 zur Java Standard Edition (siehe Quellenverzeichnis am Ende des Buches).

Obwohl dieser Kurs keine Erfahrung in der Programmierung voraussetzt, erleichtern Grundkenntnisse in einer anderen Programmiersprache wie z. B. Visual Basic, C oder C++ den Einstieg.

Die Kapitel des Buches bauen aufeinander auf und sollten deshalb bis auf die Exkurse in den Kapitel 15 und 16 in der folgenden Reihenfolge erarbeitet werden:

KAPITEL 1

gibt eine Übersicht über allgemeine Eigenschaften von Java und die Entwicklungs-umgebung.

KAPITEL 2

beschäftigt sich mit den imperativen (nicht-objektorientierten) Sprachkonzepten, wie z. B. einfache Datentypen, Operatoren und Kontrollstrukturen.

KAPITEL 3

führt die objektorientierten Sprachkonzepte (wie z. B. Klassen, Objekte, Methoden, Konstruktoren, Vererbung, Polymorphie, abstrakte Klassen, Interfaces, innere Klassen und Aufzählungstypen) ein, deren Verständnis grundlegend für alles Weitere ist.

KAPITEL 4

widmet sich der Behandlung von Ausnahmesituationen (Exceptions), die während der Programmausführung auftreten können.

KAPITEL 5

stellt einige nützliche Klassen der Klassenbibliothek vor, die in Anwendungen häufig verwendet werden.

KAPITEL 6

führt in das Thema Generics (generische Typen und Methoden) ein und präsentiert einige oft benutzte Interfaces und Klassen des Collection Frameworks.

KAPITEL 7

behandelt die mit Java SE 8 eingeführten Lambda-Ausdrücke, die im gewissen Sinne eine "funktionale Programmierung" ermöglichen.

KAPITEL 8

enthält die für den Zugriff auf Dateien (byte- und zeichenorientierte Datenströme) wichtigen Klassen und Methoden.

KAPITEL 9

bietet eine Einführung in die Programmierung mehrerer gleichzeitig laufender Anweisungsfolgen (Threads) innerhalb eines Programms.

KAPITEL 10

befasst sich mit der Entwicklung von grafischen Oberflächen sowie der Behandlung von Ereignissen, wie z. B. die Auswahl eines Menüpunkts oder das Anklicken eines Buttons. Hierzu stehen zahlreiche GUI-Komponenten zur Verfügung.

KAPITEL 11

behandelt Applets, die in HTML-Seiten eingebunden werden und dann unter der Kontrolle des Webbrowsers laufen.

KAPITEL 12

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

KAPITEL 13

beschäftigt sich mit der Programmierung von Client-Server-Anwendungen auf der Basis von TCP/IP und der Socket-Schnittstelle. Hier wird u. a. ein einfacher HTTP-Server entwickelt, der beliebige SQL-Anweisungen verarbeitet.

KAPITEL 14

enthält ein etwas umfangreicheres Fallbeispiel, das Konzepte, Methoden und Verfahren aus früheren Kapiteln verwendet.

KAPITEL 15

bietet eine Einführung in das Java Persistence API (JPA), das einen objekt-orientierten Zugriff auf relationale Datenbanken ermöglicht. JPA wurde als Teil der Spezifikation im Rahmen von Java EE definiert, kann aber auch ohne Verwendung eines Applikationsservers in einer Java-SE-Umgebung genutzt werden.

KAPITEL 16

zeigt, wie Objekte direkt – ohne Umweg über eine objektrelationale Abbildung – in einer objektorientierten Datenbank gespeichert werden können. Benutzt wird das objektorientierte Datenbanksystem db4o.

1.6 Programm- und Aufgabensammlung

Zahlreiche Beispielprogramme helfen bei der Umsetzung der Konzepte in lauffähige Anwendungen. Jedes Kapitel (mit Ausnahme von Kapitel 1 und 14) enthält am Ende Aufgaben, die den behandelten Stoff einüben und vertiefen.

Alle Programme wurden mit dem JDK für die Java Standard Edition 8 unter Windows 7 getestet.

Sämtliche Programme und Lösungen zu den Aufgaben stehen zum Download zur Verfügung. Hinweise zu weiteren Tools und Klassenbibliotheken erfolgen in den Kapiteln, in denen sie erstmalig benutzt werden. Das Quellenverzeichnis am Ende des Buches enthält die Bezugsquellen.

Den Zugang zum Begleitmaterial finden Sie auf der Website des Verlags

www.springer-vieweg.de

bei den bibliographischen Angaben zu diesem Buch.

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

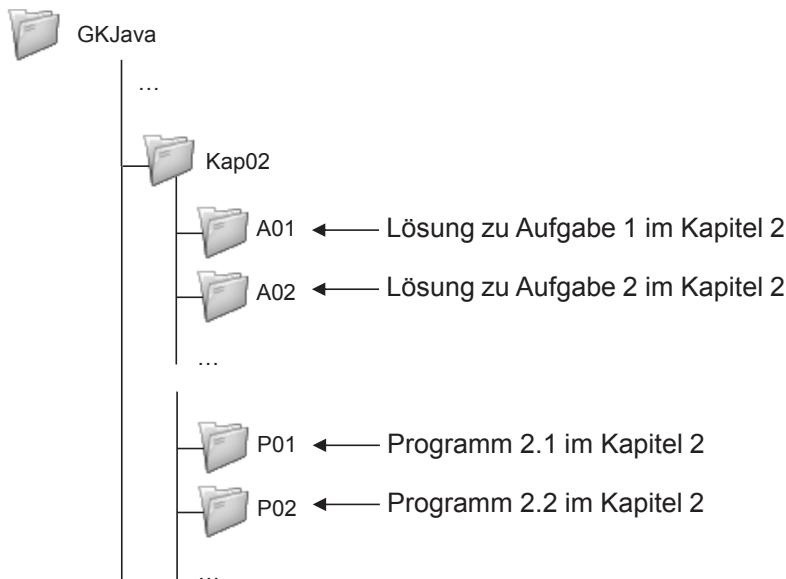


Abbildung 1-5: Ausschnitt aus der Ablagestruktur ⁴

Der Quellcode ist jeweils im Unterverzeichnis `src` gespeichert, also z. B.
GKJava/Kap02/P01/src/VarTest.java

⁴ Setzt man Eclipse ein, so empfiehlt sich, pro Kapitel einen Workspace einzurichten.

2 Imperative Sprachkonzepte

Dieses Kapitel beschreibt die *imperativen Sprachkonzepte* von Java. Insbesondere werden einfache Datentypen, Variablen, Operatoren und Anweisungen zur Ablaufsteuerung (so genannte Kontrollstrukturen) behandelt. Im Quellcode wird festgelegt, was in welcher Reihenfolge wie getan werden soll (lat. *imperare* = anordnen, befehlen).

Mitunter ist es in den Beispielen unerlässlich, weitere Sprachelemente (z. B. `class`) zu benutzen, um lauffähige Programme zu erhalten. Diese Elemente werden aber erst in den nachfolgenden Kapiteln ausführlich behandelt. Die Verständlichkeit des vorliegenden Kapitels wird hierdurch nicht beeinträchtigt.

Lernziele

In diesem Kapitel lernen Sie

- welche Datentypen zur Speicherung von Zeichen und Zahlen in Java existieren,
- mit welchen Operatoren Berechnungen, Vergleiche und Bedingungen formuliert werden können,
- wie mit Hilfe von Kontrollstrukturen der Ablauf eines Programms gesteuert werden kann.

2.1 Kommentare und Bezeichner

Kommentare im Quellcode sind frei formulierte Texte, die dem Leser hilfreiche Hinweise geben können. Sie können Zeichen einer speziellen Teilmenge des Unicode-Zeichensatzes enthalten und werden vom Compiler ignoriert.

Unicode ist ein standardisierter Zeichensatz, mit dem insbesondere die Schriftzeichen aller gängigen Sprachen dargestellt werden können. Die Zeichen der oben erwähnten *Teilmenge* sind jeweils 16 Bit lang, was demnach 65.536 verschiedene Zeichen ermöglicht. Die ersten 128 Zeichen sind die üblichen 7-Bit-ASCII-Zeichen.

Kommentare

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 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.

Bezeichner

Bezeichner sind Namen für vom Programmierer definierte Elemente wie Variablen, Marken, Klassen, Interfaces, Aufzählungstypen, Methoden und Pakete. Sie können aus beliebig vielen Unicode-Buchstaben und -Ziffern bestehen, müssen aber mit einem Unicode-Buchstaben beginnen. `_` und `$` 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 Literalen `true`, `false` und `null` übereinstimmen.

Die folgende Tabelle enthält die für Java reservierten 50 *Schlüsselwörter*. Sie dürfen nicht als Bezeichner verwendet werden:

Tabelle 2-1: Schlüsselwörter in Java

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

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.
- *Namen für Klassen, Interfaces und Aufzählungstypen* beginnen mit einem Großbuchstaben, z. B. `ErstesBeispiel`. Da Klassennamen als Teil des Namens der Datei auftauchen, die die entsprechende Klasse im Quell- bzw. Bytecode enthält, unterliegen diese auch den Regeln des jeweiligen Dateisystems.
- *Variablennamen* beginnen mit einem Kleinbuchstaben, z. B. `args`, `meinKonto`. Namen von *Konstanten* (Variablen mit unveränderbarem Wert) bestehen aus Großbuchstaben. Einzelne Wörter werden durch `_` getrennt, z. B. `KEY_FIRST`.
- *Methoden* werden in der Regel nach Verben benannt, ihre Namen beginnen ebenfalls mit einem Kleinbuchstaben, z. B. `berechne`, `zeichneFigur`.

2.2 Einfache Datentypen und Variablen

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.

Java kennt acht so genannte *einfache (primitive) Datentypen* (siehe Tabelle 2-2):

Literale

Feste Werte für einfache Datentypen, die direkt im Quelltext stehen, wie z. B. `12.345`, bezeichnet man als *Literale*.

Wahrheitswerte

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

Tabelle 2-2: Einfache Datentypen

Datentyp	Größe in Bit	Wertebereich
boolean	8	false, true
char	16	0 ... 65.535
byte	8	-128 ... 127
short	16	-32.768 ... 32.767
int	32	-2.147.483.648 ... 2.147.483.647
long	64	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
float	32	Absolutbetrag ca. $1,4 \cdot 10^{-45}$... $3,4028235 \cdot 10^{38}$, Genauigkeit ca. 7 Stellen
double	64	Absolutbetrag ca. $4,9 \cdot 10^{-324}$... $1,7976931348623157 \cdot 10^{308}$, Genauigkeit ca. 15 Stellen

Zeichen

Der *Zeichtyp* `char` dient dazu, einzelne Zeichen des Unicode-Zeichensatzes zu speichern. Literale werden in einfache Anführungszeichen eingeschlossen (z. B. `'a'`) und als Unicode-Zeichen oder als so genannte *Escape-Sequenzen* angegeben (siehe Tabelle 2-3).

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

Ä	<code>\u00c4</code>	ä	<code>\u00e4</code>	ß	<code>\u00df</code>
Ö	<code>\u00d6</code>	ö	<code>\u00f6</code>		
Ü	<code>\u00dc</code>	ü	<code>\u00fc</code>		

Ganze Zahlen

Die *ganzzahligen Typen* `byte`, `short`, `int`, `long` sind vorzeichenbehaftet. Literale können in Dezimal-, Binär- (ab Java SE 7), 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 oktalen 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. In diesem Fall sind sie vom Typ `long`. Der Typ `char` stellt auch einen ganzzahligen Typ dar und ist `short` gleichgestellt.

Tabelle 2-3: Escape-Sequenzen

Escape-Sequenz	Bedeutung
\b	Backspace
\t	Tabulator
\n	neue Zeile (Newline)
\f	Seitenvorschub (Formfeed)
\r	Wagenrücklauf (Carriage return)
\“	doppeltes Anführungszeichen "
\‘	einfaches Anführungszeichen '
\\\	Backslash \
\ddd	ASCII-codiertes Zeichen in Oktalschreibweise
\udddd	Unicode-Zeichen in Hexadezimalschreibweise (z. B. \u0020 für das Leerzeichen)

Fließkommazahlen / Gleitkommazahlen

Literale der *Fließkommatypen* `float` und `double` werden in Dezimalschreibweise notiert. Sie bestehen aus einem Vorkomma teil, einem Dezimalpunkt, einem Nachkommateil, 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 Nachkommateil 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 ab Java SE 7 in allen numerischen Literalen zur Erhöhung der Lesbarkeit eingefügt werden, z. B.

```
double y = 3.141_592_653;
```

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, 1 oder L.

Zeichenketten

Konstante *Zeichenketten* erscheinen in doppelten Anführungszeichen.

Beispiel: "Das ist eine Zeichenkette"

Einen einfachen Datentyp für Zeichenketten gibt es allerdings nicht, sondern die Klasse *String*, über die Zeichenketten erzeugt werden können. Mit dem Operator + können Zeichenketten aneinandergehängt werden.¹

Variablen

Die Definition einer Variablen erfolgt in der Form

```
Typname Variablenname;
```

Hierdurch wird Speicherplatz eingerichtet. Der Variablen kann gleich 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;
```

Im folgenden Beispiel 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.

Programm 2.1

```
public class VarTest {
    public static void main(String[] args) {
        // Variablendefinition mit Initialisierung
        boolean b = true;
        char c = 'x';
        int i = 4711;
        double d = 0.12345e1;
        int x = 0b01011111_10101010_00001111_11110000;
        double y = 3.141_592_653;

        // Ausgabe der Variablenwerte
        System.out.println("b: " + b);
        System.out.println("c: " + c);
        System.out.println("i: " + i);
        System.out.println("d: " + d);
        System.out.println("x: " + x);
        System.out.println("y: " + y);
    }
}
```

¹ Die Klasse *String* wird in Kapitel 5 ausführlich beschrieben.

Ausgabe des Programms:

```
b: true
c: x
i: 4711
d: 1.2345
x: 1604980720
y: 3.141592653
```

2.3 Ausdrücke und Operatoren

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

Ein *Ausdruck* 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.

Arithmetische Operatoren

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

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

Tabelle 2-4: Arithmetische Operatoren

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

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`.

Programm 2.2

```
public class ArithmOp {
    public static void main(String[] args) {
        int a = 5;

        System.out.println(13 / 5);
        System.out.println(13 % 5);

        System.out.println(12. / 2.5);
        System.out.println(12. % 2.5);

        System.out.println(++a + " " + a);
        System.out.println(a++ + " " + a);

        System.out.println(--a + " " + a);
        System.out.println( a-- + " " + a);
    }
}
```

Ausgabe des Programms:

```
2
3
4.8
2.0
6 6
6 7
6 6
6 5
```

Relationale Operatoren

Relationale Operatoren vergleichen Ausdrücke mit numerischem Wert miteinander. Das Ergebnis ist vom Typ `boolean`. Bei Fließkommawerten sollte die Prüfung auf exakte Gleichheit oder Ungleichheit vermieden werden, da es bei Berechnungen zu Rundungsfehlern kommen kann und die erwartete Gleichheit oder Ungleichheit nicht zutrifft. Stattdessen sollte mit den Operatoren `<` und `>` gearbeitet werden, um die Übereinstimmung der Werte bis auf einen relativen Fehler zu prüfen.

Tabelle 2-5: Relationale Operatoren

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

Logische Operatoren

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

Tabelle 2-6: Logische Operatoren

Operator	Bezeichnung	Priorität
!	NICHT	1
&	UND mit vollständiger Auswertung	7
^	exklusives ODER (XOR)	8
	ODER mit vollständiger Auswertung	9
&&	UND mit kurzer Auswertung	10
	ODER mit kurzer Auswertung	11

UND und ODER gibt es in zwei Varianten. Bei der so genannten kurzen Variante (*short circuit*) wird der zweite Operand nicht mehr ausgewertet, wenn das Ergebnis des Gesamtausdrucks schon feststeht. Beispielsweise ist A 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 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 2-7: Verknüpfung von Wahrheitswerten

a	b	<code>a & b</code> <code>a && b</code>	<code>a ^ b</code>	<code>a b</code> <code>a b</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>

Programm 2.3

```
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

```

Bitoperatoren

Bitoperatoren arbeiten auf der Binärdarstellung ganzzahliger Operanden, also mit 8 (byte), 16 (short, char), 32 (int) oder 64 Bit (long).²

Tabelle 2-8: Bitoperatoren

Operator	Bezeichnung	Priorität
<code>~</code>	Bitkomplement	1
<code><<</code>	Linksschieben	4
<code>>></code>	Rechtsschieben	4
<code>>>></code>	Rechtsschieben mit Nachziehen von Nullen	4
<code>&</code>	bitweises UND	7
<code>^</code>	bitweises exklusives ODER	8
<code> </code>	bitweises ODER	9

`~a` entsteht aus `a`, indem alle Bit von `a` invertiert werden, d. h. `0` geht in `1` und `1` in `0` über.

² Informationen zum Binärsystem findet man unter
<http://de.wikipedia.org/wiki/Dualsystem>

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 << 2` hat den Wert 32

`8 >> 2` hat den Wert 2

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

Tabelle 2-9: 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

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 gegebenenfalls 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 2-10: Zuweisungsoperatoren

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

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

Programm 2.4 demonstriert die Wirkung der Bitoperatoren $\&$, $|$ und \wedge . 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 0 setzen), Setzen von Bits (Bit 1 setzen), Umschalten von Bits (aus 0 wird 1 und umgekehrt).

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

Programm 2.4

```
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 = 0b10010000;
        //      00111010
        x ^= y;
        System.out.println(x);
    }
}
```

Ausgabe des Programms:

160

175

58

Bedingungsoperator

Der Bedingungsoperator 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.

Beispiel:

Der Ausdruck `a < 0 ? -a : a` hat den Wert des Absolutbetrags von `a`, wenn `a` eine Zahl ist. Der Ausdruck `a < b ? b : a` ergibt das Maximum von `a` und `b`.

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.

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.

Programm 2.5

```
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
```

2.4 Kontrollstrukturen

Anweisungen stellen die kleinsten ausführbaren Einheiten eines Programms dar. Eine Anweisung kann eine Deklaration 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.

Block { ... }

Die geschweiften Klammern { und } fassen mehrere Anweisungen zu einem *Block* zusammen. Dieser kann auch keine oder nur eine Anweisung enthalten. Ein Block gilt als eine einzelne 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.

Verzweigungen

Verzweigungen erlauben die bedingte Ausführung von Anweisungen.

Die if-Anweisung tritt in zwei Varianten auf:

```
if (Ausdruck)  
    Anweisung
```

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.

Programm 2.6

```
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

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` (ab Java SE 7) oder ein Aufzählungstyp (`enum`) sein.³

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 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.

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.

³ `Character`, `Byte`, `Short` und `Integer` werden in Kapitel 5.2, `String` in Kapitel 5.1 und Aufzählungstypen in Kapitel 3.11 behandelt.

Programm 2.7

```
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");
        }
    }
}
```

Schleifen

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

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.

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.

Programm 2.8 addiert die Zahlen von 1 bis zu einer vorgegebenen Zahl.

Programm 2.8

```
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

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

$$\text{for } (\text{Init}; \text{Bedingung}; \text{Update}) \\
\quad \text{Anweisung}$$

Init ist eine Liste von durch Kommas voneinander getrennten Anweisungen oder Variablenklärungen 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.

Programm 2.9

```
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);
    }
}
```

Programm 2.10 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?

Programm 2.10

```
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
```

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
```

Es gibt eine Variante der `for`-Schleife: die so genannte `foreach`-Schleife. Mit ihr können in einfacher Form Elemente eines Arrays oder einer Collection durchlaufen werden.⁴

Sprunganweisungen

Sprunganweisungen werden hauptsächlich verwendet, um Schleifendurchgänge vorzeitig zu beenden.

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.

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;`

Programm 2.11

```
public class SprungTest {
    public static void main(String[] args) {
        Hauptschleife: 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 Hauptschleife;
                }
            }
        }
    }
}
```

⁴ Siehe Kapitel 3.10, 5.4.1, 6.7.1

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
```

2.5 Aufgaben

1. Welche der folgenden Bezeichner sind ungültig?

```
Hello_Welt $Test _abc 2test
#hallo te?st Girokonto const
```

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

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

3. Schreiben Sie ein Programm, das mit einem einzigen Aufruf von `System.out.print` eine Zeichenkette in mehreren Bildschirmzeilen ausgibt.

4. 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.

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

6. Jetzt ist es x Uhr. Wieviel Uhr ist es in n Stunden? Schreiben Sie hierzu ein Programm.

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

8. 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?

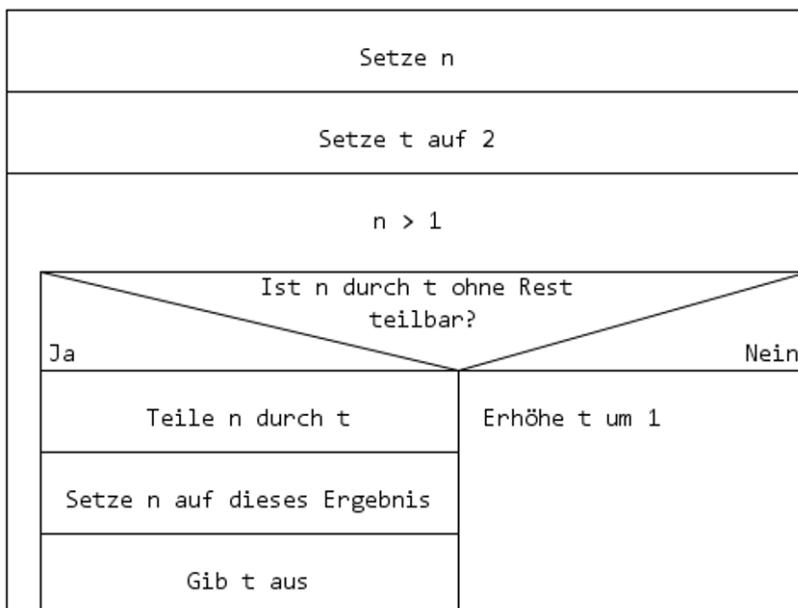
a) --a	b) a--	c) a++ + b	d) b = ++a
e) a = b++	f) -(a--) - -(--b)	g) a++ + ++a + a++	

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

10. Schreiben Sie ein Programm, das ermittelt, wie hoch ein Sparguthaben von 5.000 Geldeinheiten bei 7,5 % Verzinsung nach Ablauf eines Jahres ist.
11. 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.
12. 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 $n = 4$.
13. 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.
14. 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.
15. 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.
16. Schreiben Sie hierzu ein Programm.
 17. Schreiben Sie ein Programm, das zu einer Zahl $n \leq 20$ die Fakultät $n!$ ermittelt. Es gilt bekanntlich:
$$n! = 1 * 2 * \dots * (n-1) * n \quad \text{sowie} \quad 0! = 1$$
 18. 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.
19. Schreiben Sie ein Programm, das eine Treppe aus h Stufen einer bestimmten Breite b in der folgenden Form zeichnet (Ausgabe von Leerzeichen und "*" auf der Konsole):

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

19. Schreiben Sie ein Programm, das die Primfaktorenzerlegung einer Zahl $n \geq 2$ berechnet. Die folgende Abbildung zeigt den Entwurf des Programms in Form eines Strukturdiagramms:



3 Objektorientierte Sprachkonzepte

Dieses Kapitel behandelt die wesentlichen objektorientierten Sprachelemente von Java. Im Vordergrund stehen die Begriffe *Klasse*, *Objekt*, *Attribut*, *Methode*, *Vererbung* und *Interface*. Die objektorientierte Programmierung löst sich von dem Prinzip der Trennung von Daten und Funktionen, wie sie in den traditionellen (prozeduralen) Programmiersprachen der dritten Generation 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 Klassenvariablen und -methoden,
- wie von bestehenden Klassen neue Klassen abgeleitet werden können,
- wie abstrakte Klassen eingesetzt werden,
- was man unter Polymorphie versteht,
- wie Interfaces zur Trennung von Schnittstelle und Implementierung verwendet werden,
- welche Arten von inneren Klassen existieren und wie solche sinnvoll genutzt werden können,
- den Umgang mit Arrays und Aufzählungen,
- wie Klassen und Interfaces in Paketen organisiert werden können.

3.1 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 Bestandteile einer Klasse. 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 Attributwerte und damit den Zustand des Objekts ändern. Konstruktoren sind spezielle Methoden, die bei der Objekterzeugung aufgerufen werden.

Klasse

Der allgemeine Aufbau einer *Klasse* ist:

```
[Modifizierer] class Klassenname [extends Basisklasse][implements  
Interface-Liste] {  
  
    Attribute  
    Konstruktoren  
    Methoden  
    ...  
}
```

Die in eckigen Klammern angegebenen Einheiten sind optional und werden an späterer Stelle ausführlich erklärt. Des Weiteren können im Rumpf der Klasse noch so genannte Initialisierer, innere Klassen und Interfaces definiert werden (siehe spätere Abschnitte).

Beispiel: 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, zwei Methoden, die den Saldo erhöhen bzw. vermindern, sowie eine Methode, die die Attributwerte ausgibt.¹

Die Klasse `Konto` ist mit den Modifizierern `public` und `private` ausgestattet. Diese werden erst an späterer Stelle ausführlich beschrieben. Kurz: 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 Modifizierer² `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`.

Konto	← Klassenname
-kontonummer: int -saldo: double	← Attribute
+getKontonummer(): int +setKontonummer(int nr): void +getSaldo(): double +setSaldo(double betrag): void +zahleEin(double betrag): void +zahleAus(double betrag): void +info(): void	← Methoden

Abbildung 3-1: Klasse Konto³

¹ Methoden werden ausführlich im Kapitel 3.2 behandelt.

² Modifizierer werden in Kapitel 3.7 erläutert.

³ - steht für `private`, + für `public`.

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.⁴

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 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 (siehe Kapitel 2.2) 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` und für Referenzvariablen `null`.

⁴ Siehe auch die Methode `equals` in Kapitel 5.3.

Punktnotation

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

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

Zum Programm 3.1 gehören die Datei `KontoTest.java` und die Datei `Konto.java`, die die Definition der Klasse `Konto` aus dem oben angeführten Beispiel enthält.

Programm 3.1

```
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

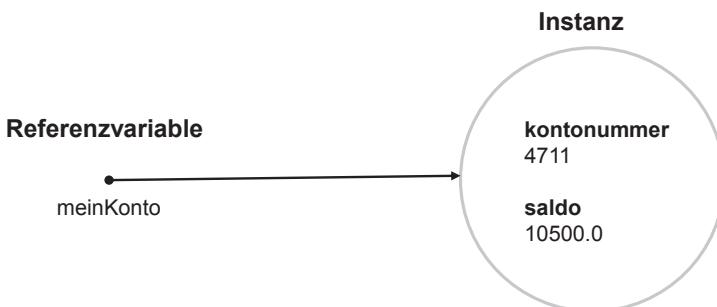


Abbildung 3-2: Objektreferenz

Wären die Attribute der Klasse `Konto` vor dem Zugriff von außen 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 (siehe Kapitel 3.7), sodass sie – wie im Programm 3.1 – nur von den öffentlichen Methoden der Klasse verändert werden können.

3.2 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 (siehe obige Definition der Klasse Konto).

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 Einheiten sind optional.⁵

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 der 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.

⁵ *Modifizierer* und *throws*-Klausel werden an späterer Stelle erklärt.

call by value / call by reference

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. Diese Art des Methodenaufrufs, bei der eine Referenz übergeben wird, wird auch als "*call by reference*" bezeichnet.

Programm 3.2 veranschaulicht den Parametermechanismus.

Programm 3.2

```
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 Initialisierungssteil 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 übereinstimmen. Definitionen lokaler Variablen können mit anderen Anweisungen gemischt werden.

Lokale final-Variablen

Beginnt die Deklaration 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 Rückgabetyp gehört nicht zur Signatur. Um eine Methode einer Klasse in einem Programm aufrufen zu können, muss nur ihre Signatur, ihr Rückgabetyp und ihre semantische Wirkung (z. B. in Form einer umgangssprachlichen Beschreibung) bekannt sein. Wie die Methode im Methodenkörper implementiert ist, interessiert nicht (*Black Box*).

Überladen

Es können mehrere Methoden mit gleichem Namen, aber unterschiedlichen Signaturen deklariert werden. Der Compiler entscheidet beim Methodenaufruf, welche der deklarierten 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.

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
```

Programm 3.3

```
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
```

3.3 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 deklariert und kann auch überladen werden.

Standardkonstruktor

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

Beispiel:

Für die Klasse Konto werden vier Konstruktoren deklariert. 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 *Kopier-Konstruktor* bezeichnet.

```
public Konto() { }

Konto(int kontonummer) {
    this.kontonummer = kontonummer;
}

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

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

Programm 3.4

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

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

        new Konto(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
```

Programm 3.4 zeigt auch, dass Objekte ohne eigene Referenzvariable erzeugt werden können (anonyme Objekte). Die Methode `info` wird hier direkt für das Konto-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 Konto(int kontonummer) {  
    this(kontonummer, 0.);  
}
```

Initialisierungsblock

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

Die verschiedenen Elemente eines Programms werden in unterschiedlichen Bereichen des Hauptspeichers verwaltet. Im *Stack* werden lokale Variablen und Methodenparameter verwaltet. Im *Heap* liegen die erzeugten Objekte einer Klasse. Im Methodenspeicher wird die Präsentation jeder geladenen Klasse verwaltet.

Garbage Collector

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

3.4 Klassenvariablen und Klassenmethoden

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

Klassenvariable

Ein Attribut einer Klasse, dessen Definition mit dem Schlüsselwort `static` versehen ist, nennt man *Klassenvariable*. Klassenvariablen (auch *statische Variablen* genannt) werden 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 kann über den Klassennamen in der Form:

Klassenname.Variablenname

erfolgen. Der Zugriff über ein Objekt der Klasse ist ebenfalls möglich.

Instanzvariable

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

Mit Klassenvariablen können beispielsweise Instanzenzähler realisiert werden, wie das folgende Beispiel zeigt.

Programm 3.5

```
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);  
    }  
}
```

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

Klassenmethode

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(...) { ... }`

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. Der Parameter der Methode `main` wird an späterer Stelle erklärt. Ein Klasse, die eine `main`-Methode enthält, nennt man auch *ausführbare Klasse*.

Programm 3.6

```
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 Methodendeklarationen 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.

3.5 Vererbung

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

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* oder *abgeleitete Klasse*, die andere als *Superklasse* oder *Basisklasse* bezeichnet.

Die Subklasse kann in ihrer Implementierung über alle Attribute und Methoden ihrer Superklasse verfügen (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.⁶ Konstruktoren werden nicht vererbt.

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

Überschreiben

Beim *Überschreiben* (Overriding) einer Methode bleibt ihre Signatur (Name, Parameterliste) unverändert. Der Rückgabetyp darf vom Rückgabetyp der überschriebenen Methode abgeleitet sein. Dies gilt für Referenztypen, einfache Datentypen und `void` bleiben unverändert. Die Anweisungen im Methodenkörper werden geändert, um so auf die Besonderheiten der Subklasse als Spezialisierung der Superklasse eingehen zu können.

Geerbte Instanzmethoden können nur durch Instanzmethoden, geerbte Klassenmethoden nur durch Klassenmethoden überschrieben werden.

Beispiel: Abgeleitete Klasse Girokonto

```
public class Girokonto extends Konto {  
    private double limit;  
  
    public Girokonto(int kontonummer, double saldo, double limit) {
```

⁶ Sofern die Zugriffsrechte es erlauben, siehe Kapitel 3.7.

```

super(kontonummer, saldo);
this.limit = limit;
}

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

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

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

```

Hier erbt die Klasse `Girokonto` die Methoden der Klasse `Konto`. 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.

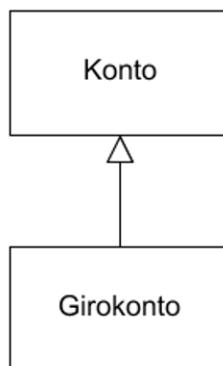


Abbildung 3-3: Girokonto ist abgeleitet von Konto

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 deklariert, 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 initialisiert werden.

Konstruktoren

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.

Zu folgendem Programm gehören `GirokontoTest.java`, `Konto.java` (aus Programm 3.4) und `Girokonto.java`.

Programm 3.7

```
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
```

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.⁷

Programm 3.8 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.

Programm 3.8

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

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

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

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

⁷ Methoden von `Object` werden in Kapitel 5.3 erläutert.

```
    }  
  
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
```

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.

Programm 3.9

```
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
```

Programm 3.9 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. Die *tatsächliche Klasse des Objekts* bestimmt also, 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.

Methodenauswahl zur Laufzeit

Allgemein gilt:

Um die auszuführende Methode (`m`) zu bestimmen, beginnt Java zur Laufzeit in der Klasse (`A`), mit deren Typ die Referenzvariable definiert ist, und sucht dann entlang der Vererbungshierarchie in Richtung der Subklasse (`C`) des referenzierten Objekts die letzte überschreibende Methode.

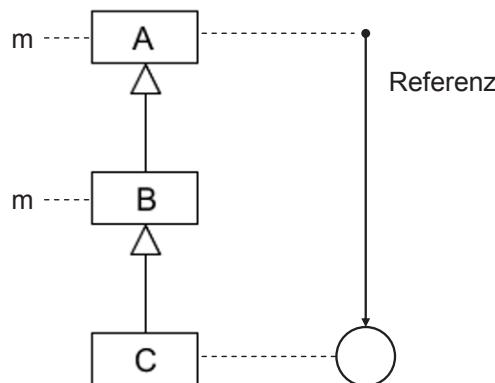


Abbildung 3-4: Ermittlung der auszuführenden Methode

`instanceof`

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.

3.6 Abstrakte Klassen

Klassen, deren Definition mit dem Schlüsselwort `abstract` beginnt, heißen *abstrakte Klassen*. Von solchen Klassen können keine Objekte erzeugt werden.

In abstrakten Klassen dürfen so genannte *abstrakte Methoden* auftreten. Abstrakte Methoden besitzen keinen Methodenkörper. Sie müssen in Subklassen überschrieben werden, andernfalls sind die abgeleiteten Klassen ebenfalls abstrakt und müssen 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:

Programm 3.10

```
public abstract class Zeit {  
    public abstract long getMinuten();  
  
    public long getSekunden() {  
        return getMinuten() * 60;  
    }  
}  
  
public class Tage extends Zeit {  
    private long tage;  
  
    public Tage(long tage) {  
        this.tage = tage;  
    }  
  
    public long getMinuten() {  
        return tage * 24 * 60;  
    }  
}  
  
public class StundenMinuten extends Zeit {  
    private long stunden;  
    private long minuten;
```

```

public StundenMinuten(long stunden, long minuten) {
    this.stunden = stunden;
    this.minuten = minuten;
}

public long getMinuten() {
    return stunden * 60 + minuten;
}
}

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());
    }
}

```

Die abstrakte Klasse `Zeit` implementiert die Methode `getSekunden` und bedient sich dabei der abstrakten Methode `getMinuten`. 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. Der Typ der Referenzvariablen legt also nicht fest, auf welchen Typ von Objekt sie zur Laufzeit zeigt.

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*). Der Einsatz dieses Mechanismus vereinfacht die Programmentwicklung wesentlich.

3.7 Modifizierer

Mit Hilfe von so genannten *Modifizierern* können bestimmte Eigenschaften von Klassen, Attributen und Methoden festgelegt werden. Die Tabelle 3-1 führt die Modifizierer auf und zeigt, bei welchen Sprachelementen diese verwendet werden können.⁸

Tabelle 3-1: Modifizierer im Überblick

Modifizierer	Klasse	Attribut	Methode	Konstruktor
public	x	x	x	x
protected		x	x	x
private		x	x	x
static		x	x	
final	x	x	x	
abstract	x		x	
native			x	
synchronized			x	
transient		x		
volatile		x		

Zugriffsrechte

Zugriffsrechte für Klassen, Attribute, Methoden und Konstruktoren werden über die Modifizierer **public**, **protected** und **private** gesteuert. In diesem Zusammenhang ist der Begriff des *Pakets*, der später ausführlich behandelt wird, von Bedeutung. Ein Paket ist eine Sammlung von Klassen, die aus bestimmten Gründen zusammengefasst werden (z. B. weil sie alle zu einem bestimmten Projekt gehören). Jede Klasse gehört zu genau einem Paket.⁹

Sind weder **public**, **protected** noch **private** angegeben, so können die Klassen, Attribute, Methoden und Konstruktoren in allen Klassen desselben Pakets benutzt werden.

public, protected, private

Klassen, die mit **public** gekennzeichnet sind, können überall (auch in anderen Paketen) genutzt werden. In einer Quelldatei darf höchstens eine Klasse als **public**

⁸ **static** und **abstract** wurden bereits in Kapitel 3.4 bzw. 3.6 erläutert. **synchronized** wird im Zusammenhang mit Threads im Kapitel 9.2 und **volatile** im Kapitel 9.1 behandelt. **transient** wird im Zusammenhang mit der Serialisierung von Objekten im Kapitel 8.9 behandelt.

⁹ Siehe Kapitel 3.12.

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 `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 (*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.

3.8 Interfaces

Interfaces sind Schnittstellenbeschreibungen, die festlegen, was man mit der Schnittstelle machen kann.

Bis zur Version Java SE 7 können Interfaces nur Konstanten, abstrakte Methoden sowie geschachtelte Klassen und Interfaces enthalten.¹⁰

Sie haben den folgenden Aufbau:

```
[public] interface Interfacename [extends Interface-Liste] {  
    Konstanten  
    abstrakte Methoden  
    geschachtelte Klassen und Interfaces  
}
```

Die in eckigen Klammern angegebenen Einheiten sind optional. Interfacenamen unterliegen den gleichen Namenskonventionen wie Klassennamen. Die Konstanten haben implizit die Modifizierer `public`, `static` und `final`. Die Methoden sind alle implizit `public` und `abstract`. Diese automatisch vergebenen Modifizierer sollten nicht explizit gesetzt werden. Geschachtelte Klassen sind automatisch `public` und `static`.¹¹

extends

Interfaces können im Gegensatz zu Klassen von mehreren anderen Interfaces abgeleitet werden. Damit können mehrere Schnittstellenbeschreibungen in einem Interface zusammengefasst und neue Konstanten und Methoden hinzugefügt werden. Die Interfacenamen werden nach `extends`, durch Kommas voneinander getrennt, aufgeführt. Ein Interface erbt alle Konstanten und alle Methoden seiner Superinterfaces.

Klassen implementieren Interfaces

Klassen können ein oder mehrere Interfaces mit Hilfe des Schlüsselwortes `implements` implementieren. Mehrere Interfacenamen werden durch Kommas voneinander getrennt. Jede nicht abstrakte Klasse, die ein Interface implementiert, muss *alle* abstrakten Methoden des Interfaces implementieren. Dabei kann die Implementierung einer Interfacemethode auch von einer Superklasse geerbt werden.

Im Falle einer Klasse als Rückgabetyp einer Methode des Interfaces darf der Rückgabetyp der implementierten Methode eine Subklasse dieser Klasse sein. Dies gilt analog auch für Interfaces als Rückgabetyp. Hier darf dann der Rückgabetyp ein Subinterface sein.

¹⁰ Ab Java SE 8 können Interfaces auch so genannte Default-Methoden sowie statische Methoden implementieren. Hierauf gehen wir in einem späteren Abschnitt ein.

¹¹ Siehe Kapitel 3.9.

Haben verschiedene Konstanten der implementierten Interfaces den gleichen Namen, so muss der Zugriff auf diese durch Qualifizierung mit dem betreffenden Interfacenamen erfolgen:

Interfacename.Konstantenname

Die Implementierung zweier Interfaces, die Methoden mit gleicher Signatur, aber unterschiedlichen Rückgabetypen haben, ist nicht möglich.

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 Interfaces sein. Eine Klasse, die ein Interface implementiert, ist auch vom Typ des Interfaces.

Eine Variable vom Typ des Interfaces X kann auf ein Objekt verweisen, dessen Klasse das Interface X implementiert. Mit Hilfe dieser Referenzvariablen kann nur auf die in X deklarierten 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.

Der Zugriff auf Objekte mit Referenzvariablen vom Interface-Typ erleichtert den Austausch einer implementierenden Klasse gegen eine andere, die dasselbe Interface implementiert.

Programm 3.11 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.

Programm 3.11

```
public interface Geo {  
    double getFlaeche();  
}  
  
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;
    }
}

public class Rechteck implements Geo {
    private double breite;
    private double hoehe;

    public Rechteck(double breite, double hoehe) {
        this.breite = breite;
        this.hoehe = hoehe;
    }

    public double getFlaeche() {
        return breite * hoehe;
    }
}

public class Kreis implements Geo {
    private double radius;
    private static final double PI = 3.14159;

    public Kreis(double radius) {
        this.radius = radius;
    }

    public double getFlaeche() {
        return PI * radius * radius;
    }
}

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:

```
50.38950000000005
50.38866575757591
0
```

Es können 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 wartbar.

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 Interfaces definiert werden. Von außen kann dann voll qualifiziert auf sie zugegriffen werden:

Typname.InterfaceName

Programm 3.12

```
public interface X {  
    void x();  
  
    interface Y {  
        void y();  
    }  
}  
  
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");  
    }  
}
```

```
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();  
    }  
}
```

Default-Methoden

Ab der Version Java SE 8 können Interfaces neben abstrakten Methoden auch so genannte *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.

Programm 3.13

Default-Methoden können abstrakte Methoden des Interfaces verwenden. Klassen, die ein Interface implementieren, müssen wie bisher alle abstrakten Methoden implementieren, können die Default-Methoden aufrufen oder diese selbst neu implementieren.

```
public interface X {  
    void m1();  
  
    default void m2() {  
        System.out.println("Methode m2");  
    }  
  
    default void m3() {  
        System.out.println("Methode m3");  
        m1();  
    }  
}  
  
public class XImpl1 implements X {  
    public void m1() {  
        System.out.println("Methode m1");  
    }  
}
```

```
public static void main(String[] args) {
    X x = new XImpl1();
    x.m2();
    x.m3();
}

public class XImpl2 implements X {
    public void m1() {
        System.out.println("Methode m1");
    }

    public void m2() {
        System.out.println("Methode m2 neu implementiert");
    }

    public static void main(String[] args) {
        X x = new XImpl2();
        x.m2();
        x.m3();
    }
}
```

Programm 3.14

Default-Methoden konkurrieren mit ererbten Methoden.

```
public interface X {
    default void m() {
        System.out.println("Methode m aus X");
    }
}

public class A {
    public void m() {
        System.out.println("Methode m aus A");
    }
}

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 Interfaces X.

Fazit

Sobald eine Methode von einer Klasse (Superklasse in der Vererbungshierarchie) implementiert ist, werden entsprechende Default-Methoden ignoriert.

Programm 3.15

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.

```
public interface X {  
    default void m() {  
        System.out.println("Methode m aus X");  
    }  
}  
  
public interface Y extends X {  
    default void m() {  
        System.out.println("Methode m aus Y");  
    }  
}  
  
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.

Programm 3.16

Implementiert die 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 Interfacename in Kombination mit `super` benutzt (siehe Kommentar im folgenden Quellcode).

```

public interface X {
    default void m() {
        System.out.println("Methode m aus X");
    }
}

public interface Y {
    default void m() {
        System.out.println("Methode m aus Y");
    }
}

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 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 Interfaces genutzt werden.

Default-Methoden können bei der Weiterentwicklung von Software sehr hilfreich sein (API-Evolution), ein entscheidender Grund für ihre Einführung.

Programm 3.17

Die Klasse `LifecycleImpl` implementiert das Interface `Lifecycle`.

```

public interface Lifecycle {
    void start();
    void stop();
}

```

```
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.

Programm 3.18

```
public interface Lifecycle {  
    void start();  
    void stop();  
  
    default void init() {  
    }  
  
    default void destroy() {  
    }  
}  
  
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 die Abwärtskompatibilität.

static-Methoden in Interfaces

Ab Java SE 8 können Interfaces auch static-Methoden implementieren. Solche Methoden sind wie bei allen anderen Interface-Methoden automatisch public.

Programm 3.19

```
public interface X {  
    static void m() {  
        System.out.println("Methode m aus X");  
    }  
}  
  
public class A {  
    public static void main(String[] args) {  
        X.m();  
    }  
}  
  
public class B implements X {  
    public static void main(String[] args) {  
        B b = new B();  
  
        // Folgende Aufrufe sind nicht möglich:  
        // b.m();  
        // B.m();  
  
        X.m();  
    }  
}
```

Auch wenn die Klasse 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.

3.9 Innere Klassen

Innere Klassen sind Klassen, die *innerhalb* einer bestehenden Klasse vereinbart werden. Solche Klassen werden intensiv bei der Ereignisbehandlung im Rahmen der Entwicklung von grafischen Oberflächen verwendet. Hilfsklassen können nahe bei den Programmstellen vereinbart werden, an denen sie gebraucht werden.

Wir behandeln im Folgenden vier Arten von inneren Klassen:

- statische Klassen
- Instanzklassen
- lokale Klassen
- anonyme Klassen

Statische Klasse

Eine *statische Klasse* B wird innerhalb einer Klasse A als static analog zu Klassenvariablen vereinbart. Ihr vollständiger Name lautet A.B.

Objekte von B können mit

```
A.B obj = new A.B(...);
```

erzeugt werden. Aus B heraus kann direkt auf Klassenvariablen und Klassenmethoden von A zugegriffen werden.

Statische Klassen bieten einen Strukturierungsmechanismus, um logisch aufeinander bezogene Klassen im Zusammenhang zu definieren. Statische Klassen werden wie Attribute vererbt. Sie können mit public, protected oder private in der üblichen Bedeutung gekennzeichnet werden.

Programm 3.20 veranschaulicht die Verwendung einer statischen Klasse. Die statische Klasse Permissions verwaltet die Zugriffsrechte eines Benutzers.

Programm 3.20

```
public class Account {
    private int userId;
    private Permissions perm;

    public Account(int userId) {
        this.userId = userId;
        perm = new Permissions();
    }

    public int getUserId() {
        return userId;
    }

    public static class Permissions {
        public boolean canRead;
        public boolean canWrite;
        public boolean canDelete;
    }

    public Permissions getPermissions() {
        return perm;
    }
}

public class Test {
    public static void main(String[] args) {
        Account account = new Account(4711);

        Account.Permissions perm = account.getPermissions();
        perm.canRead = true;

        System.out.println(perm.canRead);
        System.out.println(perm.canWrite);
        System.out.println(perm.canDelete);
    }
}
```

Instanzklasse

Objekte von *Instanzklassen* können nur im Verbund mit einer Instanz der sie umgebenden Klasse erzeugt werden. Sie werden in der Regel in Instanzmethoden der umgebenden Klasse erzeugt und haben Zugriff auf alle Attribute und Methoden dieser äußeren Klasse. Instanzklassen können keine statischen Klassen, statischen Attribute oder statischen 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.

Programm 3.21 veranschaulicht die Verwendung einer Instanzklasse. Die Klasse `Konto` verwaltet die letzte für das Konto durchgeführte Transaktion (Ein- oder Auszahlung) in einem Objekt einer Instanzklasse. Mit Hilfe der Punktnotation kann der Typname `Transaktion` außerhalb von `Konto` benutzt werden.

Programm 3.21

```
public class Konto {  
    private int kontonummer;  
    private double saldo;  
    private Transaktion last;  
  
    public Konto(int kontonummer, double saldo) {  
        this.kontonummer = kontonummer;  
        this.saldo = saldo;  
    }  
  
    public class Transaktion {  
        private String name;  
        private double betrag;  
  
        public Transaktion(String name, double betrag) {  
            this.name = name;  
            this.betrag = betrag;  
        }  
  
        public String toString() {  
            return kontonummer + ":" + name + " " + betrag + ", 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);  
    }  
}
```

```

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());
    }
}

```

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. Der Code kann außerdem auf alle lokalen `final`-Variablen eines umgebenden Blocks und alle `final`-Parameter eines umgebenden Methodenrumpfs zugreifen. Ab Java SE 8 muss eine lokale Variable, die zur Laufzeit ihren Wert nicht ändert, nicht mehr explizit mit `final` gekennzeichnet werden.

Die Instanz der äußeren Klasse A kann mit `A.this` explizit referenziert werden.

Programm 3.22 verwendet innerhalb der Methode `iterator` der Klasse `Liste` eine lokale Klasse. Die Klasse `Liste` verwaltet beliebige Objekte vom Typ `Object` in einer so genannten *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.

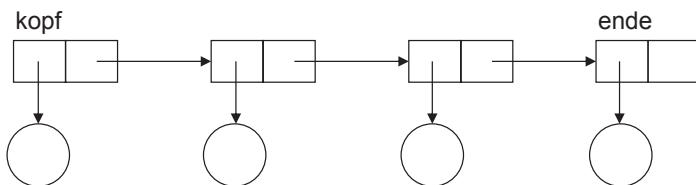


Abbildung 3-5: Verkettete Liste

Programm 3.22

```

public interface Iterator {
    boolean hasNext();

    Object next();
}

```

```
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();
    }
}

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);
        }
    }
}
```

Die Methode `iterator` liefert ein Objekt vom Typ des Interfaces `Iterator`, mit Hilfe dessen Methoden die verkettete Liste bequem durchlaufen werden kann.

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 Interfaces. 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 SE 8 muss eine lokale Variable, die zur Laufzeit ihren Wert nicht ändert, nicht mehr explizit mit `final` gekennzeichnet werden. Die Instanz der äußeren Klasse A kann mit `A.this` explizit referenziert werden. Alle abstrakten Methoden der Basisklasse bzw. des Interfaces müssen implementiert werden.

Anstelle der lokalen Klasse im Programm 3.22 kann eine anonyme Klasse verwendet werden.

Programm 3.23

```
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;
        }
    };
}
}

```

Im Kapitel 7 wird gezeigt, wie Lambda-Ausdrücke (neu in Java SE 8) anonyme Klassen in vielen Fällen ersetzen und somit die Programmierung vereinfachen können.

3.10 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.

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 Defintion bereits fest, wie viele Elemente das Array aufnehmen soll, können beide Anweisungen zusammengefasst werden.

Beispiel:

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

Initialisierung

Die Elemente eines Arrays werden bei ihrer Erzeugung mit Standardwerten vorbesetzt, ebenso wie die Attribute eines Objekts. 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};
```

Diese letzte Form kann beispielsweise genutzt werden, um ein so genanntes anonymes Array als Argument beim Aufruf einer Methode zu übergeben:

```
methode(new int[] {1, 10, 4, 0});
```

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.

Länge

Die Größe eines Arrays kann erst zur Laufzeit festgelegt werden. Sie kann dann aber nicht mehr verändert werden. Die Anzahl der Elemente eines Arrays kann über das Attribut `length` abgefragt werden. Für das Array `x` im letzten Beispiel gilt: `x.length` hat den Wert 4.

Zugriff auf 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 Ausnahme `ArrayIndexOutOfBoundsException` ausgelöst (siehe Kapitel 4).

Programm 3.24

```
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] + " ");
        }
    }
}
```

Wenn beim Durchlaufen einer `for`-Schleife der Schleifenindex nicht benötigt wird, kann ab Java SE 5 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.

```
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. Referenzvariablen für solche Arrays werden erzeugt, indem mehrere Klammernpaare hintereinander angegeben werden. Der Zugriff auf ein Element erfolgt durch Angabe aller erforderlichen Indizes.

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

Beispiel:

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

erzeugt eine 2x3-Matrix.

Diese Anweisung hat dieselbe Wirkung wie:

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

Alternativ mit Initialisierung:

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

x.length hat den Wert 2.

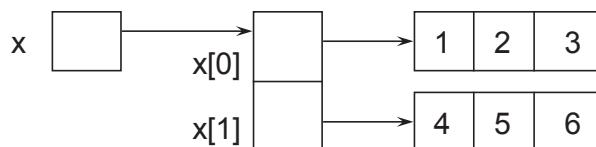


Abbildung 3-6: 2x3-Matrix

Bei der Initialisierung können für jedes Element einer Dimension unterschiedlich viele Elemente initialisiert werden. Das folgende Programm zeigt, wie "nicht rechteckige" Arrays erzeugt werden können.

Programm 3.25

```
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[] a : x) {
    for (int b : a) {
        System.out.print(b);
    }

    System.out.println();
}
}
```

Ausgabe des Programms:

```
1
12
123
1234
12345
...
```

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.

Programm 3.26

```
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 bin 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

Varargs

Ab Java SE 5 lassen sich mit *Varargs* (variable length argument lists) Methoden mit einer beliebigen Anzahl von Parametern desselben Typs definieren. Hierzu wird nur ein einziger Parameter, der so genannte 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.

Programm 3.27 zeigt die Methoden `sum` und `min`, die mit beliebig vielen `int`-Argumenten aufgerufen werden können.

Programm 3.27

```
public class VarargTest {
    public static void main(String[] args) {
        System.out.println(sum(1, 2));

        // äquivalent zum ersten Aufruf:
        System.out.println(sum(new int[] { 1, 2 }));

        System.out.println(sum(1, 10, 100, 1000));
        System.out.println(sum());

        System.out.println(min(3, 5, 2));
    }

    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;
}

```

Ausgabe des Programms:

```

3
3
1111
0
2

```

3.11 Aufzählungen

Eine *Aufzählung* ist ein Datentyp, dessen Wertebereich aus einer Gruppe von Konstanten besteht.

Beispiel:

Der Wertebereich des Aufzählungstyps `Ampelfarbe` wird durch die Aufzählung der Konstanten `ROT`, `GELB` und `GRUEN` beschrieben.

In Programm 3.28 werden die drei Ampelfarben als vordefinierte Konstanten vom Typ `int` in einer eigenen Klasse vereinbart.

Programm 3.28

```

public class Ampelfarbe {
    public static final int ROT = 0;
    public static final int GELB = 1;
    public static final int GRUEN = 2;
}

public class Test {
    public static void info(int farbe) {
        switch (farbe) {
            case Ampelfarbe.ROT:
                System.out.println(farbe + ": Anhalten");
                break;
            case Ampelfarbe.GELB:
                System.out.println(farbe + ": Achtung");
                break;
            case Ampelfarbe.GRUEN:
                System.out.println(farbe + ": Weiterfahren");
                break;
        }
    }
}

```

```
public static void main(String[] args) {
    info(Ampelfarbe.ROT);
    info(Ampelfarbe.GELB);
    info(Ampelfarbe.GRUEN);

    // unsinniger Argumentwert
    info(4711);
}
```

Ausgabe des Programms:

```
0: Anhalten
1: Achtung
2: 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`.

In Programm 3.29 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.

Programm 3.29

```
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 int value;

    private Ampelfarbe(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

public class Test {
    public static void info(Ampelfarbe farbe) {
        if (farbe == Ampelfarbe.ROT)
            System.out.println(farbe.getValue() + ": Anhalten");
        else if (farbe == Ampelfarbe.GELB)
```

```

        System.out.println(farbe.getValue() + ": Achtung");
    else if (farbe == Ampelfarbe.GRUEN)
        System.out.println(farbe.getValue() + ": Weiterfahren");
    }

    public static void main(String[] args) {
        info(Ampelfarbe.ROT);
        info(Ampelfarbe.GELB);
        info(Ampelfarbe.GRUEN);
    }
}

```

enum

Typsichere Aufzählungen können ab Java SE 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 `Enum` abgeleitet.

`enum`-Konstanten können als `case`-Konstanten in `switch`-Anweisungen genutzt werden.

Die Methode `ordinal` eines Aufzählungstyps liefert die Positionsnummer der Konstanten in der Aufzählung. Die Methode `toString` liefert den Namen der Konstanten, wie er in der Werteliste angegeben ist. Die Klassenmethode `values` liefert ein Array mit allen Aufzählungskonstanten.

Programm 3.30 nutzt den Aufzählungstyp `Ampelfarbe`.

Programm 3.30

```

public enum Ampelfarbe {
    ROT, GELB, GRUEN
}

public class Test {
    public static void info(Ampelfarbe farbe) {
        switch (farbe) {
            case ROT:
                System.out.println(farbe.ordinal() + ": Anhalten");
                break;
            case GELB:
                System.out.println(farbe.ordinal() + ": Achtung");
                break;
            case GRUEN:
                System.out.println(farbe.ordinal() + ": Weiterfahren");
                break;
        }
    }
}

```

```
public static void main(String[] args) {
    info(Ampelfarbe.ROT);
    info(Ampelfarbe.GELB);
    info(Ampelfarbe.GRUEN);

    for (Ampelfarbe farbe : Ampelfarbe.values()) {
        System.out.println(farbe.toString());
    }
}
```

Ein Aufzählungstyp kann wie eine Klasse weitere Attribute und Methoden haben. Bei der Deklaration der Konstanten werden in runden Klammern Argumente für den Konstruktorauftruf angegeben.

Programm 3.31 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".

Programm 3.31

```
public enum Note {
    SEHR_GUT(82, 90), GUT(70, 81), BEFRIEDIGEND(58, 69), AUSREICHEND(46, 57),
    MANGELHAFT(0, 45);

    private int von, bis;

    private 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;
    }
}

public class Test {
    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

```

3.12 Pakete

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

Der *Name eines Pakets* besteht im Allgemeinen aus mehreren mit Punkt getrennten Teilen, z. B. `projekt1.gui`.

package

Um eine Klasse bzw. ein Interface 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 projekt1.gui;
```

im Unterverzeichnis `gui` des Verzeichnisses `projekt1` liegen.

Obwohl nicht vorgegeben, sollte der zugehörige Quellcode ebenfalls in einem Unterverzeichnis mit Namen `gui` liegen. Das erhöht die Übersichtlichkeit.

Also z. B. Quellcode im Verzeichnis

```
src\projekt1\gui
```

und Bytecode im Verzeichnis

```
bin\projekt1\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.

Auf die gleiche Art und Weise können auch Interfaces importiert werden.

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

```
import Paketname.*;
```

Beispiel:

```
import projekt1.*;
```

macht alle Paketinhalte zugänglich, die direkt in `projekt1` liegen, aber nicht die Inhalte, die in `gui` liegen. Hierzu ist dann

```
import projekt1.gui.*;
```

nötig.

Eine Klasse wird vom Compiler erst dann gesucht, wenn sie im Programm benötigt wird. Wie wird der Bytecode vom Compiler bzw. Interpreter gefunden?

Beispiel:

```
import projekt1.gui.MeinFenster;
```

`MeinFenster.class` ist im Verzeichnis `projekt1\gui` gespeichert. Wo dieses Verzeichnis im Dateisystem liegt, kann in der Umgebungsvariablen `CLASSPATH` angegeben werden, bei Windows z. B. über den `set`-Befehl:

```
set CLASSPATH=.;D:\workspace
```

Compiler und Interpreter suchen nach dem Unterverzeichnis `projekt1` im aktuellen Verzeichnis und im Verzeichnis `D:\workspace`.

Default-Paket

Klassen oder Interfaces, 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`, `java.awt`, `java.applet` und `javax.swing`.

`java.lang` enthält grundlegende Klassen und Interfaces. Diese werden ohne explizite `import`-Anweisung in jeden Code automatisch importiert.

Sollte die entwickelte Software später ausgeliefert werden, ist es wichtig, dass die vollständigen Namen für Klassen und Interfaces 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:

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.

Der Umgang mit Paketen wird anhand des folgenden Beispiels erläutert.

Programm 3.32

```
package bank;

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

    public Konto() {
    }

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

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

    public Konto(Konto k) {
        kontonummer = k.kontonummer;
        saldo = k.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);
}

import bank.Konto;

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

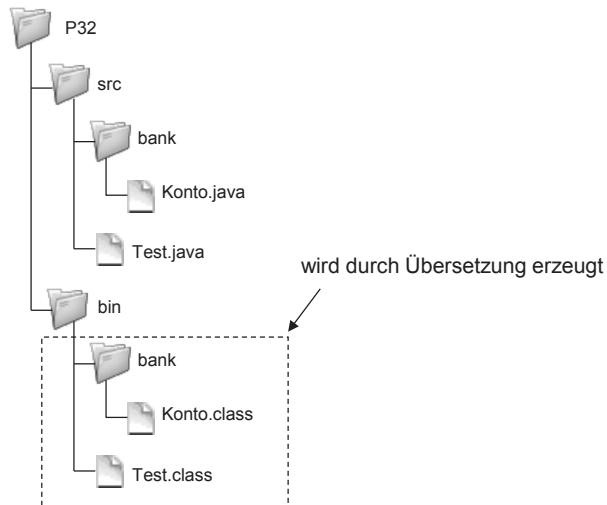


Abbildung 3-7: Ablagestruktur

Abbildung 3-7 zeigt, in welchen Verzeichnissen Quellcode und Bytecode abgelegt sind. Das Verzeichnis bin ist zunächst leer. Übersetzung und Ausführung erfolgen im Verzeichnis P32.

1. Übersetzung

```
javac -d bin src/bank/Konto.java

set CLASSPATH=bin
javac -d bin src/Test.java
```

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

Alternativ kann auch eingegeben werden (Das Setzen von CLASSPATH entfällt.):

```
javac -d bin -cp bin src/Test.java
```

2. Ausführung

```
set CLASSPATH=bin
java Test

oder

java -cp bin Test
```

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 (hier nur: bank\Konto.class):

```
jar cf bank.jar -C bin bank
```

Dann kann das Programm wie folgt aufgerufen werden:

```
java -cp bin;bank.jar Test
```

Das Verzeichnis bank wird für die Ausführung nicht benötigt.

Statische import-Klausel

Statische Methoden und statische Variablen werden über den Klassennamen angesprochen. Ab Java SE 5 können diese statischen Elemente einer Klasse so importiert werden, dass sie ohne vorangestellten Klassennamen verwendet werden können:

```
import static Paketname.Klassenname.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.

Programm 3.33

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

public class Test {
    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`.

3.13 Aufgaben

1. Wie kann eine Methode Daten von außen aufnehmen und nach außen weiterreichen?
2. Besitzt folgende Klasse einen *Standardkonstruktor*?

```
public class A {
    private int a;

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

3. Testen Sie den unter "Lokale Variable" in Kapitel 3.2 beschriebenen Sachverhalt.
4. 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.

5. 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 (siehe auch Aufgabe 11 in Kapitel 2). Die Buchwerte für die einzelnen Jahre sollen am Bildschirm ausgegeben werden.
Testen Sie die Methoden dieser Klasse.
6. 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.
7. Was versteht man unter der *Signatur* einer Methode und worin besteht der Unterschied zwischen dem *Überladen* und dem *Überschreiben* einer Methode?
8. Sind folgende Anweisungen korrekt? **K2** sei hier Subklasse von **K1**.

```
K1 p1 = new K1();  
K2 p2 = new K2();  
p1 = p2;  
p2 = (K2) p1;
```

9. 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.

10. Das Interface Anzeigbar soll die abstrakte Methode void zeige() enthalten. Implementieren Sie für die Klasse Sparbuch aus Aufgabe 4 dieses Interface. Die Methode zeige soll alle Attributwerte des jeweiligen Objekts ausgeben.

Definieren Sie dann die Klasse Utilities, die die Klassenmethode

```
public static void zeige(Anzeigbar a)
```

enthält. Diese Methode soll nach Ausgabe einer laufenden Nummer, die bei jedem Aufruf um 1 erhöht wird, die Anzeigbar-Methode zeige aufrufen.

11. Implementieren Sie die folgende Methode, die das Maximum beliebig vieler int-Zahlen liefert:

```
public static int max(int firstValue, int... remainingValues)
```

12. Eine "Liste ganzer Zahlen größer oder gleich 0" soll als Interface IntegerList mit den folgenden abstraktenMethoden 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 mit Hilfe eines Arrays in der Klasse `ArrayList` und testen Sie alle Methoden.

13. 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 Stacks.

Ist das Array voll, soll 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 soll in der Methode `push` erfolgen.

14. Definieren Sie den Aufzählungstyp `Wochentag` mit den Werten `MO`, `DI`, ..., `SO` und der Methode

```
public boolean wochenende(),
```

die angibt, ob ein Tag zum Wochenende gehört (vgl. Programm 3.31).

15. 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... auftraege)
```

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

16. 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`.

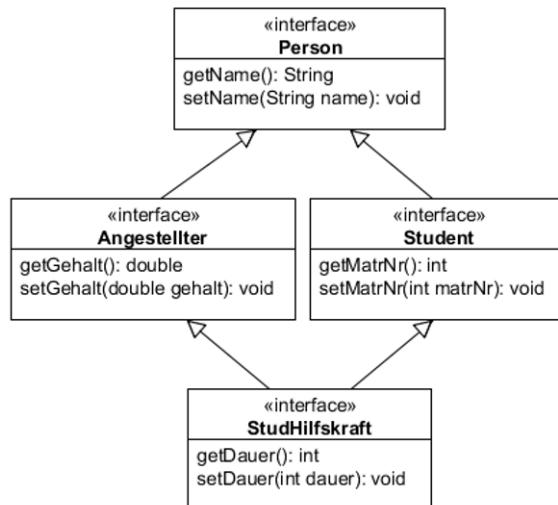
Schreiben Sie ein Testprogramm, das die Methoden für verschiedene Figuren testet. Nehmen Sie hierzu die Figuren in einem Array vom Typ `Figur` auf.

17. 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.

18. Lösen Sie Aufgabe 17 mit Hilfe eines Interfaces anstelle der abstrakten Klasse `Obst`.
19. Erstellen Sie die in der folgenden Abbildung dargestellten Interfaces sowie die Klasse `HiWi`, die das Interface `StudHilfskraft` implementiert.



20. Warum terminiert der Aufruf von `main` in der unten aufgeführten Klasse `B` mit einer `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();
    }
}

```

21. 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 ($n \geq 0$), dann gilt:

$$\begin{aligned} k(0) &= \text{kapital} \\ k(n) &= k(n-1) * (1 + \text{zinssatz}) \text{ für } n > 0 \end{aligned}$$

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 $(1 + \text{zinssatz})$ und dem Rückgabewert der aufgerufenen Methode.

22. Implementieren Sie einen Stack (Stapel) mit den Methoden `push` und `pop` (siehe Aufgabe 13). Die Elemente des Stapels sind Objekte vom Typ `Node`:

```
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.

23. 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 vom Typ `Node`:

```
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.

24. 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.

25. 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.

4 Ausnahmebehandlung

Während der Ausführung eines Programms können diverse Fehler bzw. *Ausnahmen* (*Exceptions*) auftreten, z. B. eine ganzzahlige Division durch 0, der Zugriff auf ein Arrayelement mit einem Index außerhalb der Grenzen des Arrays oder der Zugriff auf eine nicht vorhandene Datei.

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.

Ausnahme-Mechanismus

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 behandelt werden oder sie kann an die aufrufende Methode weitergereicht werden.
- Die Methode, an die die Ausnahme weitergereicht wurde, hat nun ihrerseits die Möglichkeit, sie entweder abzufangen und zu behandeln oder ebenfalls weiterzureichen.
- Wird die Ausnahme nur immer weitergereicht und in keiner Methode behandelt, bricht das Programm 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,
- wie Sie eigene Ausnahmeklassen definieren können.

4.1 Ausnahmetypen

Ausnahmen sind Objekte der Klasse `Throwable` oder ihrer Subklassen. Die in Abbildung 4-1 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 repräsentieren alle schwerwiegenden Fehler, die innerhalb des Laufzeitsystems auftreten und von einem Anwendungsprogramm nicht behandelt werden können.

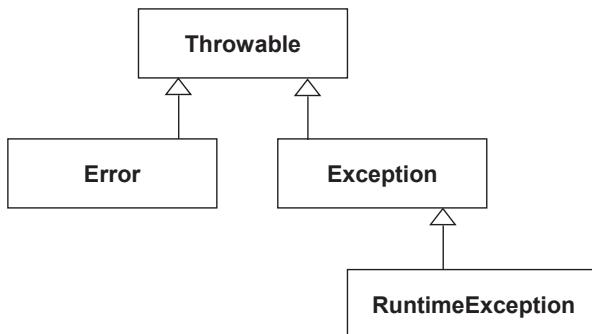


Abbildung 4-1: Typenhierarchie der Ausnahmen¹

Kontrollierte und nicht kontrollierte Ausnahmen

Die Klasse `Exception` und ihre Subklassen repräsentieren so genannte normale Fehler eines Programms. Dabei gibt es zwei Varianten: solche Ausnahmen, die vom Programm behandelt werden müssen (*kontrollierte Ausnahmen*) und solche, die vom Programm behandelt werden können, aber nicht müssen (*nicht kontrollierte Ausnahmen*).

Die Klasse `RuntimeException` und ihre Subklassen repräsentieren die *nicht kontrollierten Ausnahmen*.

Programm 4.1

```

public class Division {
    public static void main(String[] args) {
        for (int i = 5; i >= 0; i--) {
            System.out.println(10 / i);
        }
    }
}
  
```

¹ Diese Klassen implementieren alle das Interface `java.io.Serializable`. Dieses wird im Kapitel 8.9 behandelt.

Das Laufzeitsystem erzeugt eine *nicht kontrollierte Ausnahme*, die zum Programmabbruch führt:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
      at Division.main(Division.java:4)
```

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.²

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 in Kapitel 4.4 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.
```

Klassen, die direkt oder indirekt von `Exception`, aber nicht von `RuntimeException` angeleitet sind, repräsentieren *kontrollierte Ausnahmen*.

Die Ausnahmeklasse `KontoAusnahme` wird in den nachfolgenden Beispielen genutzt, wenn ein negativer Betrag ein- bzw. ausgezahlt oder zu viel Geld abgehoben werden soll.

² Siehe auch Kapitel 4.4.

Eine selbst definierte Ausnahmeklasse

```
public class KontoAusnahme extends Exception {
    public KontoAusnahme() {
    }

    public KontoAusnahme(String message) {
        super(message);
    }
}
```

4.2 Auslösung und Weitergabe von Ausnahmen

Kontrollierte Ausnahmen müssen entweder abgefangen und behandelt oder an den Aufrufer weitergegeben werden (*catch or throw*). Der Compiler achtet auf die Einhaltung dieser Regel.

throw

Mit Hilfe der Anweisung

```
throw Ausnahmeobjekt;
```

wird eine Ausnahme ausgelöst. `throw` unterricht das Programm an der aktuellen Stelle, um die Ausnahme zu behandeln oder die Weitergabe auszuführen.

throws

Wird die 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. 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 Programm 4.2 Ausnahmen vom Typ `KontoAusnahme`³ auslösen. Die Ausnahmen werden dann an den Aufrufer der jeweiligen Methode zur Behandlung weitergereicht.

Programm 4.2

```
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)
```

³ Siehe Beispiel im Kapitel 4.1.

```

        throw new KontoAusnahme("Betrag > Saldo");
        saldo -= betrag;
    }

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

public class Test {
    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:

Exception in thread "main" KontoAusnahme: Betrag > Saldo

Das Fehlen der `throws`-Klausel in diesem Beispiel würde zu einer Fehlermeldung des Compilers führen.

4.3 Abfangen von Ausnahmen

`try ... catch`

Das Abfangen 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 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

Ab Java SE 7 können mehrere Ausnahmetypen 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 (siehe Programm 4.3).

Programm 4.3

```
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);
        } catch (ArithmetricException e) {
            System.out.println("Autsch! " + e.getMessage());
        }
    }
}
```

Im Programm 4.3 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.⁴ `ArithmetricException` (bei Division durch 0) wird anders behandelt.

In der `catch`-Klausel könnte hier natürlich auch eine gemeinsame Superklasse wie `RuntimeException` oder `Exception` stehen.

Gegenüber Programm 4.2 werden im folgenden Programm mögliche Ausnahmen abgefangen.

Programm 4.4

```
public class Test {
    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();
    }
}
```

Ausgabe des Programms:

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,

⁴ Siehe Kapitel 5.2: Integer-Methode `parseInt`.

- 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.

Programm 4.5 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 Fällen wird der finally-Block durchlaufen.

Programm 4.5

```
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);  
                            break;  
                        case 2:  
                            kto.zahleAus(700);  
                            break;  
                        case 3:  
                            kto.zahleAus(200 / 0);  
                            break;  
                    }  
                } catch (KontoAusnahme e) {  
                    System.out.println(e);  
                } finally {  
                    System.out.println("Auszug 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.ArithemeticException: / by zero
at FinallyTest.main(FinallyTest.java:18)
```

4.4 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 wird die ursprüngliche Ausnahme als Ursache (*cause*) in der neuen Ausnahme gespeichert. 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 (Programm 4.6) werden die `Throwable`-Methoden `initCause` und `getCause` zum Speichern bzw. Abfragen der ursprünglichen Ausnahme genutzt.

Programm 4.6

```
public class MyExceptionV1 extends Exception {
}
```

```
public class ChainingTestV1 {  
    public static String getNachbarn(int[] x, int i) throws MyExceptionV1 {  
        try {  
            return x[i - 1] + " " + x[i] + " " + x[i + 1];  
        } catch (RuntimeException e) {  
            throw (MyExceptionV1) new MyExceptionV1().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 (MyExceptionV1 e) {  
            System.out.println("Ursache: " + e.getCause());  
        }  
  
        try {  
            System.out.println(getNachbarn(b, 4));  
        } catch (MyExceptionV1 e) {  
            System.out.println("Ursache: " + e.getCause());  
        }  
    }  
}
```

Im zweiten Fall (Programm 4.7) enthält die Ausnahmeklasse `MyExceptionV2` einen Konstruktor, an den die Referenz auf die ursprüngliche Ausnahme direkt als Argument übergeben wird.

Programm 4.7

```
public class MyExceptionV2 extends Exception {  
    public MyExceptionV2() {}  
  
    public MyExceptionV2(Throwable t) {  
        super(t);  
    }  
}  
  
public class ChainingTestV2 {  
    public static String getNachbarn(int[] x, int i) throws MyExceptionV2 {  
        try {  
            return x[i - 1] + " " + x[i] + " " + x[i + 1];  
        } catch (RuntimeException e) {  
            throw new MyExceptionV2(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 (MyExceptionV2 e) {
            System.out.println("Ursache: " + e.getCause());
        }

        try {
            System.out.println(getNachbarn(b, 4));
        } catch (MyExceptionV2 e) {
            System.out.println("Ursache: " + e.getCause());
        }
    }
}

```

Ausgabe beider Programme:

```

Ursache: java.lang.NullPointerException
Ursache: java.lang.ArrayIndexOutOfBoundsException: 5

```

4.5 Aufgaben

- Erläutern Sie den Mechanismus der Ausnahmebehandlung. Welche Java-Schlüsselwörter sind mit den einzelnen Schritten verbunden?
- Was besagt die *catch-or-throw*-Regel?
- 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 Parameter 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.

- Die ganzzahlige 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.

5. Erstellen Sie eine Klasse `Person`, die nur einen Wert zwischen 0 und 150 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. Die 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.

5 Ausgewählte Standardklassen

In diesem Kapitel werden einige wichtige Klassen und Interfaces aus den Java-Standardpaketen vorgestellt. Sie können in verschiedenen Anwendungssituationen eingesetzt werden.

Lernziele

In diesem Kapitel lernen Sie

- das Leistungsangebot wichtiger Klassen kennen, um z. B. Zeichenketten zu verarbeiten, Objekte in so genannten Containern zu verwalten, Arrays zu sortieren und zu durchsuchen, Objekte von Klassen zu erzeugen, deren Existenz zum Zeitpunkt der Entwicklung des nutzenden Programms noch nicht bekannt sein müssen und vieles mehr,
- wie Datum und Zahlen sprach- und länderspezifisch dargestellt und sprachabhängige Texte verwaltet werden können.

5.1 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`, `StringBuffer` und `StringTokenizer`.

5.1.1 Die Klasse String

Die Klasse `java.lang.String` repräsentiert Zeichenketten, bestehend aus Unicode-Zeichen. Objekte vom Typ `String` (im Folgenden 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 (z. B. "Hallo") werden als Objekte dieser Klasse implementiert. So zeigt z. B. die Referenzvariable `s` nach der Initialisierung durch `String s = "Hallo"` auf das Objekt, das die Zeichenkette "Hallo" repräsentiert.

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)
```

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 der Plattform verwendet. In den beiden letzten Fällen werden length Zeichen bzw. Bytes aus einem Array beginnend mit dem Index offset in eine Zeichenkette umgewandelt.

Im Folgenden werden wichtige Methoden der Klasse `String` vorgestellt.

`int length()`

liefert die Anzahl Zeichen in der Zeichenkette.

`String concat(String s)`

liefert einen neuen String, der aus der Aneinanderreihung des Strings, für den die Methode aufgerufen wurde, und 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.

Vergleichen

`boolean equals(Object obj)`

liefert true, wenn obj ein String ist und beide Strings die gleiche Zeichenkette repräsentieren.

`boolean equalsIgnoreCase(String s)`

wirkt wie `equals`, ignoriert aber eventuell vorhandene Unterschiede in der Groß- und Kleinschreibung.

`boolean startsWith(String s, int start)`

liefert true, wenn dieser String mit der Zeichenkette in s an der Position start beginnt.

`boolean startsWith(String s)`

ist gleichbedeutende mit `startsWith(s, 0)`.

`boolean endsWith(String s)`

liefert true, wenn dieser String mit der Zeichenkette in 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 in s lexikographisch und liefert 0, wenn beide Zeichenketten gleich sind, einen negativen Wert, wenn

diese Zeichenkette kleiner ist als die in s, und einen positiven Wert, wenn diese Zeichenkette größer ist als die in s.

Das Laufzeitsystem verwaltet alle String-Literale eines Programms in einem Pool und liefert für alle Objekte, die das gleiche String-Literal repräsentieren, die gleiche Referenz zurück. Programm 5.1 demonstriert den Vergleich von Strings mit `==` und `equals`.

Programm 5.1

```
public class Vergleiche {  
    public static void main(String[] args) {  
        String a = "Hallo";  
        String b = new String(a);  
        String c = "Hallo";  
  
        System.out.println(a.equals(b)); // true  
        System.out.println(a == b); // false  
        System.out.println(a == c); // 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)
```

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.

Ersetzen

`String replace(char c, char t)`

erzeugt einen neuen String, in dem jedes Auftreten des Zeichens `c` in diesem String durch das Zeichen `t` ersetzt ist.

`StringtoLowerCase()`

wandelt die Großbuchstaben der Zeichenkette in Kleinbuchstaben um und liefert diesen String dann zurück.

`StringtoUpperCase()`

wandelt die Kleinbuchstaben der Zeichenkette in Großbuchstaben um und liefert diesen String dann zurück.

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()` geliefert.

Object-Methode `toString`

Die Klasse `Object` 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.

```
byte[] getBytes()
    liefert ein Array von Bytes, das die Zeichen des Strings in der Standardcodierung enthält.

char[] toCharArray()
    liefert ein char-Array, das die Zeichen des Strings enthält.
```

Das folgende Programm demonstriert einige der hier aufgeführten Methoden.

Programm 5.2

```
public class StringTest {
    public static void main(String[] args) {
        String nachname = "Schmitz", vorname = "Hugo";

        // Verketten
        String name = nachname + ", " + vorname;
        name = name.concat(" Egon");
        // alternativ: name += " Egon";
        System.out.println("Name: " + name);

        // Länge
        System.out.println("Länge des Namens: " + name.length());

        // Extrahieren
        System.out.println("Teilstring: " + name.substring(0, 13));

        // Vergleichen
        if (name.endsWith("Egon"))
            System.out.println("Name endet mit: \"Egon\"");
        if (nachname.equals("Schmitz"))
            System.out.println("Nachname ist \"Schmitz\"");
        if (vorname.compareTo("Egon") > 0)
            System.out.println("\"" + vorname + "\" ist größer als \"Egon\"");

        // Suchen
        if (name.indexOf("Hugo") >= 0)
            System.out.println("Name enthält \"Hugo\"");

        // Ersetzen
        System.out.println("Name in Kleinbuchstaben: " + name.toLowerCase()
            + "\"");

        // Konvertieren
        String s = String.valueOf(3.57);
        System.out.println("Zahl mit Dezimalkomma: " + s.replace('.', ','));
    }
}
```

Ausgabe des Programms:

```
Name: Schmitz, Hugo Egon
Länge des Namens: 18
Teilstring: Schmitz, Hugo
Name endet mit: "Egon"
Nachname ist "Schmitz"
```

"Hugo" ist größer als "Egon"
 Name enthält "Hugo"
 Name in Kleinbuchstaben: "schmitz, hugo egon"
 Zahl mit Dezimalkomma: 3,57

5.1.2 Die Klassen StringBuffer und StringBuilder

Da String-Objekte unveränderbar sind, entstehen bei String-Manipulationen fortlaufend neue String-Objekte als Zwischenergebnisse, was relativ zeitaufwändig ist. Hier helfen die Klassen `StringBuffer` und `StringBuilder`.

Die Klasse StringBuffer

Objekte der Klasse `java.lang.StringBuffer` enthalten *veränderbare* Zeichenketten. Bei Änderungen wird der benötigte Speicherplatz automatisch in der Größe angepasst.

Konstruktoren

`StringBuffer()`

erzeugt einen leeren Puffer mit einer Anfangsgröße von 16 Zeichen.

`StringBuffer(int capacity)`

erzeugt einen leeren Puffer mit einer Anfangsgröße von `capacity` Zeichen.

`StringBuffer(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 `StringBuffer`-Objekt in einen String um.

Extrahieren

`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.
```

Änderungen

```
StringBuffer append(Typ x)
```

hängt die Stringdarstellung von x an das Ende der Zeichenkette im Puffer. *Typ* steht hier für boolean, char, int, long, float, double, char[], String oder Object.

```
StringBuffer append(StringBuffer sb)
```

hängt die Zeichenkette in sb an das Ende der Zeichenkette im Puffer.

```
StringBuffer insert(int i, Typ x)
```

fügt die Stringdarstellung von x an der Position i der Zeichenkette im Puffer ein. *Typ* wie bei append.

```
StringBuffer delete(int start, int end)
```

löscht den Teilstring ab der Position start bis zur Position end - 1 im Puffer.

```
StringBuffer 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.

Zeichen lesen und ändern

```
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.

Die Klasse StringBuilder

Seit Java SE 5 gibt es die Klasse java.lang.StringBuilder, die die gleichen Methoden wie die Klasse StringBuffer anbietet.

Da die Methoden von StringBuffer dort, wo es nötig ist, synchronisiert sind, können mehrere Threads¹ 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.

¹ Siehe Kapitel 9.

Programm 5.3

```
public class StringBuilderTest {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Ich");

        sb.append("was soll es bedeuten ...");
        System.out.println(sb);

        sb.insert(3, " weiß nicht, ");
        System.out.println(sb);

        int len = sb.length();
        for (int i = 0; i < len; i++) {
            if (sb.charAt(i) == 'e')
                sb.setCharAt(i, 'u');
            if (sb.charAt(i) == 'E')
                sb.setCharAt(i, 'U');
        }
        System.out.println(sb);
    }
}
```

Ausgabe des Programms:

```
Ichwas soll es bedeuten ...
Ich weiß nicht, was soll es bedeuten ...
Ich wuiß nicht, was soll us buduutun ...
```

5.1.3 Die Klasse StringTokenizer

Die Klasse `java.util.StringTokenizer` ist ein nützliches 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"² 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.

Programm 5.4

```
import java.util.StringTokenizer;

public class TextZerlegen {
    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());
        }
    }
}
```

Ausgabe des Programms:

```
Ich
weiß
nicht
was
soll
es
bedeuten
```

² Siehe Tabelle 2-2.

5.2 Hüllklassen und Autoboxing

Zu jedem einfachen Datentyp gibt es eine so genannte *Hüllklasse* (*Wrapper-Klasse*), deren Objekte Werte dieses Datentyps speichern. Somit können einfache Werte indirekt als Objekte angesprochen werden.

Hüllklassen bieten eine Reihe von nützlichen Methoden. So können z. B. Zeichenketten in Werte des entsprechenden einfachen Datentyps umgewandelt werden.

Tabelle 5-1: Hüllklassen

Einfacher Datentyp	Hüllklasse
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Hüllobjekte sind nach ihrer Erzeugung nicht mehr veränderbar.

Konstruktoren

Zur Erzeugung von "Hüllobjekten" kann dem Konstruktor ein Literal bzw. eine Variable vom einfachen Datentyp oder ein String mit entsprechendem Inhalt übergeben werden. Bei der Übergabe von inkompatiblen Strings wird im Fall der numerischen Hüllklassen die nicht kontrollierte Ausnahme `java.lang.NumberFormatException` ausgelöst.

```
Boolean(boolean val)
Boolean(String s)
Character(char val)
Byte(byte val)
Byte(String s)
Short(short val)
Short(String s)
Integer(int val)
Integer(String s)
Long(long val)
Long(String s)
Float(float val)
Float(double val)
Float(String s)
Double(double val)
Double(String s)
```

Erzeugung von Hüllobjekten

Zu jeder Hüllklasse 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 Hüllklasse 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 Hüllklasse Boolean, Byte, Short, Integer, Long, Float bzw. Double.

Für die Hüllklassen 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 Hüllobjekt, das die ganze Zahl 5 umhüllt.

Für Character gibt es die Methode

```
static Character valueOf(char c)
```

Weitere Methoden

Für alle Hüllklassen gibt es die Methoden equals und `toString`:

```
boolean equals(Object obj)
```

vergleicht das Hüllobjekt mit obj und gibt true zurück, falls obj vom Typ der Hüllklasse ist und denselben Wert umhüllt.

```
String toString()
```

liefert den Wert des Hüllobjekts als Zeichenkette.

Für die Hüllklassen Integer und Long existieren die Methoden:

```
static String toBinaryString(typ i)
```

```
static String toOctalString(typ i)
```

```
static String toHexString(typ i)
```

Sie erzeugen eine Stringdarstellung 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) oderWhiteSpace (Leerzeichen, Tabulator, Seitenvorschub, Zeilenende) eingesetzt werden.

Rückgabe umhüllter Werte

Die folgenden Methoden liefern den durch das Objekt der jeweiligen Hüllklasse 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 Hüllobjekts ermittelt und der Variablen `x` zugewiesen.

Umwandlung von Strings in Zahlen

Die Umwandlung von Strings in Zahlen kann 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 Hüllklassen `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 Hüllklasse 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 hinausführen, brechen nicht ab. Sie werden auf dem darstellbaren Bereich ausgeführt und erzeugen dann durch Überlauf falsche Ergebnisse.

Programm 5.5

```
public class HuellTest {
    public static void main(String[] args) {
        // Zeichenkette in Zahl umwandeln
        String s = "10500";
        Integer intObj = Integer.valueOf(s);
        int x = intObj.intValue();
        System.out.println("'" + s + "' -> " + x);

        // oder kuerzer
        x = Integer.valueOf(s).intValue();
        System.out.println("'" + s + "' -> " + x);

        // oder
        x = Integer.parseInt(s);
        System.out.println("'" + s + "' -> " + x);

        // Zahl in Zeichenkette umwandeln
        double y = 123.76;
        Double doubleObj = new Double(y);
        s = doubleObj.toString();
        System.out.println(y + " -> '" + s + "'");

        // oder kuerzer
        s = String.valueOf(y);
        System.out.println(y + " -> '" + s + "'");

        // 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));

        // 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:

```

"10500" -> 10500
"10500" -> 10500
"10500" -> 10500
123.76 -> "123.76"
123.76 -> "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 Hüllobjekten ist umständlich.

Beispiel:

Umwandlung eines int-Werts in ein Integer-Objekt:

```

int i = 4711;
Integer iobj = new Integer(i);

```

Umwandlung eines Integer-Objekts in einen int-Wert:

```
i = iobj.intValue();
```

Ab Java SE 5 steht das so genannte *Autoboxing* zur Verfügung. Beim Autoboxing wandelt der Compiler bei Bedarf einfache Datentypen in Objekte der entsprechenden Hüllklasse um. *Auto-Unboxing* bezeichnet den umgekehrten Vorgang: die automatische Umwandlung eines Hüllobjekts 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. Dies zeigt Programm 5.6.

Programm 5.6

```
public class BoxingTest {
    public static void main(String[] args) {
        Integer integer = 1234; // boxing
        int i = integer; // unboxing

        IntegerBox box = new IntegerBox();

        // Lösung vor Java SE 5
        box.setValue(new Integer(4711));
        i = box.getValue().intValue();
        System.out.println(i);

        // Lösung ab Java SE 5
        box.setValue(4711);
        i = box.getValue();
        System.out.println(i);

        // 1. unboxing
        // 2. Ausführung der Rechenoperation
        // 3. boxing
        integer++;
        System.out.println(integer);
        integer += 10;
        System.out.println(integer);
    }
}

public class IntegerBox {
    private Integer value;

    public void setValue(Integer value) {
        this.value = value;
    }

    public Integer getValue() {
        return value;
    }
}
```

5.3 Die Klasse Object

Die Klasse `java.lang.Object` ist die Wurzel der Klassenhierarchie. Jede Klasse, die nicht explizit von einer anderen Klasse abgeleitet ist (`extends`), 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*³ weitere allgemeine Methoden, von denen wir hier einige vorstellen.

5.3.1 Die Methoden equals und hashCode

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` 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`.

Für Objekte `x` und `y` gilt: `x.equals(y)` hat den Wert `true` genau dann, wenn `y.equals(x)` den Wert `true` hat. 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`. Für ein Objekt `x` gilt: `x.equals(null)` hat den Wert `false`.

Die zu vergleichenden Objekte müssen vom gleichen Typ sein. Diese 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 Kapitel 5.6).
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 Programm 5.7.

³ Siehe Kapitel 9.

hashCode

```
int hashCode()
```

liefert einen ganzzahligen Wert, den so genannten *Hashcode*, der beispielsweise für die Speicherung von Objekten in *Hashtabellen* gebraucht wird.⁴

Wurde in einer Klasse `equals` überschrieben, so sollte `hashCode` so überschrieben werden, dass zwei "gleiche" Objekte auch den gleichen Hashcode haben. Zwei gemäß `equals` verschiedene Objekte dürfen den gleichen Hashcode haben.

In der Klasse `Konto` aus Programm 5.7 sind die beiden geerbten Methoden `equals` und `hashCode` neu implementiert. Zwei `Konto`-Objekte sind genau dann gleich, wenn ihre Kontonummern (`ktoId`) gleich sind. Der Hashcode ist der `int`-Wert der Kontonummer.

Programm 5.7

```
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;  
    }  
}
```

⁴ Siehe Kapitel 5.4.2.

```
public Kunde getKunde() {
    return kunde;
}

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

public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (!(obj instanceof Konto))
        return false;
    Konto other = (Konto) obj;
    if (id != other.id)
        return false;
    return true;
}

public int hashCode() {
    return id;
}
}

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;
    }
}
```

```

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

```

5.3.2 Flache und tiefe Kopien

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 so genanntes Markierungs-Interface), implementieren. Wird `clone` für ein Objekt aufgerufen, dessen Klasse `Cloneable` nicht implementiert, so 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.

In Programm 5.8 ist die Klasse `Konto` aus Programm 5.7 um die Methode `clone` ergänzt worden. `Konto` implementiert das Interface `Cloneable`.

Programm 5.8

```

public class Konto implements Cloneable {

    ...

    public Konto clone() {

```

```

        try {
            return (Konto) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }

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 = 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. 42, 40880 Ratingen

```

Der Test zeigt, dass zwar ein Klon des Objekts `konto1` erzeugt wird, dass aber die Instanzvariablen `kunde` des Originals und des Klons beide dasselbe `Kunde`-Objekt referenzieren. Es wurde eine *flache Kopie* erzeugt.

So genannte *tiefe Kopien* müssen auch die referenzierten Objekte berücksichtigen.

Die `clone`-Methode der Klasse `Konto` in Programm 5.9 erzeugt mit Hilfe des Konstruktors `Kunde(Kunde other)` der Klasse `Kunde` ein neues `Kunde`-Objekt und weist es der Instanzvariablen `kunde` des Klons zu.

Programm 5.9

```
public class Konto implements Cloneable {  
    ...  
    public Konto clone() {  
        try {  
            Konto k = (Konto) super.clone();  
            k.kunde = new Kunde(kunde);  
            return k;  
        } catch (CloneNotSupportedException e) {  
            return null;  
        }  
    }  
}  
  
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 = 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

In Programm 5.9 wird der so genannte 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:

```
public Konto clone() {
    try {
        Konto k = (Konto) super.clone();
        k.kunde = kunde.clone();
        return k;
    } catch (CloneNotSupportedException e) {
        return null;
    }
}
```

5.4 Container

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.⁵

Das Paket `java.util` enthält daneben eine Reihe von Interfaces und Klassen, die ein leistungsfähiges Framework für Container bereitstellen: das *Collection Framework*.

Ab Java SE 5 können diese Container mit ihrem Elementtyp parametrisiert werden, sodass nur Objekte des vorgegebenen Typs abgelegt werden können.⁶ In dem hier vorliegenden Kapitel gehen wir auf diese Möglichkeit nicht ein. Diesbezügliche Warnungen des Compilers können ignoriert werden.

⁵ Siehe auch Kapitel 8.2.

⁶ Siehe Kapitel 6.

5.4.1 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 *Collection 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.

Programm 5.10 nutzt die Klasse Kunde aus Kapitel 5.3.

Programm 5.10

```
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 in Kapitel 3.10 kann auch hier ab Java SE 5 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` wird verwendet, 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.

Programm 5.11

```
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());
        }
    }
}
```

5.4.2 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.

Beispiel:

Schlüssel	Wert
Farbe	rot
Hoehe	100.0
Durchmesser	150.0

Die Klassen der Objekte, die in die Tabelle als Schlüssel eingetragen werden sollen, müssen die Methoden `equals` und `hashCode` in geeigneter Form implementieren.⁷

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 verwendet, um den Speicherplatz in der Tabelle herauszusuchen.

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.

⁷ Siehe Kapitel 5.3.

```
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.
```

Das folgende Programm demonstriert den Umgang mit Hashtabellen.

Programm 5.12

```
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
```

5.4.3 Property-Listen

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 Dateiinhalt (siehe folgendes Beispiel).⁸

Datei `properties.txt`:

```
#Beispiel Version 1
Durchmesser=150.0
Gewicht=5.0
Farbe=rot
Hoehe=100.0
```

Zeilen werden durch `#` auf Kommentar gesetzt.

⁸ `InputStream` und `OutputStream` werden im Kapitel 8 behandelt.

Programm 5.13 lädt und speichert eine Property-Liste im Dateisystem.⁹

Programm 5.13

```

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();

        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();
    }
}

```

Ab Java SE 5 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

```

Programm 5.14 zeigt die XML-Version von Programm 5.13.

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>

```

⁹ Die Konstrukte, die die Dateiverarbeitung betreffen, werden im Kapitel 8 ausführlich behandelt.

```

<entry key="Gewicht">5.0</entry>
<entry key="Farbe">rot</entry>
<entry key="Hoehe">100.0</entry>
</properties>

```

Programm 5.14

```

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();
    }
}

```

5.5 Die Klasse System

Die Klasse `java.lang.System` enthält wichtige Methoden zur Kommunikation mit dem zu Grunde liegenden 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

```

Sie sind die *Standarddatenströme* zur Eingabe, Ausgabe und Fehlerausgabe.

Die Klassen `java.io.InputStream` und `java.io.PrintStream` werden im Kapitel 8 behandelt. Mit `System.out.println(...)` wird die Methode `println` der Klasse `PrintStream` für das Objekt `out` aufgerufen.

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.

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 übergeben. Üblicherweise signalisiert 0 ein fehlerfreies Programmende.

Umgebungsvariablen

`static String getenv(String name)`

liefert den Wert der Umgebungsvariablen `name` des Betriebssystems.

System Properties

`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 ...`

Programm 5.15 listet alle Java-Systemeigenschaften auf.

Tabelle 5-2: Einige Java-Systemeigenschaften

Property	Bedeutung
file.separator	Dateipfadtrennzeichen
java.class.path	aktueller Klassenpfad
java.class.version	Version der Klassenbibliothek
java.home	Installationsverzeichnis
java.vendor	Herstellername
java.vendor.url	URL des Herstellers
java.version	Java-Versionsnummer
line.separator	Zeilentrennzeichen
os.arch	Betriebssystemarchitektur
os.name	Betriebssystemname
os.version	Betriebssystemversion
path.separator	Trennzeichen in PATH-Angaben
user.dir	aktuelles Arbeitsverzeichnis
user.home	Home-Verzeichnis
user.name	Anmeldename

Programm 5.15

```

import java.util.Enumeration;
import java.util.Properties;

public class SystemTest {
    public static void main(String[] args) {
        Properties p = System.getProperties();

        Enumeration e = p.propertyNames();
        while (e.hasMoreElements()) {
            String key = (String) e.nextElement();
            System.out.println(key + " = " + System.getProperty(key));
        }
    }
}

```

5.6 Die Klasse Class

Jeder Typ (Klasse, Interface, Aufzählung, Array, einfacher Datentyp, `void`) einer laufenden Java-Anwendung wird durch ein Objekt der Klasse `java.lang.Class` beschrieben. `Class` hat keine `public`-Konstruktoren. Objekte vom Typ `Class` entstehen automatisch, wenn Klassen geladen werden. Zu jedem Typ existiert genau ein `Class`-Objekt.

Typinformationen zur Laufzeit

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.¹⁰

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 das Schlüsselwort `void` gibt es Konstanten des Typs `Class`. Diese werden durch den Typnamen mit Suffix `.class` gebildet, z. B. `int.class`, `int[].class`, `void.class`. Ebenso erhält man für eine Klasse, ein Interface oder eine 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.

Programm 5.16

```
public class ClassTest1 {
    public static void main(String[] args) throws ClassNotFoundException {
        String className = "java.util.Vector";
        System.out.println(className.getClass().getName());
        System.out.println(String.class.getName());
        System.out.println(Class.forName(className).getName());
    }
}
```

¹⁰ Siehe Programm 5.18.

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 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.

`Object newInstance() throws java.lang.InstantiationException,
java.lang.IllegalAccessException`

erzeugt ein neues Objekt der Klasse, die durch das `Class`-Objekt repräsentiert wird, für das die Methode aufgerufen wurde. Diese Methode ruft den parameterlosen Konstruktor der Klasse auf und gibt eine Referenz auf das erzeugte Objekt zurück.

`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 dem auf `'/'` folgenden Zeichen gebildet. Andernfalls wird der absolute Name aus dem Paketnamen der Klasse gebildet, wobei ein `'. '` durch ein Pfadtrennzeichen ersetzt wird.

Beispiel:

Klasse: de.gkjava.addr.ConnectionManager

name	absoluter Name
/dbparam.txt	dbparam.txt
dbparam.txt	de/gkjava/addr/dbparam.txt

Ressourcen können auch in jar-Dateien zusammengefasst werden. Diese jar-Dateien müssen dann in den CLASSPATH eingebunden werden.¹¹

`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*.^{12 13}

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.

Das folgende Beispielprogramm liefert Informationen über Klassen und Interfaces, deren Namen als Parameter beim Aufruf mitgegeben werden.

Programm 5.17

```
public class ClassTest2 {
    public static void main(String[] args) throws ClassNotFoundException {
        Class c = Class.forName(args[0]);
        boolean isInterface = c.isInterface();
        System.out.print(isInterface ? "Interface: " : "Klasse: ");
        System.out.println(c.getName());

        Class s = c.getSuperclass();
        if (s != null)
            System.out.println("Superklasse: " + s.getName());

        Class[] interfaces = c.getInterfaces();
        if (interfaces.length > 0) {
            if (isInterface)
                System.out.println("Superinterfaces:");
            else
                System.out.println("implementierte Interfaces:");
        }
    }
}
```

¹¹ Siehe Kapitel 10.6.

¹² Siehe Kapitel 13.1.

¹³ Die Methode `getResource` wurde bereits im Programm 1.2 aus Kapitel 1 verwendet.

```
        for (int i = 0; i < interfaces.length; i++)
            System.out.println("\t" + interfaces[i].getName());
    }
}
```

Beispiel:

```
java -cp bin ClassTest2 java.util.Vector  
Klasse: java.util.Vector  
Superklasse: java.util.AbstractList  
implementierte Interfaces:  
    java.util.List  
    java.util.RandomAccess  
    java.lang.Cloneable  
    java.io.Serializable
```

Programm 5.18 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 RechenTest nutzt Class-Methoden, um ein Objekt zu erzeugen und hierfür die Schnittstellenmethode auszuführen.

Programm 5.18

```
public interface Berechnung {  
    int berechne(int a, int b);  
}  
  
public class Addition implements Berechnung {  
    public int berechne(int a, int b) {  
        return a + b;  
    }  
}  
  
public class Subtraktion implements Berechnung {  
    public int berechne(int a, int b) {  
        return a - b;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) throws Exception {  
        Class c = Class.forName(args[0]);  
        Berechnung b = (Berechnung) c.newInstance();  
        System.out.println(b.berechne(10, 5));  
    }  
}
```

Aufrufbeispiele:

```
java -cp bin RechenTest Addition  
15  
  
java -cp bin RechenTest Subtraktion  
5
```

5.7 Die Klasse Arrays

Die Klasse `java.util.Arrays` bietet Methoden zum komfortablen Arbeiten mit Arrays.

`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`.

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` (siehe Kapitel 5.3) 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.

`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 Rückgabewert ist ≥ 0 genau dann, wenn `key` gefunden wurde.

`static String toString(Typ[] a)`

liefert die Zeichenkettendarstellung des Arrays `a`. Als `Typ` kann hier ein einfacher Datentyp oder `Object` eingesetzt werden.

Programm 5.19 sortiert zum einen Zahlen, zum anderen Konten nach ihrer Kontonummer. Die Klasse `Konto` implementiert dazu das Interface `Comparable`.

Programm 5.19

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

    public Konto() {}

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

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

    public Konto(Konto k) {
        kontonummer = k.kontonummer;
        saldo = k.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 String toString() {
    return kontonummer + " " + saldo;
}

public int compareTo(Object obj) {
    return kontonummer - ((Konto) obj).kontonummer;
}
}

import java.util.Arrays;

public class ArraysTest {
    public static void main(String[] args) {
        int[] z = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
        Arrays.sort(z);
        System.out.println(Arrays.toString(z));

        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, 3, 4, 5, 6, 7, 8, 9]
[1234 400.0, 4044 300.0, 4711 100.0, 6000 200.0]
```

5.8 Mathematische Funktionen

Die Klasse `java.lang.Math` enthält die beiden `double`-Konstanten `E` (Eulersche Zahl e) und `PI` (Kreiskonstante π) sowie grundlegende *mathematische Funktionen*, die als Klassenmethoden realisiert sind.

Tabelle 5-3: Einige Methoden der Klasse `Math`

Klassenmethode	Erläuterung
<code>Typ abs(Typ x)</code>	Absolutbetrag von <code>x</code> . <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: <code>-1</code> für $x < 0$, <code>0</code> für $x = 0$, <code>1</code> für $x > 0$

Klassenmethode	Erläuterung
<i>Typ</i> <code>min(<i>Typ</i> x, <i>Typ</i> y)</code>	Minimum von x und y. <i>Typ</i> steht für <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> .
<i>Typ</i> <code>max(<i>Typ</i> x, <i>Typ</i> y)</code>	Maximum von x und y. <i>Typ</i> 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
<code>double cos(double x)</code>	Cosinus von x
<code>double tan(double x)</code>	Tangens von x
<code>double sinh(double x)</code>	Sinus hyperbolicus von x
<code>double cosh(double x)</code>	Cosinus hyperbolicus von x
<code>double tanh(double x)</code>	Tangens hyperbolicus von x
<code>double asin(double x)</code>	Arcussinus von x
<code>double acos(double x)</code>	Arcuscosinus von x
<code>double atan(double x)</code>	Arcustangens von x
<code>double random()</code>	Zufallszahl ≥ 0 und < 1

Programm 5.20

```
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("Flaeche: " + 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
Flaeche: 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.

Der String `pattern` kann sich aus folgenden Zeichen zusammensetzen:

- # Ziffer, führende Nullen werden nicht angezeigt
- 0 Ziffer, führende Nullen werden als 0 angezeigt
- .
- ,
- %
- Dezimalpunkt
- Symbol für Tausendergruppierung
- 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");
String s = f.format(24522.4567);
```

`s` enthält die Zeichenkette "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 ≥ 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 Standardabweichung 1.
```

Programm 5.21

```
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  
22 74 96 17 17 65 53 42 53 92
```

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 Stringdarstellung `s`.

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 $1 - 0.5^{100}$ eine Primzahl ist, sonst `false`.

Programm 5.22 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 bei der Verschlüsselung (Public-Key-Verfahren) eine wichtige Rolle.

Programm 5.22

```

import java.math.BigInteger;
import java.util.Random;

public class Primzahlen {
    private static final BigInteger NULL = new BigInteger("0");
    private static final BigInteger EINS = new BigInteger("1");
    private static final BigInteger ZWEI = new BigInteger("2");

    // erzeugt eine zufällige n-stellige Zahl
    public static BigInteger getZahl(int n) {
        Random r = new Random();
        StringBuilder s = new StringBuilder("");

        s.append(String.valueOf(1 + r.nextInt(9)));
        for (int i = 1; i < n; i++)
            s.append(String.valueOf(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(ZWEI).equals(NULL)) // gerade Zahl?
            start = start.add(EINS);
        else
            start = start.add(ZWEI);

        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):

765875739440413583163146849699704065189845501609985070259497
 765875739440413583163146849699704065189845501609985070259527
 765875739440413583163146849699704065189845501609985070259557

...

5.9 Datum und Zeit

In diesem Kapitel werden die Klassen `Date`, `TimeZone` und `GregorianCalendar` aus dem Paket `java.util` benutzt, um Datum und Uhrzeit anzugeben bzw. zu berechnen.¹⁴

Date

Ein Objekt der Klasse `Date` repräsentiert einen Zeitpunkt.

Konstruktoren:

`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: `Sat Jun 16 18:51:45 CEST 2012`

`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.

Der String `pattern` kann sich aus 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

¹⁴ Mit Java SE 8 wurde ein neues, umfangreiches *Date And Time API* eingeführt.

- E Tag als abgekürzter Text, `EEEE` 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): "05.05.2014 17:45:46"

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`.

`String getDisplayName()`

liefert eine ausführliche Angabe zur Zeitzone. Beispiel: `Zentraleuropäische Zeit`

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. 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 5-4).

Tabelle 5-4: 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

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 Kalender-Objekts als `Date`-Objekt.

`void setTime(Date d)`

setzt den Zeitpunkt des Kalender-Objekts auf den durch `d` bestimmten Zeitpunkt.

`boolean equals(Object obj)`

liefert `true`, wenn die Kalender-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`.

Programm 5.23

```
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 1.1.2014 setzen
        GregorianCalendar cal2 = new GregorianCalendar();
        cal2.set(2014, 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:

```
05.05.2014 17:52:17  
Zeitzone: America/New_York  
Tag: 5  
Monat: 5  
Jahr: 2014  
Stunde: 11  
Minute: 52  
Sekunde: 17  
01.01.2014  
10.02.2014
```

5.10 Internationalisierung

Häufig wird ein und dieselbe Softwareversion in mehreren Ländern und unterschiedlichen Sprachregionen eingesetzt. Dabei ist es wichtig, dass die Programme leicht und ohne den Quellcode ändern zu müssen an die regionalen Gegebenheiten angepasst werden können. Java erlaubt eine sprach- und länder-spezifische Datumsformatierung, Dezimaldarstellung und Einstellung von Währungssymbolen und Textelementen.

Diesen Vorgang nennt man *Internationalisierung* (engl. *internationalization*), abgekürzt mit *I18N*.¹⁵

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)

¹⁵ Im englischen Wort befinden sich 18 Buchstaben zwischen *I* und *N*.

```

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)

```

Locale

Die Klasse `java.util.Locale` verwaltet Sprach- und Länderangaben und repräsentiert damit eine geografische, politische oder kulturelle Region (`Locale = Gebietsschema`).

Konstruktoren sind:

```

Locale(String language)
Locale(String language, String country)

```

Sprach- und Ländereinstellungen werden standardmäßig von den Systemeigenschaften `user.language` und `user.country` übernommen.

Einige Methoden:

```

static Locale getDefault()
    liefert die aktuelle Standardeinstellung.

```

```

String getLanguage()
String getCountry()
    liefern den Sprach- bzw. Ländercode.

```

```

String getDisplayLanguage()
String getDisplayCountry()
String getDisplayName()
    liefern die Kurzbeschreibung von Sprache, Land bzw. beidem.

```

```

static Locale[] getAvailableLocales()
    liefert alle verfügbaren Locales.

```

`Locale` enthält eine Reihe von Konstanten für Sprachen und Länder.

Programm 5.24

```

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 = new Locale("en");
print(locale);

locale = new Locale("en", "US");
print(locale);

print(Locale.FRANCE);
print(Locale.FRENCH);
}

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 von Amerika
DisplayName: Englisch (Vereinigte Staaten von Amerika)
```

```
Locale: fr_FR
Language: fr
Country: FR
DisplayLanguage: Französisch
DisplayCountry: Frankreich
DisplayName: Französisch (Frankreich)
```

```
Locale: fr
Language: fr
Country:
DisplayLanguage: Französisch
DisplayCountry:
DisplayName: Französisch
```

Durch Aufruf von z. B.

```
java -Duser.language=en -Duser.country=US LocaleTest1
```

kann die Standardeinstellung geändert werden.

Programm 5.25 zeigt alle verfügbaren Locales.

Programm 5.25

```
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()
                + "\t" + locale.getDisplayName());
        }
    }
}
```

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 Aufrufparameter mitgegeben werden.

Die `DateFormat`-Methode

```
String format(Date date)
```

formatiert Datum/Uhrzeit.

Programm 5.26 zeigt, wie sich die Formatierung von Dezimalzahlen und Angaben zu Datum und Uhrzeit an das jeweilige Gebietsschema anpasst.

Programm 5.26

```
import java.text.DateFormat;
import java.text.DecimalFormat;
import java.util.Date;

public class LocaleTest3 {
    public static void main(String[] args) {
        DecimalFormat f = new DecimalFormat("###,##0.00");
        System.out.println(f.format(24522.4567));

        Date now = new Date();
        DateFormat[] forms = {
            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) };

        for (DateFormat df : forms) {
            System.out.println(df.format(now));
        }
    }
}
```

Ausgabebeispiel:

```
24.522,46
05.05.2014
05.05.14
05.05.2014
5. Mai 2014
Montag, 5. Mai 2014
18:03:05
18:03
18:03:05
18:03:05 MESZ
18:03 Uhr MESZ
05.05.2014 18:03:05
05.05.2014 18:03
```

Durch Aufruf von z. B.

```
java -cp bin -Duser.language=en -Duser.country=US LocaleTest3
```

kann die Standardeinstellung geändert werden.

ResourceBundle

Die Klasse `java.util.ResourceBundle` repräsentiert ein so genanntes *Ressourcenbündel*. Ein Ressourcenbündel fasst Dateien zusammen, die Schlüssel und Texte in Übersetzungen 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)
```

In unserem Beispiel befindet sich das Bündel im Verzeichnis "ressources" und hat den Namen der Klasse, die dieses Bündel nutzt. `baseName` enthält den voll qualifizierten Klassennamen:

```
ressources.InternationalTest
```

Inhalt von `InternationalTest.properties`:

```
greeting=Welcome!
firstname=First name
lastname=Last name
email=email address
```

Inhalt von `InternationalTest_de.properties`:

```
greeting=Willkommen!
firstname=Vorname
lastname=Nachname
email=E-Mail
```

Programm 5.27

```
import java.util.ResourceBundle;

public class InternationalTest {
    private static ResourceBundle res;

    static {
        res = ResourceBundle.getBundle("ressources.InternationalTest");
    }
}
```

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

5.11 Aufgaben

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.
```

2. 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 'ß' sollen  
wie "ae" bzw. "ss" behandelt werden. Alle Zeichen, die keine Buchstaben sind,  
sollen ignoriert werden.
```

3. Schreiben Sie ein Programm, das die Grundrechenarten Addition, Subtraktion, Multiplikation und Division für Gleitkommazahlen ausführen kann.
Dabei soll das Programm jeweils mit drei Parametern aufgerufen werden,
z. B. `java Rechner 114.5 + 2.5`

4. 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.

5. 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 zurückgibt. Tipp: Nutzen Sie ein `Vector`-Objekt als Zwischenspeicher.

6. Entwickeln Sie eine Variante der Lösung zu Aufgabe 5. 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.

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`.

8. Erzeugen Sie ein Array aus Objekten der Klasse `Kunde` (siehe Programm 5.7) 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.

9. Nutzen Sie die Klasse `BigInteger`, um beliebig große Fakultäten $n! = 1 * 2 * \dots * n$ zu berechnen.

10. Implementieren Sie eine Klasse `AktuellesDatum` mit einer Klassenmethode, die das aktuelle Datum wie in folgendem Beispiel als Zeichenkette liefert:
`Montag, den 5.5.2014.`

11. Berechnen Sie die Zeit in Tagen, Stunden, Minuten und Sekunden, die seit dem 1.1.2014 um 00:00:00 Uhr vergangen ist.

12. Testen Sie den Performance-Unterschied zwischen `String`-, `StringBuilder`- und `StringBuffer`-Operationen.

In einer Schleife, die genügend oft durchlaufen wird, ist bei jedem Schleifendurchgang eine Zeichenkette mit einem einzelnen Zeichen "x" zu verketten; zum einen durch Verkettung von Strings mit `+`, zum anderen durch Anwendung der `StringBuilder`- bzw. `StringBuffer`-Methode `append`.

Messen Sie die Uhrzeit unmittelbar vor und nach der Ausführung der entsprechenden Anweisungen mit `System.currentTimeMillis()`.

13. 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.

14. Das Interface Service deklariert die Methode `void work()`. Erstellen Sie eine Klasse `ServiceFactory` mit der folgenden Methode:

```
public static Service createService(String className)
```

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.

15. 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`.

16. Realisieren Sie eine Methode, die die Differenz zwischen zwei Datumsangaben ermittelt:

```
public 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.

6 Generische Typen

Klassen, 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, 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 Laufzeit-system (JVM) arbeitet mit normalen Klassen und Interfaces. Generics gibt es ab Java SE 5.

Lernziele

In diesem Kapitel lernen Sie

- wie generische Klassen, Interfaces und Methoden eingesetzt werden können,
- den Umgang mit einigen generischen Containern des Collection Frameworks.

6.1 Motivation und Definition

In der im Programm 6.1 erzeugten `Box`-Instanz können Objekte verschiedenen Typs aufgenommen werden.

Programm 6.1

```
public class Box {  
    private Object value;  
  
    public void setValue(Object value) {  
        this.value = value;  
    }  
  
    public Object getValue() {  
        return value;  
    }  
  
}  
  
public class BoxTest {  
    public static void main(String[] args) {
```

```

    Box box1 = new Box();
    box1.setValue("Hugo Meier");

    Box box2 = new Box();
    box2.setValue(4711); // Autoboxing

    System.out.println((String) box1.getValue().length());
    System.out.println((Integer) box2.getValue() + 1);
}
}

```

Ziel ist es aber, 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`) zu definieren, ist als Lösung indiskutabel.

Man will zum Zeitpunkt der Übersetzung bereits sicherstellen, dass bei der Ausführung des Programms dieses *typsicher* abläuft.

Generische Klasse, generisches Interface

Eine *generische Klasse* ist eine Klassendefinition, in der unbekannte Typen (nur Referenztypen, keine einfachen Datentypen) durch Typparameter (Platzhalter) vertreten sind. Ein *generisches Interface* ist eine Interfacedefinition, in der unbekannte Typen durch Typparameter vertreten sind. Allgemein spricht man von *generischen Typen*.

Um *Typparameter* zu bezeichnen, sollten einzelne Großbuchstaben verwendet werden (im Beispiel: `T`). Typparameter werden dem Klassen- bzw. Interfacenamen in spitzen Klammern hinzugefügt.

Beispiel: `class Box<T> ...`

Sind die Typparameter einer generischen Klasse durch konkrete *Typargumente* ersetzt, spricht man von einer *parametrisierten Klasse*, allgemein von einem *parametrisierten Typ*. Ähnliches gilt für Interfaces.

Beispiel: `Box<String> sbox ...`

Programm 6.2 ist die Lösung zu der obigen Problemstellung. Die hier erzeugte Box kann nur Objekte vom Typ `String` aufnehmen.

Programm 6.2

```

public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }
}

```

```
public T getValue() {
    return value;
}

public class BoxTest {
    public static void main(String[] args) {
        Box<String> box1 = new Box<String>();
        box1.setValue("Hugo Meier");

        Box<Integer> box2 = new Box<Integer>();
        box2.setValue(4711);

        System.out.println(box1.getValue().length());
        System.out.println(box2.getValue() + 1);

        // box1.setValue(4711);
    }
}
```

Die im Programm 6.2 auf Kommentar gesetzte Anweisung `sbox.setValue(4711);` führt zu einem Übersetzungsfehler. Das Programm hat sich sogar vereinfacht, da der Downcast auf `String` bzw. `Integer` nun nicht mehr nötig ist.

Diamond Operator

Ab Java SE 7 kann der Typparameter in der `new`-Anweisung bis auf die spitzen Klammern `<>` (Diamond Operator) weggelassen werden, z. B.

```
Box<String> sbox = new Box<>();
```

Der Compiler ermittelt den korrekten Typ aus dem Kontext.

Aus einer generischen Klasse werden auf diesem Weg viele unterschiedliche parametrisierte Klassen "generiert". Zu einer generischen Klasse gehört genau eine Bytecode-Datei mit Endung `.class`. Diese wird von allen zugehörigen parametrisierten Klassen benutzt.

Generische Klassen können parametrisierte bzw. generische Interfaces implementieren: z. B.

```
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 möglich: z. B. `class Demo<T, U>`

6.2 Typparameter mit Einschränkungen

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` selbst und jede Subklasse von `Number` eingesetzt werden, also z. B. `Integer`.

`extends` wird gleichermaßen für Klassen und Interfaces genutzt. Typargumente in parametrisierten Typen müssen zum Typebound kompatibel sein.

Zum Begriff "kompatibel"

Ein Typ T heißt *kompatibel* zum Typ U , wenn ein Wert vom Typ T einer Variablen vom Typ U zugewiesen werden kann.

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)`.¹

Programm 6.3

```
public class Box<T extends Number> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

public class Konto implements Comparable<Konto> {
    private int id;
    private int saldo;
```

¹ Siehe auch Kapitel 5.7.

```
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;
}

public int compareTo(Konto k) {
    if (id == k.id)
        return 0;
    if (id < k.id)
        return -1;
    return 1;
}
}

public class NewBox<T extends Comparable<T>> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

public class BoxTest {
    public static void main(String[] args) {
        // Integer und Double sind Subklassen von Number
        Box<Integer> ibox = new Box<>();
        ibox.setValue(4711);

        Box<Double> dbox = new Box<>();
        dbox.setValue(123.45);

        // Folgende Anweisungen führen zu Übersetzungsfehlern:
        // Box<String> sbox = new Box<>();
        // Box<Object> sbox = new Box<>();

        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()));
}
}

```

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.

Invarianz

Aus "A ist kompatibel zu B" folgt *nicht* "C<A> ist kompatibel zu C".

Die Kompatibilität der Typargumente überträgt sich *nicht* auf die parametrisierten Typen (*Invarianz*).

So ist beispielsweise die Zuweisung

```
Box<Number> box = new Box<Integer>();
```

nicht möglich. Damit wird verhindert, dass Objekte, die nicht vom Typ `Integer` sind, hinzugefügt werden können.

Bei Arrays ist das Verhalten anders:

`Integer` ist kompatibel zu `Number`, ebenso ist ein `Integer`-Array kompatibel zu einem `Number`-Array.

Beispiel:

```
Number[] a = new Integer[1];
a[0] = new Double(3.14);
```

Dieser Code lässt sich fehlerfrei kompilieren. Erst zur Laufzeit wird eine `ArrayStoreException` ausgelöst. Arrays überprüfen die Elementtypen zur Laufzeit.

6.3 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 eines generischen Typs 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 so genannte *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. `Box` ist der Raw Type zu `Box<T>`.

Ein parametrisierter Typ ist zu seinem Raw Type kompatibel.

Beispiel:

```
Box box = new Box<Integer>();
```

Programm 6.4

```
public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

public class BoxTest {
    @SuppressWarnings({ "rawtypes", "unchecked" })
    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(box.getValue());
    }
}
```

6.4 Wildcard-Typen

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.

Bivarianz

Zu einem unbeschränkten *Wildcard-Typ* (`C<?>`) sind alle parametrisierten Typen des gleichen generischen Typs kompatibel (*Bivarianz*).

Beispiel: Alle parametrisierten "Box-Typen" (z. B. `Box<Integer>`, `Box<Double>`) sind zu `Box<?>` kompatibel.

Weder *lesender* noch *schreibender* Zugriff auf eine Instanz eines solchen Wildcard-Typs ist erlaubt, wie Programm 6.5 zeigt. Einzige Ausnahmen sind: Rückgabetyp `Object` beim Lesen, Zuweisung von `null`.

Programm 6.5

```
public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

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

Wildcard-Typen können durch eine obere Grenze (*Upper-Typebound*) eingeschränkt werden: `C<? extends B>`.

Ist `B` eine Klasse, so bezieht sich das Typargument nur auf `B` selbst und auf Klassen, die von `B` abgeleitet sind. Ist `B` ein Interface, so bezieht sich das Typargument auf Klassen, die `B` implementieren. Mehrfache Upper-Typebounds sind auch möglich, z. B. `C<? extends K1 & I1 & I2>`.

Covarianz

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 beziehen. Dies demonstriert Programm 6.6.

Programm 6.6

```

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();
    }
}

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>`. Die `Box`-Methode `copyFrom` zeigt, dass Wildcard-Typen mit oberer Grenze auch in generischen Klassen verwendet werden können. Upper-Typebound ist hier der Typparameter `T`. Somit kann `copyFrom` für den parametrisierten Typ `Box<Number>` Werte aus parametrisierten Typen wie `Box<Integer>` oder `Box<Double>` kopieren.

Untere Grenze

Wildcard-Typen können durch eine untere Grenze (*Lower-Typebound*) eingeschränkt werden: `C<? super B>`.

Das Typargument bezieht sich hier nur auf die Klasse `B` selbst und auf Klassen, die Superklassen von `B` sind.

Contravarianz

Ist `B` kompatibel zu `A`, so ist `C<A>` kompatibel zu `C<? super B>` (*Contravarianz*).

Auf eine Instanz eines solchen Wildcard-Typs ist kein *lesender* Zugriff erlaubt. Einzige Ausnahme ist das Lesen mit Rückgabetyp `Object`. *Schreibende* Operationen können sich auf den Lower-Typebound beziehen. Dies demonstriert Programm 6.7.

Programm 6.7

```

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);
    }
}

```

```

public class BoxTest {
    public static void main(String[] args) {
        Box<Object> obox = new Box<>();
        obox.setValue("Hugo");

        Box<? super Number> 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<Object>` ist kompatibel zu `Box<? super Number>`. Die `Box`-Methode `copyTo` zeigt, dass Wildcard-Typen mit unterer Grenze auch in generischen Klassen verwendet werden können. Lower-Typebound ist hier der Typparameter `T`.

Tabelle 6-1: Übersicht Wildcard-Typen

Wildcard-Typ	Varianz	Lesen	Schreiben
?	Bivarianz	– ¹⁾	– ²⁾
? extends B	Covarianz	+ ³⁾	– ²⁾
? super B	Contravarianz	– ¹⁾	+ ³⁾

¹⁾ Ausnahme: Rückgabetypr `Object` ²⁾ Ausnahme: Zuweisung von null ³⁾ bezogen auf B

6.5 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 Rückgabetypr.

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 Box in Programm 6.8 enthält eine generische Klassenmethode, die die Inhalte zweier Boxen untereinander austauscht.

Programm 6.8

```
public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

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);
    }
}

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());
        System.out.println();

        Utils.<Integer>exchange(box1, box2);

        System.out.println("Box1: " + box1.getValue());
        System.out.println("Box2: " + box2.getValue());
        System.out.println();

        Utils.exchange(box1, box2);

        System.out.println("Box1: " + box1.getValue());
        System.out.println("Box2: " + box2.getValue());
    }
}
```

6.6 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. Autoboxing umgeht diese Einschränkung in eleganter Weise.²
- Klassenattribute und Klassenmethoden einer generischen Klasse können keine Typparameter der eigenen Klasse verwenden. Eigene Typparameter sind möglich (generische Methoden).
- Der Operator `instanceof` kann nicht mit Typparametern bzw. parametrisierten Typen verwendet werden.
- Konstruktoraufrufe 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.

6.7 Generische Container

Ab Java SE 5 sind alle Container des *Collection Frameworks* als generische Typen definiert. Container wie z. B. `Vector` und `Hashtable`³ sind hierfür bestens geeignet, da ihre Funktionsweise in der Regel nicht vom Typ des Inhalts abhängt. Zudem erreicht man, dass alle Elemente eines Containers vom gleichen vorgegebenen Typ sind.

In diesem Kapitel behandeln wir nur einen Ausschnitt aus dem *Collection Framework*.

6.7.1 Listen

Eine *Liste* ist eine sequentielle Anordnung von Elementen in einer festen Reihenfolge.

Collection

Alle Listen implementieren das Interface `java.util.Collection<T>`.

² Siehe Kapitel 5.2.

³ Siehe Kapitel 5.4.

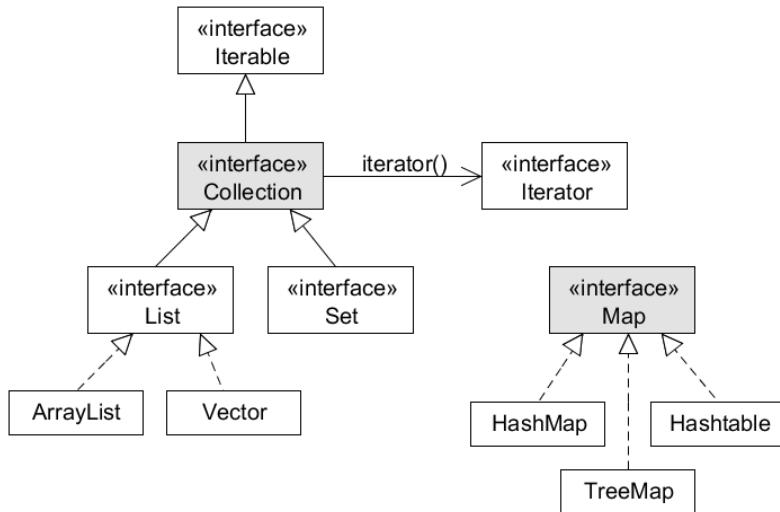


Abbildung 6-1: Collection Framework

Collection-Methoden sind:

`boolean add(T t)`

fügt t am Ende der Liste 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.

Iterable, Iterator

`Iterator<T> iterator()`

liefert einen so genannten *Iterator* zum schrittweisen Durchlaufen des Containers. Diese Methode ist im Interface `java.lang.Iterable<T>` deklariert, das von `Collection` erweitert wird.

Das Interface `java.util.Iterator<T>` deklariert die folgenden Methoden:

`boolean hasNext()`

prüft, ob es noch nicht besuchte Elemente gibt.

`T next()`
liefert das nächste Element.

`void remove()`
entfernt das zuletzt besuchte Element.

Alle Container, die das Interface `Iterable` implementieren, können mit `foreach`-Schleifen durchlaufen werden:

`for (Typ name : ref) { ... }`

Typ entspricht dem Elementtyp der Collection `ref`.

List

Das Interface `java.util.List<T>` erweitert `Collection<T>`.

Einige Methoden (außer denen aus `Collection`) sind:

`void add(int i, T t)`
fügt `t` an der Position `i` ein.

`T get(int i)`
liefert das Element an der Position `i`.

`T set(int i, T t)`
ersetzt das Element an der Position `i` durch `t` und liefert das alte Element zurück.

`T remove(int i)`
entfernt das Element an der Position `i` und liefert es zurück.

`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.

Die Klasse `java.util.ArrayList<T>` implementiert `List<T>`. Der Konstruktor `ArrayList()` erzeugt eine leere Liste.⁴

Programm 6.9

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListTest {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
```

⁴ Die bereits im Kapitel 5.4.1 ohne Berücksichtigung ihrer Generizität behandelte Klasse `java.util.Vector<T>` implementiert ebenfalls `List<T>`.

```

list.add("Anton");
list.add("Emil");
list.add("Fritz");
list.add("Hugo");

for (String s : list) {
    System.out.println(s);
}

System.out.println();

list.add(3, "Gustav");
list.remove(0);

for (String s : list) {
    System.out.println(s);
}
}
}

```

Ausgabe des Programms:

Anton
 Emil
 Fritz
 Hugo

Emil
 Fritz
 Gustav
 Hugo

6.7.2 Schlüsseltabellen

Eine *Schlüsseltabelle* ist eine Menge von Schlüssel-Wert-Paaren. Schlüssel müssen eindeutig sein.

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.

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

Set<K> keySet()
liefert alle Schlüssel als Menge. Eine *Menge* enthält im Gegensatz zur Liste keine Duplikate und es existiert keine bestimmte Reihenfolge. Das Interface `java.util. Set<T>` erweitert `Collection<T>`.

Collection<V> values()
liefert die Werte der Schlüsseltabelle als `Collection`-Objekt.

Die von den beiden letzten Methoden zurückgegebenen Objekte sind *Sichten* auf die Schlüsseltabelle. Änderungen der Schlüsseltabelle führen zu Änderungen dieser Sichten und umgekehrt. 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 Interface⁵ von `Map` und deklariert die Methoden:

K getKey()
liefert den Schlüssel des Eintrags.

V getValue()
liefert den Wert des Eintrags.

Die Klasse `java.util.HashMap<K,V>` implementiert `Map<K,V>`. Der Konstruktor `HashMap()` erzeugt eine leere Tabelle.

⁵ Siehe Kapitel 3.8.

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.⁶

Programm 6.10 baut eine Gehaltstabelle aus den Schlüssel-Wert-Paaren (Name, Gehalt) auf. Diese Tabelle wird auf zwei unterschiedliche Arten durchlaufen.

Programm 6.10

```
import java.util.Map;
import java.util.TreeMap;

public class MapTest {
    public static void main(String[] args) {
        Map<String, Double> map = new TreeMap<>();
        map.put("Meier, August", 5000.);
        map.put("Schmitz, Anton", 4500.);
        map.put("Balder, Hugo", 4700.);
        map.put("Schulze, Wolfgang", 4500.);

        // Variante 1
        for (String key : map.keySet())
            System.out.println(key + ": " + map.get(key));

        System.out.println();

        // Variante 2
        for (Map.Entry<String, Double> e : map.entrySet())
            System.out.println(e.getKey() + ": " + e.getValue());
    }
}
```

Ausgabe des Programms:

```
Balder, Hugo: 4700.0
Meier, August: 5000.0
Schmitz, Anton: 4500.0
Schulze, Wolfgang: 4500.0
...
```

⁶ Die bereits im Kapitel 5.4.2 ohne Berücksichtigung ihrer Generizität behandelte Klasse `java.util.Hashtable<K,V>` implementiert ebenfalls `Map<K,V>`.

6.8 Aufgaben

1. Definieren Sie die generische Klasse `Paar<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`.
2. Definieren Sie das generische Interface `Markierbar<S>`, das die beiden folgenden Methoden deklariert:

```
void setMarke(S m);
S getMarke();
```

Die Klasse `Paar<T,U>` aus Aufgabe 1 soll dieses Interface mit dem Typargument `String` implementieren. Die Klasse `Box<T>` aus Programm 6.2 soll ebenso dieses Interface implementieren, wobei der Typparameter `T` an den Typparameter des Interfaces gekoppelt werden soll.

3. Definieren Sie die generische Klasse `ZahlenPaar`, die von der Klasse `Paar<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 `ZahlenPaar` soll eine Methode enthalten, die die Summe der beiden Zahlen des Paars als `double`-Zahl zurückgibt.
4. Implementieren Sie die Methode

```
public static void print(Paar<?,?> p),
```

die die beiden Werte `a` und `b` eines Paars in der Form `(a, b)` ausgibt. Nutzen Sie die Klasse `Paar<T,U>` aus Aufgabe 1.
5. Implementieren Sie die Methode

```
public static double sum(Paar<? extends Number, ? extends Number> p),
```

die die beiden Werte eines Paars addiert. Nutzen Sie die Klasse `Paar<T,U>` aus Aufgabe 1 und die `Number`-Methode `double doubleValue()`.
6. Lösen Sie Aufgabe 5 durch Implementierung einer generischen Methode (ohne Verwendung von Wildcard-Typen).
7. Erstellen Sie die generische Variante zu Programm 5.11 mit Hilfe der generischen Klasse `Vector<T>` und des generischen Interfaces `Enumeration<T>`.
8. Implementieren Sie eine statische Methode `printList`, die die Elemente einer beliebigen Liste vom Typ `List` am Bildschirm ausgibt:

```
public static void printList(List<?> list)
```

Verwenden Sie hierzu die Methoden des Interfaces `Iterator`.

9. Erstellen Sie die generische Variante zu Programm 5.12 mit Hilfe der Klasse `Hashtable<K,V>` und des Interfaces `Enumeration<T>`.

10. Erzeugen Sie eine Instanz der generischen Klasse `ArrayList`, die Objekte vom Typ `Konto` aus Programm 5.19 speichert. 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`:

```
public static <T extends Comparable<? super T>> void sort(  
    List<T> list)
```

11. Erzeugen Sie einen generischen Stack mit Hilfe der Klasse `ArrayList`. Folgende Methoden sind zu implementieren:

```
public boolean empty()
```

liefert `true`, wenn der Stack leer ist, ansonsten `false`.

```
public T push(T item)
```

legt `item` oben auf den Stack und gibt `item` zurück.

```
public T pop()
```

löscht das oberste Element des Stacks und liefert es zurück.

```
public T peek()
```

liefert das oberste Element des Stacks.

12. 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()`.

13. Erzeugen Sie eine generische Liste, die auf einem Array fester Länge basiert.

Konstruktor:

```
Liste(int length)
```

Folgende Methoden sollen implementiert werden:

```
int size()
```

```
void add(T t)
```

```
T get(int i)
```

Tipp: Da Konstruktoraufrufe mit Typparametern nicht verwendet werden können (siehe Kapitel 6.6), ist

```
list = (T[]) new Object[length];
```

zu nutzen, wenn `list` wie folgt vereinbart ist: `T[] list`

7 Lambda-Ausdrücke

Bis hin zur Version Java SE 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 Klasse¹ definiert werden, die die in Frage stehende Methode enthält. Diese Instanz wird dann als Argument beim Aufruf übergeben. Programm 7.1 demonstriert diese Vorgehensweise.

Ab Java SE 8 können so genannte *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.²

Da Lambda-Ausdrücke in Java mit speziellen Interfaces (so genannten 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,
- wie Lambda-Ausdrücke die Programmierung vereinfachen können.

Im folgenden Beispiel wird die Methode `calculate` an die Methode `someMethod` über den Umweg "anonyme Klasse" übergeben.

Programm 7.1

```
public interface X {  
    int calculate(int a, int b);  
}  
  
public class A {  
    private int i;
```

¹ Siehe Kapitel 3.9.

² Der Lambda-Kalkül wurde in den 1930er Jahren von den US-amerikanischen Mathematikern Alonzo Church und Stephen Cole Kleene eingeführt.

```

public A(int i) {
    this.i = i;
}

public void someMethod(X x) {
    int result = i + x.calculate(2, 3);
    System.out.println(result);
}

public static void main(String[] args) {
    A a = new A(1);

    a.someMethod(new X() {
        public int calculate(int a, int b) {
            return a + b;
        }
    });
}
}

```

Ausgabe des Programms:

6

7.1 Funktionsinterfaces

Ein *Funktionsinterface* ist ein Interface mit genau einer abstrakten Methode (Ausnahme siehe nächster Absatz). Default-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.³

Beispiel:

```

@FunctionalInterface
public interface X {
    void m();

    boolean equals(Object obj);

    default void a() {
        System.out.println("Hallo");
    }
}

```

³ Annotationen werden hier nicht weiter behandelt. Näheres hierzu findet man im Kapitel 15.3.

```
    static void b() {  
        System.out.println("Welt");  
    }  
}
```

Bekannte Interfaces wie `java.lang.Iterable`⁴, `java.lang.Runnable`⁵ und `java.awt.event.ActionListener`⁶ sind Funktionsinterfaces.

Das Paket `java.util.function` enthält eine Reihe weiterer Funktionsinterfaces, beispielsweise:

```
public interface Predicate<T> {  
    boolean test(T t);  
    ...  
}
```

Dieses kann zum Filtern von Elementen einer Liste verwendet werden.

Programm 7.2

```
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;  
            }  
        });  
    }  
}
```

⁴ Siehe Kapitel 6.7.

⁵ Siehe Kapitel 9.1.

⁶ Siehe Kapitel 10.6.

```

        for (int n : result) {
            System.out.println(n);
        }
    }
}

```

Ausgabe des Programms:

```

2
4
6
8

```

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.

Im Folgenden sind die abstrakten Methoden einiger Funktionsinterfaces aus `java.util.function` aufgeführt.

`Function<T,R>`
`R apply(T t)`
 bildet das Argument auf einen anderen Wert ab.

`BinaryOperator<T>`
`T apply(T t1, T t2)`
 verknüpft zwei Argumente zu einem Wert.

`Consumer<T>`
`void accept(T t)`
 verarbeitet das Objekt t.

`Supplier<T>`
`T get()`
 liefert ein Ergebnis.

7.2 Lambda-Ausdrücke

Ein *Lambda-Ausdruck* repräsentiert eine "anonyme Funktion". Er besteht aus einer Parameterliste, dem Operator `->` und einem Methodenrumpf:

`(parameter) -> { body }`

Beispiele:

```

x -> x * x
(x, y) -> x + y
(x, y) -> {return x + y;}
(String s1, String s2) -> System.out.println(s1.length() + s2.length())
() -> 4711

```

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.
- 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.

Funktionsinterfaces können mit Hilfe von Lambda-Ausdrücken implementiert werden. Ein Lambda-Ausdruck ist zuweisungskompatibel zu jedem Funktionsinterface, dessen abstrakte Methode die passende Parameterliste und den passenden Rückgabetyp hat.

Beispiel:

Das Funktionsinterface `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.

`(x, y) -> x - y` ist zuweisungskompatibel zu `Comparator<Integer>`, denn zwei Zahlen als Parameter liefern eine Zahl als Ergebnis:

```
Comparator<Integer> comp = (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 (siehe Programm 7.2) zu ersetzen.

Programm 7.3

```

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);

        for (int n : result) {
            System.out.println(n);
        }
    }
}

```

Die Ausgabe des Programms gleicht der Ausgabe des Programms 7.2.

Ein Lambda-Ausdruck kann in verschiedenen Kontexten genutzt werden. Die Parametertypen werden aus dem Zuweisungskontext ermittelt (*Typ-Inferenz*).

Programm 7.4

```

import java.util.function.BinaryOperator;

public class ArrayTest {
    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);  
    }  
}
```

Ausgabe des Programms:

```
abc  
6
```

Hier wird der Lambda-Ausdruck $(s1, s2) \rightarrow s1 + s2$ verwendet, um einen `BinaryOperator` zu implementieren, der Zahlen aufaddiert bzw. Strings verkettet.

Der Rumpf eines Lambda-Ausdrucks hat Zugriff auf alle Attribute (Instanz- und Klassenvariablen) der umgebenden Klasse und die *unveränderlichen* lokalen Variablen der Definitionsumgebung. Unveränderlich bedeutet, dass die Variable ohne Fehlermeldung des Compilers mit `final` versehen werden könnte.

Programm 7.5

```
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
```

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 ganz anderen Umgebung, in der die Methode (hier `run`) aufgerufen wird, wieder zur Verfügung.

7.3 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:

Klassenname::*Klassenmethodenname*
Klassenname::new
Klassenname::*Instanzmethodenname*
Objektreferenz::*Instanzmethodenname*

Programm 7.6

Hier wird die Referenz auf eine statische Methode genutzt:

Klassenname::*Klassenmethodenname*

Rückgabetyp und Parametertypen müssen zur abstrakten Methode des Funktionsinterfaces passen.

```
public class MyPredicates {  
    public static boolean isEven(Integer n) {  
        return n % 2 == 0;  
    }  
  
    public static boolean isOdd(Integer n) {  
        return n % 2 != 0;  
    }  
}  
  
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);
    for (int n : result1) {
        System.out.println(n);
    }

    List<Integer> result2 = filter(list, MyPredicates::isOdd);
    for (int n : result2) {
        System.out.println(n);
    }
}
```

Ausgabe des Programms:

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

MyPredicates::isEven ist gleichsam eine Abkürzung für den Lambda-Ausdruck

```
    Integer n -> MyPredicates.isEven(n)
```

Gleichtes gilt für isOdd.

Programm 7.7

Der Konstruktor der Klasse Person mit dem ParameterTyp String ist kompatibel zur abstrakten Methode des Funktionsinterfaces Function. Aus einem String wird mit Hilfe des Konstruktors ein Person-Objekt erzeugt.

Der zweite Teil des Beispiele zeigt, dass auch Referenzen auf Instanzmethoden genutzt werden können. Argument der apply-Methode des Funktionsinterfaces Function ist die Referenz auf ein Person-Objekt.

Person::getName bildet ein Person-Objekt auf einen String ab.

Alternativ kann hier auch der Lambda-Ausdruck p -> p.getName() eingesetzt werden.

```
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;
}
}

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);
        for (Person p : persons) {
            System.out.println(p.getName());
        }

        names = map(persons, Person::getName);
        for (String s : names) {
            System.out.println(s);
        }

        names = map(persons, p -> p.getName());
        for (String s : names) {
            System.out.println(s);
        }
    }
}
```

Ausgabe des Programms:

```
Hugo
Emil
Anton
Hugo
Emil
Anton
Hugo
Emil
Anton
```

Programm 7.8

Dieses Programm zeigt, dass auch die Referenz auf die Instanzmethode für ein spezielles Objekt (`System.out` bzw. `t`) genutzt werden kann.

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class ConsumingTest {
    private String header;

    public ConsumingTest(String header) {
        this.header = header;
    }

    public void output(String s) {
        System.out.println(header + " " + s);
    }

    public static <T> void forEach(List<T> list, Consumer<T> consumer) {
        for (T item : list) {
            consumer.accept(item);
        }
    }

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Hugo");
        list.add("Emil");
        list.add("Anton");

        forEach(list, n -> System.out.println(n));
        forEach(list, System.out::println);

        list.forEach(n -> System.out.println(n));
        list.forEach(System.out::println);

        ConsumingTest t = new ConsumingTest("#");
        list.forEach(t::output);
    }
}
```

Ausgabe des Programms:

```
Hugo
Emil
Anton
Hugo
Emil
Anton
Hugo
Emil
Anton
Hugo
Emil
Anton
```

```
# Hugo
# Emil
# Anton
```

Die `PrintStream`-Methode `println` ist kompatibel zur Methode `accept` des Funktionsinterfaces `Consumer`, gleiches gilt für die Instanzmethode `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.

7.4 Ein Beispiel zum Stream-API für Collection-Klassen

Das Interface `java.util.Collection<T>` enthält die folgende Default-Methode

```
default Stream<T> stream()
```

Das Interface `java.util.stream.Stream` repräsentiert eine Folge von Elementen und transformiert diese mit Hilfe von Operationen wie `filter`, `map`, `sorted` in einen anderen Stream (*intermediate operation*) oder erzeugt ein Endergebnis (*terminal operation*) wie `collect`, `count`, `forEach`, `reduce`. Ein Stream speichert keine Daten, sondern transportiert Daten durch eine Pipeline von Operationen. Die Verarbeitung der einzelnen Schritte wird erst mit Beginn der *terminal operation* ausgelöst. Ein Stream kann das Ergebnis nur einmal bereitstellen und somit nicht wiederverwendet werden. Die zugrunde liegende Datenquelle wird nicht verändert. Bei Verwendung von Streams wird nur beschrieben, *was* zu tun ist, nicht aber *wie*.

Programm 7.9

```
public class Artikel {
    private int id;
    private char typ;
    private double preis;

    public Artikel() {}

    public Artikel(int id, char typ, double preis) {
        this.id = id;
        this.typ = typ;
        this.preis = preis;
    }

    public int getId() {
        return id;
    }
}
```

```
public void setId(int id) {
    this.id = id;
}

public char getTyp() {
    return typ;
}

public void setTyp(char typ) {
    this.typ = typ;
}

public double getPreis() {
    return preis;
}

public void setPreis(double preis) {
    this.preis = preis;
}

public String toString() {
    return id + " " + typ + " " + preis;
}
}

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class StreamTest {
    public static void main(String[] args) {
        List<Artikel> list = new ArrayList<Artikel>();
        list.add(new Artikel(4712, 'A', 12.));
        list.add(new Artikel(4714, 'A', 20.));
        list.add(new Artikel(4713, 'B', 10.));
        list.add(new Artikel(4715, 'B', 10.));
        list.add(new Artikel(4711, 'A', 10.));

        // Sortieren und ausgeben
        list.stream().sorted((a, b) -> a.getId() - b.getId())
            .forEach(System.out::println);

        // Filtern und zählen
        long count = list.stream().filter(a -> a.getTyp() == 'A')
            .count();
        System.out.println(count);

        // Filtern und Ergebnis als Liste erzeugen
        List<Artikel> filtered = list.stream().filter(a -> a.getTyp() == 'A')
            .collect(Collectors.toList());

        // 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 12.0
4713 B 10.0
4714 A 20.0
4715 B 10.0
3
Durchschnittspreis für Artikel vom Typ A: 14.0
```

Erläuterung der hier verwendeten Stream-Methoden:

- `sorted` erwartet einen `Comparator` als Argument und sortiert die Elemente des Stream (*intermediate operation*). Ist kein Argument angegeben, wird gemäß der natürlichen Ordnung sortiert.
- `forEach` verlangt ein `Consumer`-Objekt als Argument (*terminal operation*).
- `filter` filtert einzelne Elemente des Streams anhand eines `Predicate`-Objekts (*intermediate operation*).
- `count` liefert die Anzahl Elemente im Stream (*terminal operation*).
- Mit `collect(java.util.stream.Collectors.toList())` wird ein Stream in eine Liste überführt (*terminal operation*).
- `map` wendet die angegebene Operation (`Function`-Objekt) auf jedes einzelne Element an (*intermediate operation*).
- `reduce` reduziert die Liste aller Preise zu einem Ergebniswert (*terminal operation*). Dabei werden, ausgehend von einem Startwert, zwei Elemente zu einem Zwischenwert verknüpft (`BinaryOperator`), der wiederum mit dem folgenden Element verknüpft wird.

Das Stream-API bietet darüber hinaus noch zahlreiche Methoden. 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.

7.5 Aufgaben

1. 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.
2. Eine Reihe von Guthaben soll verzinst werden. Ermitteln Sie die Guthaben nach Verzinsung und geben Sie diese der Reihe nach aus. Lösen Sie das Problem zunächst traditionell und dann mit Hilfe der Stream-Methoden `map`

und `forEach`. Um ein Array von Zahlen als Liste zu erhalten, können Sie die `java.util.Arrays`-Methode `static <T> List<T> asList(T... a)` verwenden.

3. Modifizieren Sie die Lösung zu Aufgabe 2, um als Ergebnis die Summe der verzinsten Guthaben zu erhalten. Lösen Sie das Problem zunächst traditionell und dann mit Hilfe der `Stream`-Methoden `map` und `reduce`.
4. Mit Hilfe von `Stream`-Methoden sollen als Kommandozeilenparameter 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)
```

5. Gegeben sei eine Liste von `Person`-Objekten. Mit Hilfe von `Stream`-Methoden sollen die Namen dieser Personen in Großbuchstaben gewandelt und dann sortiert in einem Array ausgegeben werden. Nutzen Sie die `Stream`-Methode `Object[] toArray()`.
6. Mit Hilfe von `Stream`-Methoden soll zu einer vorgegebenen 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 usw.

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.

7. Implementieren Sie das Funktionsinterface

```
Function<Consumer<String>, Consumer<String>>
```

mit Hilfe der geschachtelten Lambda-Ausdrücke:

```
(Consumer<String> c) -> { return (String t) -> { c.accept(t);  
c.accept(t); } }
```

oder kurz

```
c -> t -> { c.accept(t); c.accept(t); }
```

Testen Sie das Funktionsinterface mit unterschiedlichen Implementierungen für das Interface `Consumer`, z. B. Ausgabe eines Strings oder Hinzufügen eines Strings in ein `List`-Objekt.

8. 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`.

9. 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.

Beispiel: Hugo, Fritz, John

Implementieren Sie für `reduce` das Funktionsinterface `BinaryOperator<String>` so, dass zwei Strings kombiniert werden.

8 Ein- und Ausgabe

Java bietet eine umfangreiche Bibliothek von Klassen und Interfaces im Paket `java.io` 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

- den Unterschied zwischen byte- und zeichenorientierten Datenströmen kennen,
- welche grundlegenden Methoden zum Schreiben und Lesen von Dateien existieren,
- wie komplette Objekte dauerhaft in Dateien aufbewahrt werden können,
- wie Dateien komprimiert werden können.

8.1 Die Klasse File

Objekte der Klasse `java.io.File` repräsentieren Dateien und Verzeichnisse.

Konstruktoren:

```
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.

Beispiel:

```
File file = new File("C:\\\\Test\\\\info.txt");
```

Hier muss für Microsoft Windows der Backslash doppelt angegeben werden.¹ Es kann aber auch der normale Schrägstrich genutzt werden, also beispielsweise

```
C:/Test/info.txt
```

¹ Siehe Tabelle 2-3.

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 ".." (aktuelles 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.

Programm 8.1

```
import java.io.File;
import java.io.IOException;
import java.util.Date;

public class FileInfo {
    public static void main(String[] args) throws IOException {
        File file = new File(args[0]);
        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()));
    }
}

```

Der Aufruf

```
java -cp bin FileInfo ..\P01\src\FileInfo.java
```

im Verzeichnis P01 liefert z. B.

```

Name: FileInfo.java
Path: ..\P01\src\FileInfo.java
AbsolutePath: D:\GKJava\Kap08\P01..\P01\src\FileInfo.java
CanonicalPath: D:\GKJava\Kap08\P01\src\FileInfo.java
Parent: ..\P01\src
exists: true
canRead: true
canWrite: true
isFile: true
isDirectory: false
isAbsolute: false
length: 947
lastModified: Fri Feb 07 11:10:52 CET 2014

```

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.

Zugriffe 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.

Interface FileFilter

`java.io.FileFilter` ist ein Interface 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.

Programm 8.2

```
import java.io.File;
import java.io.FileFilter;

public class FilterTestV1 {
    public static void main(String[] args) {
        final String dir = args[0];
        final String suffix = args[1];

        File file = new File(dir);
        if (!file.isDirectory()) {
            System.err.println(dir + " ist kein Verzeichnis");
            System.exit(1);
        }

        FileFilter filter = new FileFilter() {
            public boolean accept(File pathname) {
                return pathname.isFile()
                    && pathname.getName().endsWith("." + suffix);
            }
        };

        File[] list = file.listFiles(filter);
        for (File f : list) {
            System.out.println(f.getName());
        }
    }
}
```

`FileFilter` ist ein Funktionsinterface. In der folgenden Programmvariante wird dieses Interface mit einem *Lambda-Ausdruck* implementiert.

```
import java.io.File;
import java.io.FileFilter;

public class FilterTestV2 {
    public static void main(String[] args) {
        final String dir = args[0];
```

```
final String suffix = args[1];

File file = new File(dir);
if (!file.isDirectory()) {
    System.err.println(dir + " ist kein Verzeichnis");
    System.exit(1);
}

FileFilter filter = p -> p.isFile()
    && p.getName().endsWith("." + suffix);

File[] list = file.listFiles(filter);
for (File f : list) {
    System.out.println(f.getName());
}
}
```

8.2 Datenströme

Die sequentielle Ein- und Ausgabe wird mittels so genannter 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.²

Umlenkung

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

² Siehe auch Kapitel 5.5.

Angabe von `2>datei` 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. -senke (z. B. Datei, Array, String), nach der Art der Übertragung (z. B. gepuffert, gefiltert) und nach der Art der Dateneinheiten, die sie behandeln, unterschieden werden.

8.2.1 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 8-1 gibt eine Übersicht über die wichtigsten Klassen. Die Namen der abstrakten Klassen sind kursiv gedruckt. Die Vererbungshierarchie wird durch Einrückungen wiedergegeben.

Tabelle 8-1: 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-Eingabe- bzw. -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 komplette Objekte schreiben bzw. wieder rekonstruieren.

PipedInputStream/PipedOutputStream

bieten Methoden, um Daten zwischen zwei unabhängig laufenden Programmen (*Threads*) über so genannte *Pipes* auszutauschen.³

SequenceInputStream

kann aus mehreren Eingabeströmen 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.

³ Siehe Kapitel 9.3.

```
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.
```

```
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.
```

8.2.2 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 8-2 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`. Wenn keine Codierung spezifiziert ist, wird eine voreingestellte Codierung für diese Umwandlung (z. B. Windows-Codepage 1252, siehe Java-Systemeigenschaft `file.encoding`) benutzt.

FileReader/FileWriter

liest aus einer bzw. schreibt in eine Datei.

PipedReader/PipedWriter

bieten Methoden, um Daten zwischen zwei unabhängig laufenden Programmen (*Threads*) über so genannte *Pipes* auszutauschen.

StringReader/StringWriter

liest Zeichen aus einem String bzw. schreibt Zeichen in einen String.

PrintWriter

gibt Werte verschiedener Datentypen im Textformat aus.

Tabelle 8-2: 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>

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.

`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 Ausgabestrom.

`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.

8.3 Dateien byteweise kopieren

Mit Hilfe der Klassen `FileInputStream` und `FileOutputStream` kann byteorientiert auf Dateien des Betriebssystems zugegriffen werden.

FileInputStream

```
FileInputStream(File file) throws FileNotFoundException  
FileInputStream(String filename) throws FileNotFoundException  
erzeugen jeweils einen FileInputStream für die angegebene Datei.
```

FileOutputStream

```
FileOutputStream(File file) throws FileNotFoundException  
FileOutputStream(String filename) throws FileNotFoundException  
FileOutputStream(String filename, boolean append)  
throws FileNotFoundException  
erzeugen jeweils einen FileOutputStream 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`.

Programm 8.3 kopiert eine Datei byteweise.

Programm 8.3

```
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(args[0]);  
            out = new FileOutputStream(args[1]);  
  
            int c;  
            while ((c = in.read()) != -1) {  
                out.write(c);  
            }  
        } finally {  
            try {  
                if (in != null) {  
                    in.close();  
                }  
            } catch (IOException e) {}  
            try {  
                if (out != null) {  
                    out.close();  
                }  
            } catch (IOException e) {}  
        }  
    }  
}
```

```

        out.flush();
        out.close();
    }
} catch (IOException e) {
}
}
}
}
```

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 SE 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.java`).

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.net.Socket`, `java.net.ServerSocket` sowie die Klassen, die `java.sql.Connection`, `java.sql.Statement` und `java.sql.ResultSet` implementieren.

```

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(args[0]));
             FileOutputStream out = new FileOutputStream(args[1])) {

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }
    }
}
```

Programm 8.4 ist die im Allgemeinen schnellere, gepufferte Version von Programm 8.3. Die Klassen `BufferedInputStream` und `BufferedOutputStream` verwenden intern eine Pufferung, um Lese- bzw. Schreibzugriffe zu optimieren.

BufferedInputStream

```
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

```
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.

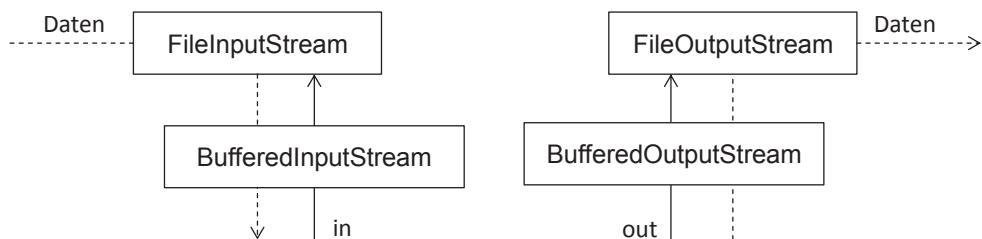


Abbildung 8-1: Verschachtelte Ströme

Eine Instanz vom Typ `BufferedInputStream` wird erzeugt, indem man dem Konstruktor eine Instanz vom Typ `FileInputStream` übergibt.

Das Beispiel zeigt, wie die Klasse `BufferedInputStream` die Funktionalität der Klasse `FileInputStream` mit der Fähigkeit der Pufferung "dekoriert".

Programm 8.4

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy {
    public static void main(String[] args) throws IOException {
        try (BufferedInputStream in = new BufferedInputStream(
            new FileInputStream(args[0]));
            BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream(args[1]))) {
```

```

        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    }
}

```

8.4 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 angegebenen `OutputStream` 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. 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.

Programm 8.5 schreibt und liest Daten im Binärformat.

Programm 8.5

```

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 e) {
        }
    }
}

```

Die Datei *zahlen.data* enthält 416 Bytes: $(2 + 14) + 4 * 100$

8.5 Pushback

Mit Hilfe eines `PushbackInputStream` können bereits gelesene Bytes in den Eingabestrom zurückgestellt und anschließend wieder gelesen werden. Programm 8.6 zeigt, wie ein "Vorauslesen" sinnvoll angewandt werden kann.

`PushbackInputStream`

```

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.

Programm 8.6 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.

Programm 8.6

```
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(args[0]));
            BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream(args[1]))) {

            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(z);
                    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);
}

}
```

8.6 Zeichencodierung

Byteströme können als Zeichenströme auf der Basis eines Zeichensatzes interpretiert werden. Hierbei helfen die beiden Klassen `InputStreamReader` und `OutputStreamWriter`.

InputStreamReader

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String charsetName)
    throws UnsupportedEncodingException
```

erzeugen jeweils einen InputStreamReader, der aus dem angegebenen 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

```
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, String charsetName)
    throws UnsupportedEncodingException
erzeugen jeweils einen OutputStreamWriter, der in den angegebenen
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. Codierungsschemas sind:

US-ASCII, ISO-8859-1, UTF-8, UTF-16.

Programm 8.7 nutzt *UTF-8*, um einen String in eine Datei zu schreiben. Schließlich wird der String aus der Datei wieder zurückgelesen, indem der gleiche Zeichensatz verwendet wird. Dieser String wird dann mit dem Originalstring auf Gleichheit geprüft.

Programm 8.7

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

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"), "UTF-8")) {
            out.write(s1);
        }

        try (InputStreamReader in = new InputStreamReader(new FileInputStream(
                "data.txt"), "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 statt *UTF-8* hier *US-ASCII* verwendet, können die Originalstrings nicht mehr rekonstruiert werden. Der Vergleich liefert `false`.

8.7 Zeichenweise Ein- und Ausgabe

Für Zeichenströme existieren die zu den Byteströmen in Kapitel 8.3 analogen Klassen und Konstruktoren:

FileReader

```
FileReader(File file) throws FileNotFoundException
FileReader(String name) throws FileNotFoundException
```

FileWriter

```
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

```
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

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

Die BufferedWriter-Methode

```
void newLine() throws IOException
```

schreibt einen Zeilentrenner gemäß der Java-Systemeigenschaft `line.separator`.

Programm 8.8 liest Eingaben von der Tastatur. Die Eingabeschleife kann durch Leereingabe (Return-Taste) beendet werden. Die Tastenkombination `Strg + Z` auf Windows bzw. `Strg + D` auf Unix signalisiert ebenfalls das Ende der Eingabe (`readLine` liefert `null`).

Programm 8.8

```
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.length() == 0)
                    break;

                System.out.println(line);
            }
        }
    }
}
```

Programm 8.9 gibt gelesene Zeilen einer Textdatei mit ihrer Zeilennummer aus.

Programm 8.9

```
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(args[0])));
            BufferedWriter out = new BufferedWriter(new FileWriter(args[1]))) {

            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 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 Zeilentrener gemäß der Java-Systemeigenschaft `line.separator` geschrieben wird.

void println()

gibt den Zeilentrener aus.

Formatierte Ausgabe mit printf

`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 nutzen hier eine vereinfachte Format-Syntax zur Ausgabe der einzelnen Argumente, 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

Für weitere Einzelheiten sei auf die Dokumentation zu Java SE verwiesen.

PrintStream

Diese Methoden existieren auch für den Bytestrom `PrintStream`. Ebenso existieren entsprechend die ersten vier Konstruktoren des `PrintWriter` auch für `PrintStream`.

Das folgende Programm zeigt den Nutzen von `printf`.

Programm 8.10

```
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

	51,97

8.8 Gefilterte Datenströme

FilterReader

Die abstrakte Klasse `FilterReader` besitzt

- die Variable `protected Reader in` und den
- Konstruktor `protected FilterReader(Reader in)`.

FilterWriter

Die abstrakte Klasse `FilterWriter` besitzt

- die Variable `protected Writer out` und den
- Konstruktor `protected FilterWriter(Writer out)`.

FilterInputStream

Die Klasse `FilterInputStream` besitzt

- die Variable `protected InputStream in` und den
- Konstruktor `protected FilterInputStream(InputStream in)`.

FilterOutputStream

Die Klasse `FilterOutputStream` besitzt

- die Variable `protected OutputStream out` und den
- Konstruktor `public FilterOutputStream(OutputStream out)`.

Alle Lese- bzw. Schreibaufrufe werden an den Strom `in` bzw. `out` weitergeleitet. Eigene Subklassen dieser Klassen können genutzt 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.

Programm 8.11 nutzt eine Subklasse der Klasse `FilterWriter`, um Umlaute und 'ß' in die Zeichen ae, oe, ue bzw. ss zu wandeln.

Programm 8.11

```
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");
                break;
            case 'ö':
                super.write("oe");
                break;
            case 'ü':
                super.write("ue");
                break;
            case 'Ä':
                super.write("Ae");
                break;
            case 'Ö':
                super.write("Oe");
                break;
            case 'Ü':
                super.write("Ue");
                break;
            case 'ß':
                super.write("ss");
                break;
            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());
    }
}
```

```
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(args[0]));
             UmlautWriter out = new UmlautWriter(new BufferedWriter(
                     new FileWriter(args[1])))) {
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }
    }
}
```

Die an den `UmlautWriter` gerichteten Schreibaufrufe werden über einen `BufferedWriter` an einen `FileWriter` weitergeleitet.

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*).

8.9 Serialisierung von Objekten

Die beiden folgenden Programme schreiben bzw. lesen komplexe 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

Konstruktoren:

`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` deklariert sind, werden geschrieben. Dies gilt auch für ggf. in diesem Objekt referenzierte andere Objekte.

Die `ObjectInputStream`-Methode

```
Object readObject() throws IOException, ClassNotFoundException
```

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 Markierungs-Interface `java.io.Serializable` implementieren. Dieses Interface besitzt keine Methoden. 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.

`serialVersionUID`

Die Klasse, deren Objekt serialisiert wurde, muss kompatibel sein zu der Klasse, für die das Objekt wieder deserialisiert wird. Würde man nach der Serialisierung beispielweise das Attribut `adresse` der Klasse `Kunde` im folgenden Programm entfernen, so könnte das ursprüngliche Objekt nicht mehr rekonstruiert werden. Das Hinzufügen eines weiteren Attributs würde hingegen keine Probleme bereiten.

Bei der Serialisierung erzeugt das Laufzeitsystem eine Versionsnummer auf Basis verschiedener Aspekte derserialisierbaren 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 z. B. später Erweiterungen der Klasse stattfinden, die keinen Einfluss auf die Rekonstrierbarkeit eines Objekts haben (wie z. B. das oben geschilderte Hinzufügen eines Attributs). In diesem Fall lässt man dann die alte Versionsnummer bestehen.

Die Klasse `Kunde` implementiert das Interface `Serializable`. Ihre Superklasse `Object` ist nicht serialisierbar, hat aber einen parameterlosen `public`-Konstruktor. Damit erfüllt `Kunde` die hier gemachten Voraussetzungen. Die Klasse `ArrayList` implementiert ebenfalls `Serializable`.

Programm 8.12

```
public class Kunde implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private String adresse;

    public Kunde(String name, String adresse) {
        this.name = name;
        this.adresse = 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;
    }
}

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

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");

        ArrayList<Kunde> kunden = new ArrayList<Kunde>();
        kunden.add(k1);
        kunden.add(k2);
        kunden.add(k2);

        try (ObjectOutputStream out = new ObjectOutputStream(
                new FileOutputStream("kunden.ser"))) {
            out.writeObject(kunden);
            out.flush();
        }
    }
}
```

```

import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.ArrayList;

public class DeserializeTest {
    public static void main(String[] args) throws Exception {
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(
                "kunden.ser"))) {
            @SuppressWarnings("unchecked")
            ArrayList<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 `ArrayList`-Objekt `kunden` speichert das `Kunde`-Objekt `k1` und zweimal dasselbe `Kunde`-Objekt `k2`. Das Ergebnis der Deserialisierung ist ein *Objektgraph*, der zum Eingabegraph äquivalent ist, d. h. auch im Ergebnis referenzieren die Listen-Einträge mit der Nummer 1 und 2 dasselbe Objekt. Ein Objekt wird nur einmal serialisiert, auch wenn es mehrfach referenziert wird.

Sonderbehandlung bei Serialisierung

Programm 8.13

Die Klasse `Document` 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-Serialisierung und -Deserialisierung kann mittels der

ObjectOutputStream-Methode `defaultWriteObject` bzw.
ObjectInputStream-Methode `defaultReadObject`

genutzt werden.

Das Beispiel zeigt, dass zunächst die Standard-Serialisierung aufgerufen und anschließend der Klassenvariablenwert geschrieben bzw. gelesen wird.

```
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 int id;
    private String content;

    public Document() {
        id = ++nextId;
    }

    public Document(String content) {
        id = ++nextId;
        this.content = content;
    }

    public int getId() {
        return id;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    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();
    }
}

import java.io.FileOutputStream;
import java.io.IOException;
```

```

import java.io.ObjectOutputStream;

public class SerializeTest {
    public static void main(String[] args) throws IOException,
        ClassNotFoundException {
        try (ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("documents.ser"))) {
            out.writeObject(new Document("AAA"));
            out.writeObject(new Document("BBB"));
            out.flush();
        }
    }
}

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"));
    }
}

```

8.10 Wahlfreier Dateizugriff

Objekte der Klasse `RandomAccessFile` ermöglichen den Zugriff auf so genannte Random-Access-Dateien. Eine *Random-Access-Datei* ist eine Datei mit wahlfreiem Zugriff, 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

`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 Zugriffsroutinen lösen im Fehlerfall Ausnahmen vom Typ `IOException` aus.

```
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)
```

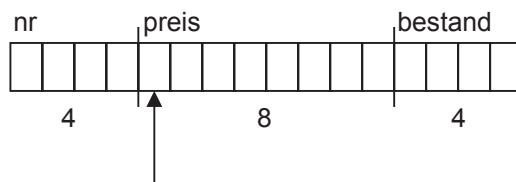
setzt die Größe der Datei auf newLength Bytes. Ist die aktuelle Größe der Datei größer als newLength, so wird die Datei abgeschnitten, ist sie kleiner als newLength, so wird sie mit von der Implementierung gewählten Bytewerten auf die neue Länge vergrößert.

```
void close()
```

schließt die Datei.

Die Klasse `RandomAccessFile` enthält die gleichen in Kapitel 8.4 spezifizierten 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.



Nachdem die Artikelnummer in `getArtikel` gefunden wurde, steht der Dateizeiger hier.

Abbildung 8-2: Datensatzstruktur

Programm 8.14

```
public class Artikel {  
    private int nr;  
    private double preis;  
    private int bestand;  
  
    public Artikel(int nr, double preis, int bestand) {  
        this.nr = nr;  
        this.preis = preis;  
        this.bestand = bestand;  
    }  
  
    public int getNr() {  
        return nr;  
    }  
  
    public double getPreis() {  
        return preis;  
    }  
  
    public int getBestand() {  
        return bestand;  
    }  
  
    public String toString() {  
        return "Artikel [nr=" + nr + ", preis=" + preis + ", bestand=" +  
               + bestand + "]";  
    }  
}  
  
import java.io.EOFException;  
import java.io.IOException;  
import java.io.RandomAccessFile;  
  
public class ArtikelManager {  
    private RandomAccessFile datei;  
  
    public ArtikelManager(String name) throws IOException {  
        datei = new RandomAccessFile(name, "rw");  
    }  
  
    public void close() throws IOException {  
        datei.close();  
    }  
  
    public Artikel getArtikel(int nr) throws IOException {  
        boolean found = false;  
        int artNr, bestand;  
        double preis;  
  
        // Dateizeiger auf den Anfang setzen  
        datei.seek(0L);  
  
        try {  
            while (!found) {  
                artNr = datei.readInt();  
                if (artNr == nr) {  
                    found = true;  
                } else {  
                    // Preis (8 Bytes) und Bestand (4 Bytes) überspringen  
                }  
            }  
        } catch (IOException e) {  
            System.out.println("Fehler beim Lesen des Artikels mit Nummer " + nr);  
        }  
    }  
}
```

```
        datei.skipBytes(12);
    }
}
} catch (EOFException e) {
    return null;
}

preis = datei.readDouble();
bestand = datei.readInt();
return new Artikel(nr, preis, bestand);
}

public void list() throws IOException {
    // Dateizeiger auf den Anfang setzen
    datei.seek(0L);

    int artNr, bestand;
    double preis;

    try {
        while (true) {
            artNr = datei.readInt();
            preis = datei.readDouble();
            bestand = datei.readInt();
            System.out.printf("%4d %8.2f %8d%n", artNr, preis, bestand);
        }
    } catch (EOFException e) {
    }
}

public boolean addArtikel(Artikel a) throws IOException {
    if (getArtikel(a.getNr()) == null) {
        // Dateizeiger auf das Ende setzen
        datei.seek(datei.length());

        datei.writeInt(a.getNr());
        datei.writeDouble(a.getPreis());
        datei.writeInt(a.getBestand());
        return true;
    } else
        return false;
}

public boolean addBestand(int nr, int zugang) throws IOException {
    Artikel a = getArtikel(nr);

    if (a == null)
        return false;
    else {
        // Dateizeiger steht hinter Bestand,
        // Dateizeiger nun auf das erste Byte von Bestand setzen
        datei.seek(datei.getFilePointer() - 4L);
        datei.writeInt(a.getBestand() + 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));

        System.out.println(manager.getArtikel(5011));
        manager.addBestand(5011, -100);
        manager.list();

        manager.close();
    }
}

```

Ausgabe des Programms:

```

Artikel [nr=5011, preis=100.0, bestand=450]
4711 140,99 1000
5011 100,00 350
1112 47,50 1000

```

8.11 Datenkomprimierung

Programm 8.15

`java.util.zip.GZIPOutputStream` und `java.util.zip.GZIPInputStream` komprimieren bzw. expandieren einen Bytestrom. Dabei wird das Kompressionsverfahren *gzip* (GNU zip) angewandt.

`GZIPOutputStream` ist ein `FilterOutputStream` und `GZIPInputStream` ist ein `FilterInputStream`.

Das Programm `RandomOutput` erzeugt Testdaten und kann wie folgt aufgerufen werden:

```
java -cp bin RandomOutput 1000000 data
```

```

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Random;

public class RandomOutput {
    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(args[0]);
        try (PrintWriter writer = new PrintWriter(new FileWriter(args[1]))) {
            Random r = new Random();
            for (int i = 0; i < n; i++) {
                writer.print(r.nextInt(100) + " ");
            }
        }
    }
}

```

Komprimierung:

```
java -cp bin Gzip data data.gz
```

```
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(args[0]));
            OutputStream out = new GZIPOutputStream(new FileOutputStream(
                args[1]))) {

            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = in.read(buffer)) != -1) {
                out.write(buffer, 0, c);
            }

            out.flush();
        }
    }
}
```

Expandierung:

```
java -cp bin Gunzip data.gz data2
```

```
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(args[0])));
            OutputStream out = new FileOutputStream(args[1])) {

            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = in.read(buffer)) != -1) {
                out.write(buffer, 0, c);
            }

            out.flush();
        }
    }
}
```

In diesem Beispiel beträgt die Kompressionsrate ca. 36 %.

Besonders interessant bei Übertragung von größerem Datenvolumen über das Netz: Komprimieren auf Senderseite, Expandieren auf Empfängerseite.

Programm 8.16

Das ZIP-Dateiformat ist ein Format für komprimierte Dateien, wobei mehrere Dateien und Verzeichnisse zusammengefasst werden können.

`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 Programm `Zip` erstellt eine Zip-Datei, wobei neben dem Ausgabedateinamen die zu komprimierenden

Dateien und Verzeichnisse als Aufrufparameter mitgegeben werden können.

```
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]));
            }
        }
    }

    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 Hilfsmethode `zip` löst die Unterverzeichnisse bis auf Dateiebene rekursiv auf.

Aufrufbeispiel:

```
java -cp bin Zip test.zip src bin
```

`java.util.zip.ZipFile` wird genutzt, um die Einträge einer ZIP-Datei zu lesen.

```
import java.io.IOException;
import java.util.Enumeration;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;

public class ZipInfo {
    public static void main(String[] args) throws IOException {
        try (ZipFile zipFile = new ZipFile(args[0])) {
            Enumeration<? extends ZipEntry> enumeration = zipFile.entries();

            while (enumeration.hasMoreElements()) {
                ZipEntry entry = enumeration.nextElement();

                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 bin ZipInfo test.zip
```

Ausgabe des Programms:

```
src\Unzip.java
  size: 995      compressed size: 420      2014-05-28 18:33:16
src\Zip.java
  size: 1071      compressed size: 439      2014-05-28 18:33:16
src\ZipInfo.java
  size: 676      compressed size: 324      2014-05-28 18:33:16
bin\Unzip.class
  size: 2126      compressed size: 1191     2014-05-28 18:33:16
bin\Zip.class
```

```
size: 2182 compressed size: 1248 2014-05-28 18:33:16
bin\ZipInfo.class
size: 1710 compressed size: 964 2014-05-28 18:33:16
```

`java.util.zip.ZipInputStream` ist ein `FilterInputStream` zum Lesen von ZIP-Dateien.

Das Programm `Unzip` entpackt die ZIP-Datei (1. Aufrufparameter) in einem vorgegebenen Verzeichnis (2. Aufrufparameter). Evtl. Unterverzeichnisse werden erstellt.

```
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();

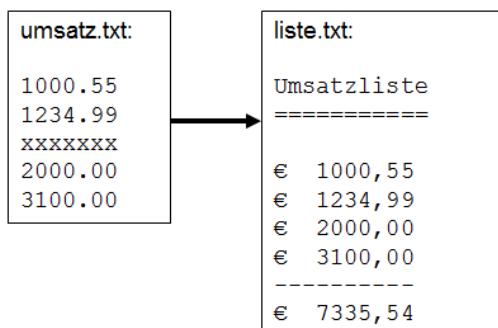
        try (OutputStream out = new FileOutputStream(file)) {
            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = in.read(buffer)) != -1) {
                out.write(buffer, 0, c);
            }
        }
    }
}
```

Aufrufbeispiel:

```
java -cp bin Unzip test.zip tmp
```

8.12 Aufgaben

1. Schreiben Sie ein Programm, das rekursiv die Unterverzeichnisse und die Dateien eines vorgegebenen Verzeichnisses auflistet.
2. Eine Textdatei soll am Bildschirm ausgegeben werden. Ist eine Bildschirmseite voll, soll das Programm anhalten, bis die Return-Taste gedrückt wird.
3. Schreiben Sie ein Programm, das die Anzahl Zeichen und Zeilenwechsel einer Textdatei ermittelt. Die Textdatei kann über Eingabe-Umlenkung mit `System.in.read()` eingelesen werden.
4. Schreiben Sie ein Programm, das eine Datei nach einem vorgegebenen Wort durchsucht und alle Zeilen, in denen das Wort gefunden wird, mit der Zeilennummer davor ausgibt.
5. Wie kann eine Instanz der Klasse `PrintWriter` genutzt werden, um Daten am Bildschirm auszugeben?
6. Fügen Sie in die Klasse `Artikel` aus Programm 8.14 eine Methode hinzu, die die Werte der Instanzvariablen dieses Objekts (`this`) in einen `DataOutputStream` schreibt. Erstellen Sie dann einen Konstruktor, der die Instanzvariablenwerte für das neue Objekt aus einem `DataInputStream` liest.
7. Schreiben Sie ein Programm, das die mittels Programm 8.6 komprimierte Datei dekomprimiert.
8. Wie kann Programm 8.6 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.
9. Schreiben Sie ein Programm, das die folgende Eingabe in eine Ausgabeliste transformiert:



Fehlerhafte Zahlen sollen beim Einlesen ignoriert werden.

10. Schreiben Sie ein Programm, das die Zeichen einer Textdatei beim Lesen über einen Filter sofort in Großbuchstaben umwandelt.

11. 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	...

12. 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:

```
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
...
69 6e 74 6c 6e 28 29 3b 0d 0a 09 09 09 7d 0d 0a intln();....}..
09 09 7d 0d 0a 09 7d 0d 0a 7d 0d 0a ..}....}...}..
```

Tipp: Nutzen Sie die Integer-Methode `toHexString`.

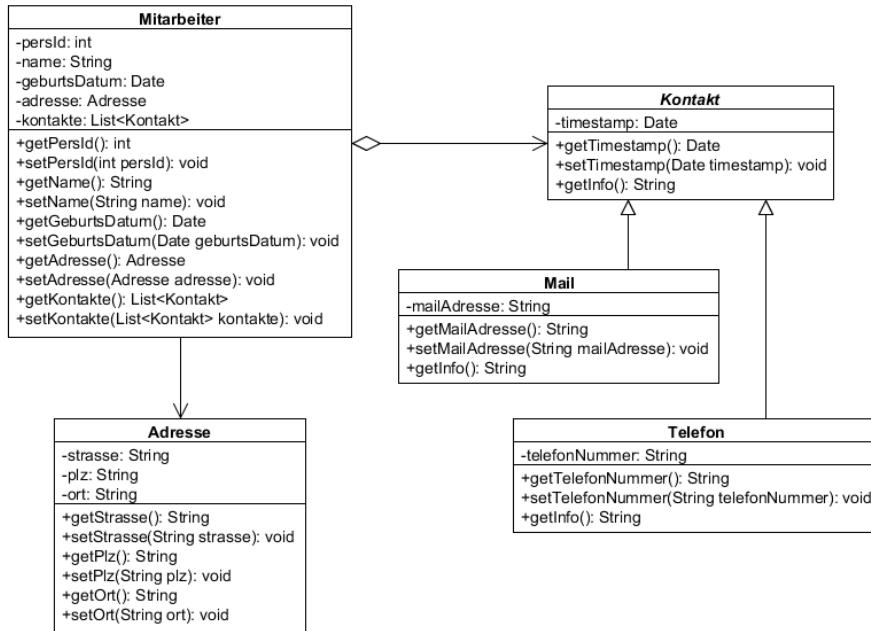
13. Dateien können byteweise und blockweise (`int read(byte[] b)`) gelesen und geschrieben (`void write(byte[] b)`) werden. Schreiben Sie zwei Programme, die eine große Datei byteweise bzw. blockweise kopieren und ermitteln Sie dabei die Laufzeit in Millisekunden sowie den Durchsatz in Bytes/Sek. Testen Sie auch verschiedene Arraygrößen.
14. Fügen Sie in die Klasse `Artikel` aus Programm 8.14 die beiden folgenden Methoden ein:

```
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.

15. Entwickeln Sie gemäß Vorgabe die Klassen des nachfolgenden Klassendiagramms. Ein Mitarbeiter hat eine Adresse und ggf. mehrere Kontaktinformationen (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.



16. 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:

- Nutzung der Ausnahme `EOFException`, die das Ende des Streams signalisiert.
- Markierung des Endes durch Einfügen des Wertes null nach dem letzten Objekt bei der Serialisierung und beim Einlesen Erkennen des Endes am Wert null.
- Schreiben einer Zahl, die die Anzahl der folgenden Objekte angibt, als ersten Wert bei der Serialisierung und Durchlaufen der Leseschleife bei der Deserialisierung so oft, wie der Zählerwert angibt.

Testen Sie diese verschiedenen Lösungsalternativen.

17. 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`.

18. Schreiben Sie ein Programm, das bei jedem Aufruf einen als Aufrufparameter mitgegebenen double-Wert zusammen mit der Systemzeit in Millisekunden (long-Wert) in eine RandomAccess-Datei einträgt. Ein weiteres Programm soll die n letzten Einträge am Bildschirm anzeigen. Die Millisekundenangabe soll für die Ausgabe in Datum und Uhrzeit umgewandelt werden.
19. Die Klasse `java.bean.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.bean.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.

9 Threads

Moderne Betriebssysteme können mehrere Programme quasi gleichzeitig (*Multi-tasking*) oder tatsächlich gleichzeitig (bei 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* (Ausführungsstrang, Handlungsstrang) ist ein einzelner in sich geschlossener Steuerfluss innerhalb eines Prozesses. Jeder Prozess besitzt einen Hauptthread, mit dem das Programm startet. Dieser führt die `main`-Methode aus. Mehrere neue Threads können nun vom Programm selbst gestartet werden (*Multi-threading*). Diese Threads laufen dann alle parallel ab, besitzen jeweils einen eigenen Zustand mit Befehlszähler, Stack usw., arbeiten aber im Gegensatz zu Prozessen auf demselben Speicherbereich im Hauptspeicher.

In einem Einprozessorsystem kann die Gleichzeitigkeit dadurch simuliert werden, dass der Prozessor alle Prozesse und Threads reihum in schneller Folge schrittweise abarbeitet (*Time Slicing*).

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 (z. B. Datenbankabfragen) verschiedener Clients gleichzeitig bearbeiten können. Zur parallelen Abarbeitung dieser Anfragen können Threads vom Serverprogramm gestartet werden.

Lernziele

In diesem Kapitel lernen Sie

- wie Sie in einem Programm parallel laufende Threads starten können,
- welche Zustände Threads im Laufe ihres Lebens einnehmen können,
- welche Probleme durch den gleichzeitigen Zugriff mehrerer Threads auf dieselben Daten auftreten können und wie solche Probleme zu vermeiden sind,
- wie Threads koordiniert an einer gemeinsamen Aufgabe arbeiten können.

9.1 Threads erzeugen und beenden

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 im obigen Sinne auszuführenden Code. Die Standardimplementierung von `run` in der Klasse `Thread` tut gar nichts.

Threads erzeugen

Um einen Thread zu erzeugen, gibt es grundsätzlich zwei Möglichkeiten:

- Die Klasse, die die Threadausführung definiert, ist von `Thread` abgeleitet oder
- die Klasse implementiert das Interface `Runnable` direkt und mit einem Objekt dieser Klasse als Konstruktorparameter wird ein `Thread`-Objekt erzeugt.

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 die Ausführung des Threads, indem sie die Methode `run` von der JVM aufrufen lässt. Der Aufrufer der Methode `start` kann mit seiner Ausführung sofort fortfahren, die nun parallel zur Ausführung des Threads läuft. `start` darf nur einmal für jeden Thread aufgerufen werden.

Konstruktoren der Klasse `Thread` sind:

`Thread()`
`Thread(String name)`

`name` ersetzt den Standardnamen eines 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 eines Threads durch einen eigenen Namen.

Programm 9.1 zeigt beide Möglichkeiten, Threads zu erzeugen. Die `Thread`-Methode `sleep` legt hier den aktuellen Thread für eine Sekunde schlafen. Die Methoden `sleep`, `getName` und `currentThread` sind weiter unten erklärt.

Programm 9.1

```
public class Test1 extends Thread {
    private int count;

    public void run() {
        String name = getName(); // Name des Threads
```

```
for (int i = 0; i < 3; i++) {
    try {
        Thread.sleep(1000); // 1 Sek. schlafen
    } catch (InterruptedException e) {
    }

    System.out.println(name + ": " + ++count);
}

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.");
}
}

public class Test2 implements Runnable {
    private int count;

    public void run() {
        String name = Thread.currentThread().getName();

        for (int i = 0; i < 3; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }

            System.out.println(name + ": " + ++count);
        }

        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:

```
Habe zwei Threads gestartet.
Thread-0: 1
Thread-1: 1
Thread-1: 2
Thread-0: 2
Thread-1: 3
Thread-0: 3
```

```
Thread-1: Ich bin fertig!
Thread-0: Ich bin fertig!
```

Abbildung 9-1 fasst die beiden Möglichkeiten zur Erzeugung eines Threads zusammen.

Ein Programm endet, 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. Programm 9.2 demonstriert diesen Sachverhalt.

Die Thread-Methode

```
final void setDaemon(boolean on)
```

markiert den Thread als Daemon-Thread, falls on den Wert true hat.

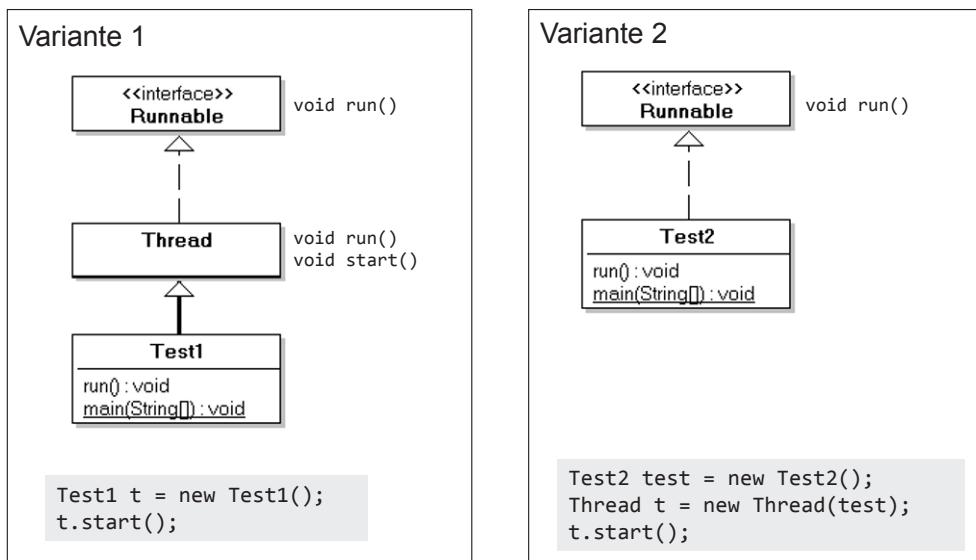


Abbildung 9-1: Zwei Varianten zur Erzeugung eines Threads

Programm 9.2

```
public class ProgramEndTest extends Thread {
    public void run() {
        System.out.println("Beginn run");

        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(1000);
            }
        }
    }
}
```

```
        System.out.print(".");
    } catch (InterruptedException e) {
    }
}

System.out.println("\nEnde run");
}

public static void main(String[] args) {
    ProgramEndTest p = new ProgramEndTest();
    p.setDaemon(true);
    p.start();
    System.out.println("Ende main");
}
}
```

Ausgabe bei normalem Thread (`on = false`):

```
Ende main
Beginn run
.....
Ende run
```

Ausgabe bei Daemon-Thread (`on = true`):

```
Ende main
Beginn run
```

Weitere 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 durch den Aufruf von `sleep`, `join` oder `wait`¹, so wird der Status "unterbrochen" gelöscht und eine `InterruptedException` ausgelöst.

`boolean isInterrupted()`

liefert `true`, falls der Thread den Status "unterbrochen" hat, sonst `false`.

¹ Siehe Kapitel 9.3.

```
static boolean interrupted()
    prüft, ob der aktuelle Thread unterbrochen wurde und löscht den Status
    "unterbrochen" im Falle von true.

static void sleep(long millis) throws InterruptedException
    hält die Ausführung des aktuellen Threads für mindestens millis Millisekunden an. 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 werden diejenigen mit höherer Priorität vor denen mit niedrigerer Priorität ausgeführt.

```
void setPriority(int p)
    setzt die Priorität des Threads.
    p muss zwischen den Werten Thread.MIN_PRIORITY = 1 und Thread.MAX_PRIORITY = 10 liegen. Thread.NORM_PRIORITY = 5 ist der Normalwert.

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.
```

Threads sollten so entwickelt werden, dass sie auf jedem System lauffähig sind.

Ob und wie *Time Slicing* unterstützt wird, hängt insbesondere vom Betriebssystem ab. Kann man sich nicht auf die Zeitscheibenzuteilung verlassen, kann ein Thread in der run-Methode mit yield jeweils für kurze Zeit deaktiviert werden, sodass andere Threads zum Zuge kommen.

Threads beenden

Programm 9.3 zeigt, wie ein Thread, der im Sekundenrhythmus die aktuelle Uhrzeit anzeigt, per interrupt beendet werden kann.

Programm 9.3

```
import java.io.IOException;

public class Zeitanzeige 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 {
        Zeitanzeige zeit = new Zeitanzeige();
        Thread t = new Thread(zeit);
        t.start();

        System.in.read(); // blockiert bis RETURN
        t.interrupt();
    }
}
```

Programm 9.4 zeigt eine weitere Möglichkeit zur Beendigung der while-Schleife innerhalb der run-Methode.

Programm 9.4

```
public class Zeitanzeige implements Runnable {
    private volatile Thread t;

    public void start() {
        t = new Thread(this);
        t.start();
    }

    public void stop() {
        t = null;
    }

    public void run() {
        while (Thread.currentThread() == t) {
            System.out.println(new java.util.Date());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) throws java.io.IOException {
        Zeitanzeige zeit = new Zeitanzeige();
        zeit.start();
    }
}
```

```
        System.in.read();  
        zeit.stop();  
    }  
}
```

Die Methode `start` erzeugt einen Thread `t` und startet diesen. Die Methode `stop` setzt die Referenz `t` auf `null`.

Die `while`-Schleife läuft so lange, wie der aktuelle Thread mit `t` übereinstimmt. Ist `t` gleich `null` (oder referenziert `t` bereits ein neues Thread-Objekt), so ist die Bedingung nicht mehr erfüllt und die Schleife wird verlassen.

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.

Der `main`-Thread in Programm 9.4 verändert über die Methode `stop` die Instanzvariable `t`. Der gestartete Thread liest `t`, 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.

Mit der Thread-Methode `join` kann ein Thread auf das Ende eines anderen Threads warten.

Programm 9.5

```
public class JoinTest extends Thread {  
    public void run() {  
        System.out.println(getName() + ": Es geht los!");  
  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException e) {  
        }  
  
        System.out.println(getName() + ": Erledigt!");  
    }  
}
```

```

public static void main(String[] args) {
    JoinTest t = new JoinTest();
    t.start();

    try {
        t.join();
    } catch (InterruptedException e) {
    }

    System.out.println("Endlich!");
}
}

```

Abbildung 9-2 zeigt die verschiedenen Zustände, die ein Thread einnehmen kann.

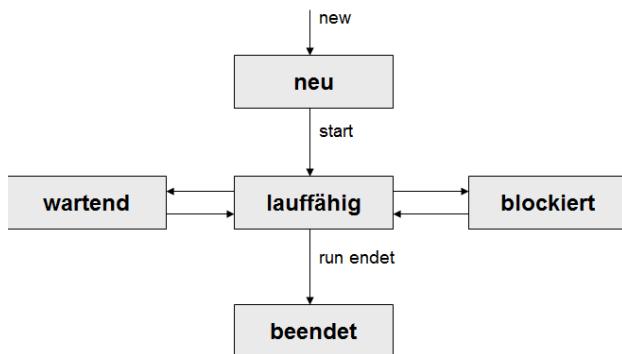


Abbildung 9-2: Thread-Zustände

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`² gelangen. Ein Thread ist *blockiert*, wenn er aufgrund einer Sperre auf den Eintritt in einen synchronisierten Block (Methode) warten muss.³

9.2 Synchronisation

Beim Multithreading können Probleme durch den gleichzeitigen Zugriff auf gemeinsame Objekte und Variablen auftreten. 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 undefinierte Ergebnisse zu

² Siehe Kapitel 9.3.

³ Siehe Kapitel 9.2.

vermeiden, müssen die Zugriffe bei Änderung gemeinsamer Daten synchronisiert werden.

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 kompletten Methode oder
- Schützen eines Blocks innerhalb einer Methode.

synchronized für eine Methode

Wir beschäftigen uns zunächst mit der ersten Möglichkeit.

Durch den Modifizierer `synchronized` kann die Ausführung von Instanzmethoden, die für *dasselbe* Objekt aufgerufen werden, synchronisiert werden, sodass *nur eine* von ihnen zu einer Zeit von einem Thread ausgeführt werden kann.

`synchronized` hat für das Überladen (Overloading) oder Überschreiben (Overriding) von Methoden keine Bedeutung.

Wird eine solche Methode aufgerufen, wird zuerst das Objekt, für das sie aufgerufen wurde, gesperrt. Dann wird die Methode ausgeführt und am Ende die Sperre aufgehoben. Ein anderer Thread, der eine `synchronized`-Methode (oder einen `synchronized`-Block) für *dasselbe* Objekt aufruft, wird so lange blockiert, bis die Sperre aufgehoben wurde.

Für jedes Objekt wird eine *Warteliste* geführt, in der die blockierten Threads eingetragen sind.

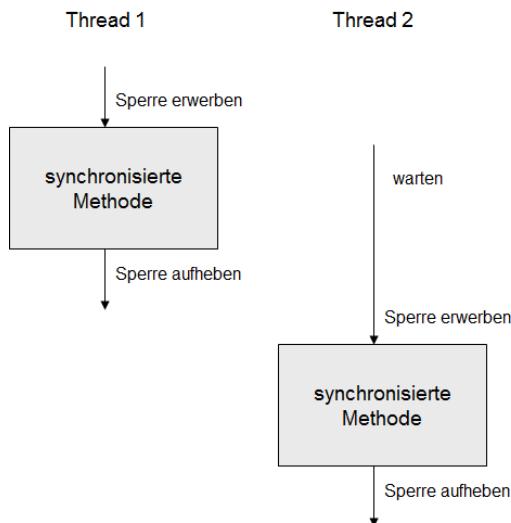


Abbildung 9-3: Sperrkonzept

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.

Synchronisierte Methoden können andere synchronisierten Methoden *dasselben* Objekts benutzen.

Sperrobjekt bei synchronisierten Klassenmethoden (`static`-Methoden) ist das `Class`-Objekt der jeweiligen Klasse, sodass zu jedem Zeitpunkt nur immer eine synchronisierte Klassenmethode dieser Klasse ausgeführt werden kann.

Programm 9.6 demonstriert diesen Sachverhalt. Die Klasse `Work` enthält die synchronisierte Instanzmethode `doWorkA` und die synchronisierte Klassenmethode `doWorkB`. Die `main`-Methode startet zunächst zwei Threads unmittelbar hintereinander, die `workA` für dasselbe Objekt aufrufen. Nach Beendigung der beiden Threads wird nochmals ein Thread wie vorher gestartet, anschließend wird die Klassenmethode `doWorkB` aufgerufen.

Programm 9.6

```
public class Work {
    public synchronized void doWorkA(String name) {
        System.out.println(name + ": Beginn A");

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }

        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 e) {
        }

        System.out.println(name + ": Ende B");
    }
}

public class SyncTest extends Thread {
    private Work w;

    public SyncTest(Work w) {
        this.w = w;
    }
}
```

```

public void run() {
    w.doWorkA(Thread.currentThread().getName());
}

public static void main(String[] args) {
    Work w = new Work();
    SyncTest t1 = new SyncTest(w);
    SyncTest t2 = new SyncTest(w);
    SyncTest t3 = new SyncTest(w);

    t1.start();
    t2.start();

    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
    }

    t3.start();
    Work.doWorkB(Thread.currentThread().getName());
}
}

```

Ausgabe des Programms:

```

Thread-0: Beginn A
Thread-0: Ende A
Thread-1: Beginn A
Thread-1: Ende A
main: Beginn B
Thread-2: Beginn A
main: Ende B
Thread-2: Ende A

```

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 Ausführungszeit der gesamten Methode kürzere Zeitdauer beansprucht.

- Als Sperrobjekte können neben `this` auch andere Objekte gewählt werden.

Eine `synchronized`-Methode

```
public void synchronized xyz() { ... }
```

ist äquivalent zu:

```
public void xyz() {
    synchronized(this) { ... }
}
```

Deadlock

Programm 9.7 zeigt eine *Deadlock*-Situation: Zwei Threads warten auf die Aufhebung der Sperre durch den jeweils anderen Thread.

Programm 9.7

```
public class DeadlockTest {
    private Object a = new Object();
    private 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 e) {
            }

            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(new Runnable() {
        public void run() {
            test.f("Thread 1");
        }
    });

    Thread t2 = new Thread(new Runnable() {
        public void run() {
            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 dieses 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.

Grundregel: synchronisierte Zugriffe

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.

Programm 9.8

Programm 9.8 zeigt anhand einer einfachen Summenberechnung, wie eine Anwendung parallelisiert werden kann, um die Laufzeit des Programm zu reduzieren. Allerdings kommt es darauf an, abhängig von der Aufgabenstellung richtig im Sinne der Anforderung zu synchronisieren.⁴

In `SimpleSum` berechnet der `main`-Thread alleine die Summe.

Ausgabe:

```
Sum: 499999999500000000  
Time: 1436
```

```
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 < 1000000000; n++) {  
            sum += n;  
        }  
  
        long time = System.currentTimeMillis() - startTime;  
  
        System.out.println("Sum: " + sum);  
        System.out.println("Time: " + time);  
    }  
}
```

In `ThreadedSum` teilen sich vier Threads die Aufgabe der Summenberechnung. Sie addieren jeweils Zahlen aus den Bereichen 0 ... 249999999, 250000000 ... 499999999, 500000000 ... 749999999 und 750000000 ... 999999999.

```
public class ThreadedSum {  
    private long sum;  
    private Thread[] threads = new Thread[4];  
  
    public static void main(String[] args) {  
        new ThreadedSum().sum();  
    }  
  
    public void sum() {  
        long startTime = System.currentTimeMillis();  
    }
```

⁴ 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.

```

for (int i = 0; i < 4; i++) {
    final long start = i * (1000000000 / 4);
    final long end = (i + 1) * (1000000000 / 4);

    threads[i] = new Thread() {
        public void run() {
            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 e) {
    }
}
long time = System.currentTimeMillis() - startTime;

System.out.println("Sum: " + sum);
System.out.println("Time: " + time);
}
}

```

Ausgabe:

Sum: 156248923502032952
 Time: 562

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 Summenbildung stattfinden.

In `SynchronizedSum` erfolgt jeder Berechnungsschritt synchronisiert. Das Ergebnis ist korrekt, die Laufzeit hat aber erheblich zugenommen.

```

public class SynchronizedSum {
    private long sum;
    private 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 * (1000000000 / 4);

```

```
final long end = (i + 1) * (1000000000 / 4);

threads[i] = new Thread() {
    public void run() {
        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 e) {
    }
}

long time = System.currentTimeMillis() - startTime;

System.out.println("Sum: " + sum);
System.out.println("Time: " + time);
}
}
```

Ausgabe:

```
Sum: 499999999500000000
Time: 100948
```

In `ConcurrentSum` nutzt jeder Thread eine eigene Ergebnisvariable. Die berechnete Teilsumme wird jeweils am Ende der Threadausführung synchronisiert aufaddiert.

```
public class ConcurrentSum {
    private long sum;
    private 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 * (1000000000 / 4);
            final long end = (i + 1) * (1000000000 / 4);

            threads[i] = new Thread() {
                public void run() {
                    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 e) {
    }
}

long time = System.currentTimeMillis() - startTime;

System.out.println("Sum: " + sum);
System.out.println("Time: " + time);
}
```

Ausgabe:

Sum: 499999999500000000
Time: 421

Die Laufzeit ist erheblich geringer als im ersten Beispiel (`SimpleSum`) und das Ergebnis stimmt.

9.3 Kommunikation zwischen Threads

Programm 9.10 demonstriert ein so genanntes *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 zwischen- gespeichert. 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`. Sie 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.

Programm 9.9

Bevor das oben erwähnte Producer/Consumer-Problem behandelt wird, soll mit dem Programm 9.9 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.

```
import java.io.IOException;

public class WaitNotifyDemo {
    private Object lock = new Object();
    private boolean wait = true;

    public WaitNotifyDemo() {
        new Waiter().start();
        new Notifier().start();
    }

    class Waiter extends Thread {
        public void run() {
            synchronized (lock) {
                while (wait) {
                    System.out.println("Waiter: wait = " + wait);
                    System.out.println("Waiter wartet");

                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                    }
                }
            }
        }
    }

    class Notifier extends Thread {
        public void run() {
            synchronized (lock) {
                wait = false;
                lock.notify();
            }
        }
    }
}
```

```

        System.out.println("Waiter: wait = " + wait);
    }
}

class Notifier extends Thread {
    public void run() {
        System.out.println("Notifier: Weiter mit RETURN");

        try {
            System.in.read();
        } catch (IOException e) {
        }

        synchronized (lock) {
            wait = false;
            lock.notify();
            System.out.println("Notifier: notify aufgerufen");
        }
    }
}

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

```

Ausgabe des Programms:

Waiter: wait = true
 Waiter wartet
 Notifier: Weiter mit RETURN

Notifier: notify aufgerufen
 Waiter: wait = false

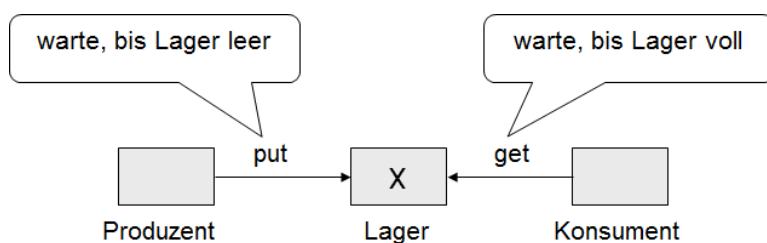


Abbildung 9-4: Kommunikation zwischen Threads

Die Methoden `get` und `put` der Klasse `Lager` im Programm 9.10 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.

Programm 9.10

```
import java.util.Random;

public class Produzent extends Thread {
    private static final int MAX = 10;
    private Lager lager;

    public Produzent(Lager lager) {
        this.lager = lager;
    }

    public void run() {
        try {
            Random random = new Random();
            for (int i = 0; i < MAX; i++) {
                int value = random.nextInt(100);
                System.out.println((i + 1) + ". Wert " + value + " wird produziert");
                Thread.sleep(1000 + random.nextInt(4000));
                lager.put(value);
                System.out.println((i + 1) + ". Wert " + value + " auf Lager");
            }

            lager.put(-1); // Produktion gestoppt
        } catch (InterruptedException e) {
        }
    }
}

import java.util.Random;

public class Konsument extends Thread {
    private 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" + (++i) + ". Wert " + value
                    + " aus Lager entfernt");
                Thread.sleep(1000 + random.nextInt(1000));
                System.out.println("\t" + i + ". Wert " + value
                    + " konsumiert");
            }
        } catch (InterruptedException e) {
        }
    }
}
```

```
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();  
        }  
  
        // wenn das Lager leer ist  
        this.value = value;  
        full = true;  
        notify();  
    }  
  
    public synchronized int get() throws InterruptedException {  
        while (!full) { // solange das Lager leer ist  
            wait();  
        }  
  
        // wenn das Lager voll ist  
        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();  
    }  
}
```

Ausgabebeispiel:

1. Wert 82 wird produziert
1. Wert 82 auf Lager
 - 1. Wert 82 aus Lager entfernt
2. Wert 57 wird produziert
 - 1. Wert 82 konsumiert
2. Wert 57 auf Lager
3. Wert 97 wird produziert
 - 2. Wert 57 aus Lager entfernt
 - 2. Wert 57 konsumiert
3. Wert 97 auf Lager
4. Wert 45 wird produziert
 - 3. Wert 97 aus Lager entfernt
 - 3. Wert 97 konsumiert
4. Wert 45 auf Lager
5. Wert 23 wird produziert
 - 4. Wert 45 aus Lager entfernt
 - 4. Wert 45 konsumiert
5. Wert 23 auf Lager
6. Wert 24 wird produziert
 - 5. Wert 23 aus Lager entfernt

6. Wert 24 auf Lager
7. Wert 53 wird produziert
 5. Wert 23 konsumiert
 6. Wert 24 aus Lager entfernt
 6. Wert 24 konsumiert
7. Wert 53 auf Lager
8. Wert 88 wird produziert
 7. Wert 53 aus Lager entfernt
 7. Wert 53 konsumiert
8. Wert 88 auf Lager
9. Wert 30 wird produziert
 8. Wert 88 aus Lager entfernt
 8. Wert 88 konsumiert
9. Wert 30 auf Lager
 9. Wert 30 aus Lager entfernt
10. Wert 49 wird produziert
10. Wert 49 auf Lager
 9. Wert 30 konsumiert
 10. Wert 49 aus Lager entfernt
 10. Wert 49 konsumiert

Achtung: wait immer in einer while-Schleife

wait sollte immer in einer while-Schleife aufgerufen werden:

```
while (Bedingung)
    wait();
```

Es könnte Situationen geben, in denen beim Aufwecken die Wartebedingung noch weiterhin erfüllt ist. if anstelle von while würde dann ggf. zu einem logischen Fehler führen.

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, 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.

Ab Java SE 5 existieren verschiedene Implementierungen des Interfaces `java.util.concurrent.BlockingQueue<T>` zur Lösung derartiger Probleme.⁵

Pipes

`PipedInputStream` und `PipedOutputStream` (siehe Tabelle 8-1) bieten Methoden, um Daten zwischen zwei Threads über eine so genannte *Pipe* auszutauschen.

⁵ Siehe auch Aufgabe 8 dieses Kapitels.

Eine Pipe verbindet einen `PipedOutputStream src` mit einem `PipedInputStream snk`. Ein Thread schreibt Daten in `src`, die von einem anderen Thread aus `snk` gelesen werden.

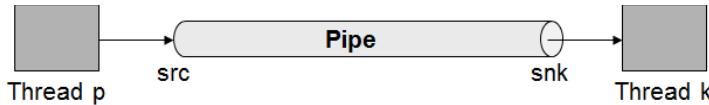


Abbildung 9-5: Eine Pipe

Schreib- und Lesevorgänge sind über einen internen Puffer entkoppelt.

Konstruktoren:

`PipedInputStream()`
`PipedOutputStream()`

Ein `PipedOutputStream` muss mit einem `PipedInputStream` verbunden werden. Dies geschieht entweder mit der `PipedOutputStream`-Methode

`void connect(PipedInputStream snk)`

oder mit der `PipedInputStream`-Methode

`void connect(PipedOutputStream src).`

Programm 9.11 demonstriert den Pipe-Mechanismus. Ein Produzent erzeugt Zahlen, die der Konsument sukzessive aufaddiert.

Programm 9.11

```
import java.io.IOException;
import java.util.Random;

public class Produzent extends Thread {
    private Pipe pipe;

    public Produzent(Pipe p) {
        this.pipe = p;
    }

    public void run() {
        Random random = new Random();
        try {
            for (int i = 0; i < 10; i++) {
                int value = random.nextInt(10);
                System.out.println("Produzent: " + value);
                pipe.put(value);
                Thread.sleep(1000 + random.nextInt(2000));
            }
        } catch (InterruptedException e) {
        } catch (IOException e) {
        }
    }
}
```

```
        System.err.println(e);
    }
}

import java.io.EOFException;
import java.io.IOException;

public class Konsument extends Thread {
    private Pipe pipe;

    public Konsument(Pipe p) {
        this.pipe = p;
    }

    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 e) {
        } catch (EOFException e) {
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class Pipe {
    private DataOutputStream out;
    private 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();
}
```

Threads anhalten und fortsetzen

Die Methoden `wait` und `notify` können benutzt werden, um die Ausführung eines Threads vorübergehend anzuhalten und wieder fortzusetzen.

Programm 9.12

```
import java.io.IOException;

public class Zeitanzeige extends Thread {
    private boolean stop;

    public void run() {
        try {
            while (true) {
                synchronized (this) {
                    while (stop)
                        wait();
                }
                System.out.println(new java.util.Date());
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
        }
    }

    public synchronized void doSuspend() {
        stop = true;
    }

    public synchronized void doResume() {
        stop = false;
        notify();
    }

    public static void main(String[] args) throws IOException {
        Zeitanzeige zeit = new Zeitanzeige();
        zeit.start();

        int b;
        Steuerung: while (true) {
            b = System.in.read();
```

```
        switch (b) {
            case 's': // stoppen
                zeit.doSuspend();
                break;

            case 'e': // beenden
                zeit.interrupt();
                break Steuerung;

            case 'w': // weiter
                zeit.doResume();
            }
        }
    }
```

9.4 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.

Programm 9.13 (`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.

Programm 9.13

```
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;

public class ShutdownTest1 {
    private 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(5000);
        } catch (InterruptedException e) {
        }

        throw new RuntimeException();
    }

    public void close() throws IOException {
        if (log != null)
            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.

```
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;

public class ShutdownTest2 {
    private BufferedWriter log;

    public ShutdownTest2() throws IOException {
        log = new BufferedWriter(new FileWriter("log.txt"));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                try {
                    close();
                } catch (IOException e) {
                }
            }
        });
    }

    public void process() throws IOException {
        log.write("Das ist ein Test.");
        log.newLine();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }

        throw new RuntimeException();
    }

    public void close() throws IOException {
        if (log != null)
            log.close();
    }

    public static void main(String[] args) throws IOException {
        ShutdownTest2 test = new ShutdownTest2();
        test.process();
    }
}
```

Hier wird ein Thread-Objekt erzeugt (im Beispiel als Instanz einer anonymen inneren Klasse), dessen run-Methode die close-Methode aufruft. Dieses Thread-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.

Ein Objekt der Klasse Runtime repräsentiert das gestartete Programm und die Umgebung, in der es abläuft. Die Klassenmethode Runtime.getRuntime() liefert eine Referenz auf das aktuelle Runtime-Objekt.

In Eclipse (zumindest bei Version "Luna Release 4.4.0") wird der Shutdown-Hook nicht gestartet, wenn das Programm über den Terminate-Button beendet wird. Dieser Button "killt" den JVM-Prozess direkt, sodass der Shutdown-Hook nicht mehr ausgeführt werden kann.

9.5 Aufgaben

1. Erstellen Sie ein Programm mit zwei Threads, das die beiden Wörter "Hip" und "HOP" in einer Endlosschleife unterschiedlich schnell ausgibt. Realisieren Sie zwei Programmvarianten:
 - Ableitung von der Klasse Thread
 - Implementierung des Interfaces Runnable.
2. Schreiben Sie ein Programm, das den Namen und die Priorität des Threads ausgibt, der die Methode main ausführt.
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.
4. 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.

5. 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.
6. 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 anderer Thread (Konsument) verbraucht diese Zahlen, indem er sie in einem Balkendiagramm (Ausgabe von so vielen Zeichen '*' wie die Zahl angibt) darstellt. Ist die Zahl verbraucht, 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. Die Produktion soll zudem über Tastatureingabe angehalten und wieder fortgesetzt werden können. Nutzen Sie zur Synchronisation `synchronized`-Blöcke sowie die Methoden `wait` und `notify`.
7. Erstellen Sie eine Variante zur Lösung von Aufgabe 6, die anstelle eines Vektors den Pipe-Mechanismus nutzt (siehe Programm 9.11).
8. Erstellen Sie eine Variante zur Lösung von Aufgabe 6, 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 hinzu; 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 Konstruktur `LinkedBlockingQueue(int capacity)` erzeugt eine Instanz mit der angegebenen Aufnahmekapazität.

9. 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 200000000 ausgegeben? Offensichtlich gehen Erhöhungen des

Zählers verloren. Wie kann das Programm so geändert werden, dass stets der Endstand 200000000 beträgt?

10. Ein Patient besucht eine Arztpraxis mit genau einem Behandlungsraum. Er erhält eine Wartenummer 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 nächste auszugebende Wartenummer, die aktuell aufgerufene Nummer und den Status "besetzt". Die Klasse enthält die folgenden Methoden:

```
int registrieren()
```

liefert die nächste Wartenummer. Die Registrierung dauert 3 bis 8 Sekunden.

```
void betreten(int nummer)
```

Diese Methode wartet, falls der Raum besetzt ist oder die eigene Wartenummer nicht mit der aufgerufenen Nummer übereinstimmt. Anschließend wird der Raum als besetzt gekennzeichnet und die aufgerufene Nummer um 1 erhöht.

```
void verlassen()
```

Der Raum wird freigegeben und alle wartenden Threads werden geweckt.

Nutzen Sie geeignete Synchronisationmechanismen.

11. Ein Ringpuffer soll Messpunkte speichern. Ein Messpunkt-Objekt verwaltet eine Zeitpunktangabe und einen numerischen Wert. Die Implementierung des Puffers soll analog zu Aufgabe 24 in Kapitel 3 erfolgen. Schreibende und lesende Zugriffe sollen bei wechselseitigem Ausschluss in jeweils eigenen Threads stattfinden. Die Threads sollen mehrere schreibende bzw. lesende Zugriffe hintereinander mit zeitlichem Verzug in einer Schleife ausführen.
12. 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 waren 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.⁶

Implementieren Sie Programm, das dieses Verhalten simuliert.

⁶ Siehe <http://de.wikipedia.org/wiki/Philosophenproblem>

10 Grafische Benutzeroberflächen

Die Klassen des Pakets `java.awt` stellen grundlegende Elemente zur Erstellung grafischer Benutzeroberflächen, auch *GUI (Graphical User Interface)* genannt, bereit. Die Abkürzung AWT steht für *Abstract Window Toolkit*. Mit Hilfe dieser Klassen können Fenster mit Menüs, Textfelder und Bedienungselementen, wie z. B. Schaltflächen und Auswahllisten, realisiert werden. Zur Darstellung vieler AWT-Komponenten wird intern auf das jeweilige Betriebssystem zurückgegriffen (*Peer-Ansatz*). So hat z. B. ein Button auf den verschiedenen Plattformen immer das für die Plattform typische Erscheinungsbild. Damit beschränkt sich AWT auf diejenigen GUI-Komponenten, die von allen unterstützten Betriebssystemen dargestellt werden können. AWT ist Bestandteil der Sammlung *Java Foundation Classes* (JFC), die weitere GUI-Komponenten auf Basis eines geänderten Konzepts (*Swing-Klassen*) und grafische Verbesserungen enthält.

Diese *Swing-Klassen* befinden sich in dem Paket: `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, weiter. Dieses Kapitel zeigt die vielfältigen Möglichkeiten, die AWT und *Swing* bieten.

Es gibt eine Reihe von Design-Tools zur visuellen Erstellung grafischer Oberflächen mit Swing, z. B. das Eclipse-Plugin *WindowBuilder*.¹

Lernziele

In diesem Kapitel lernen Sie

- Methoden zum *Zeichnen* von einfachen 2-dimensionalen geometrischen Figuren,
- *GUI-Komponenten* für die Interaktion mit dem Benutzer,
- *Container*, die andere Komponenten aufnehmen können,
- *Layout-Manager*, die die Komponenten in einem Container anordnen und
- Methoden zur Behandlung von Ereignissen (*Event-Handling*) kennen.

¹ <http://www.eclipse.org/windowbuilder/>

10.1 Übersicht

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. Abbildung 10-1 enthält die *Hierarchie der hier behandelten Klassen*.

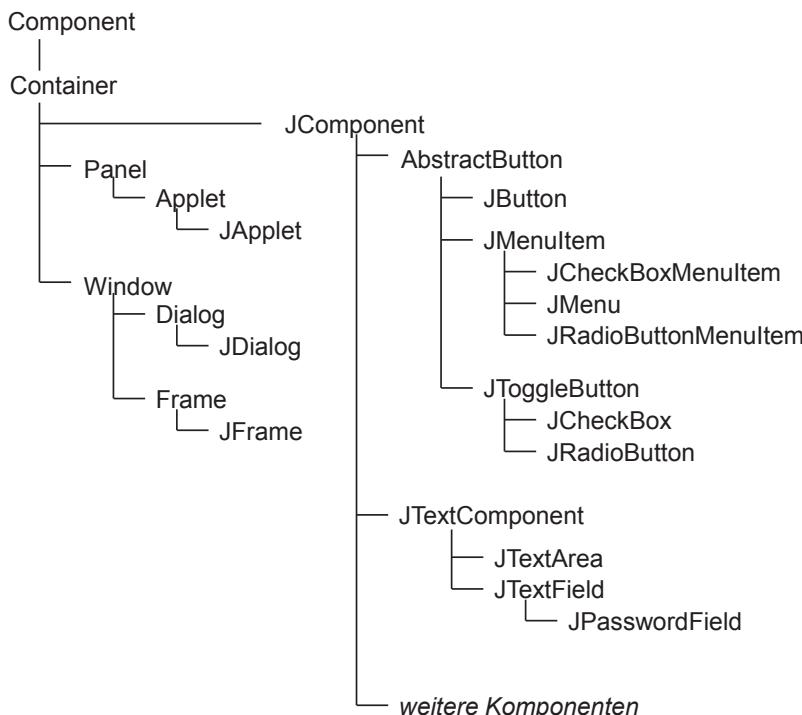


Abbildung 10-1: Klassenhierarchie Komponenten und Container

Die Klassen `Component`, `Container`, `Panel`, `Window`, `Dialog` und `Frame` sind AWT-Klassen aus dem Paket `java.awt`, `Applet` liegt im Paket `java.applet`, alle anderen Klassen des Diagramms in Abbildung 10-1 liegen im Paket `javax.swing`. Weitere Komponenten sind:

<code>JComboBox</code>	<code>JOptionPane</code>	<code>JSlider</code>
<code>JFileChooser</code>	<code>JPanel</code>	<code>JSplitPane</code>
<code>JLabel</code>	<code>JPopupMenu</code>	<code>JTabbedPane</code>
<code>JList</code>	<code>JProgressBar</code>	<code>JTable</code>
<code>JMenuBar</code>	<code>JScrollPane</code>	<code>JToolBar</code>

Eigenschaften von Komponenten

Die verschiedenen 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 (z. B. die Auswahlliste `JList`, die Tabelle `JTable`) 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 Ansicht (*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 Viewer oft zu einer Einheit zusammengefasst.

10.2 JFrame

Objekte der Klasse `javax.swing.JFrame` sind frei bewegliche Fenster mit Rahmen und Titelleiste.

Konstruktoren sind:

```
JFrame()  
JFrame(String title)
```

Im zweiten Fall erhält das Fenster den Titel `title`.

`JFrame`-Methoden:

```
void setTitle(String title)  
    setzt den Titel des Fensters.  
  
String getTitle()  
    liefert den Titel des Fensters.
```

```
void setDefaultCloseOperation(int op)
```

legt die Operation fest, die ausgeführt wird, wenn das Fenster durch Anklicken des entsprechenden Buttons in der Titelleiste geschlossen wird. Für op kann eine der folgenden JFrame-Konstanten gewählt werden: DO NOTHING ON CLOSE, HIDE ON CLOSE (Voreinstellung), DISPOSE ON CLOSE, EXIT ON CLOSE

```
void setResizable(boolean b)
```

gibt an, ob die Größe des Fensters durch den Benutzer verändert werden kann.

```
boolean isResizable()
```

prüft, ob die Größe des Fensters vom Benutzer verändert werden kann.

Dimension

Die Klasse `java.awt.Dimension` repräsentiert Breite und Höhe. Sie besitzt die beiden public-Attribute `width` und `height` vom Typ `int` und den Konstruktor

```
Dimension(int width, int height)
```

Point

Die Klasse `java.awt.Point` repräsentiert einen Punkt durch seine Koordinaten. Sie besitzt die beiden public-Attribute `x` und `y` vom Typ `int` und den Konstruktor

```
Point(int x, int y)
```

Component-Methoden

Größe und Position einer Komponente (und damit auch eines Fensters) können mit folgenden Methoden der Klasse `Component` festgelegt bzw. abgefragt werden:

```
void setSize(int width, int height)
```

```
void setSize(Dimension d)
```

setzen die Größe der Komponente auf den Wert Breite mal Höhe (in Pixel).

```
Dimension getSize()
```

liefert die aktuelle Größe der Komponente.

```
void setLocation(int x, int y)
```

```
void setLocation(Point p)
```

platzieren die linke obere Ecke der Komponente an Position (x,y) der übergeordneten Komponente.

```
Point getLocation()
```

liefert die linke obere Ecke der Komponente relativ zum Koordinatensystem der übergeordneten Komponente.

Koordinatensystem

Der Ursprung ($0,0$) des *Koordinatensystems* liegt in der linken oberen Ecke. Positive x-Werte erstrecken sich nach rechts, positive y-Werte nach unten.

Die Größe des Bildschirms kann über den folgenden Aufruf ermittelt werden:

```
getToolkit().getScreenSize()
```

Zurückgegeben wird eine Instanz der Klasse Dimension. Die Component-Methode getToolkit liefert eine Instanz der Klasse `java.awt.Toolkit`.

```
void setIconImage(Image image)
```

legt das Icon fest, das links in der Titelleiste des Fensters angezeigt wird.

Ein `Image`-Objekt kann über das Toolkit-Objekt bereitgestellt werden.²

```
void setVisible(boolean b)
```

macht die Komponente sichtbar (`true`) oder unsichtbar (`false`).

Die Window-Methode `void dispose()` löscht das Fenster und gibt die belegten Ressourcen frei.

Mit der Window-Methode `setCursor` wird die Darstellung des Mauszeigers festgelegt:

```
void setCursor(Cursor c)
```

Cursor

Die Klasse `java.awt.Cursor` repräsentiert den *Mauszeiger*.

Cursor-Methoden:

```
static Cursor getPredefinedCursor(int type)
```

liefert einen vordefinierten Zeiger. Für `type` kann z. B. eine der folgenden Klassenkonstanten vom Typ `int` eingesetzt werden: CROSSHAIR_CURSOR, DEFAULT_CURSOR, HAND_CURSOR, MOVE_CURSOR, TEXT_CURSOR oder WAIT_CURSOR.

```
String getName()
```

liefert den Namen des Zeigers.

Das folgende Beispielprogramm wechselt die Zeigerdarstellung nach einer Wartezeit. Dazu ist der Mauszeiger vom Benutzer in das Fenster zu bewegen. Die Zeigernamen werden jeweils als Titel angezeigt.

² Näheres hierzu in Kapitel 11.4.

Programm 10.1

```
import java.awt.Cursor;  
  
import javax.swing.JFrame;  
  
public class Fenster extends JFrame {  
    public Fenster() {  
        super("Einfaches Fenster");  
  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setSize(400, 200);  
        setLocation(100, 100);  
        setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        Fenster frame = new Fenster();  
  
        int[] types = { Cursor.CROSSHAIR_CURSOR, Cursor.HAND_CURSOR,  
                      Cursor.MOVE_CURSOR, Cursor.TEXT_CURSOR, Cursor.WAIT_CURSOR,  
                      Cursor.DEFAULT_CURSOR };  
  
        for (int type : types) {  
            try {  
                Thread.sleep(3000);  
            } catch (InterruptedException e) {}  
  
            Cursor c = Cursor.getPredefinedCursor(type);  
            frame.setCursor(c);  
            frame.setTitle(c.getName());  
        }  
    }  
}
```

10.3 JPanel und Methoden zum Zeichnen

JPanel

Ein *Panel* ist die "Arbeitsfläche" für GUI-Komponenten. Die Klasse `javax.swing.JPanel` ist die einfachste Form eines Containers. Als Subklasse von `javax.swing.JComponent` können Panels in andere Container eingefügt werden und auch ineinander verschachtelt werden. Um ein Panel anzeigen zu können, muss es in ein Fenster eingefügt werden.

Mit der Container-Methode `add` können neue Komponenten in einen Container aufgenommen werden.

Content Pane

Objekte vom Typ `JFrame`, `JDialog` und `JApplet` enthalten jeweils einen speziellen Container (*Content Pane*), an den der Aufruf von `add` zu richten ist. Diesen Container erhält man mit der `JFrame`-, `JDialog`- bzw. `JApplet`-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. Auch hier wird `component` in den Content Pane aufgenommen. Das gilt sowohl für `JFrame` als auch für `JDialog` und `JApplet`.

Größe von Komponenten

Die `JComponent`-Methoden

```
void setMinimumSize(Dimension size)  
void setMaximumSize(Dimension size)  
void setPreferredSize(Dimension size)
```

legen die minimale, maximale bzw. bevorzugte Größe der Komponente fest.

Diese Methoden nutzt man in der Regel bei benutzerdefinierten Komponenten.

Die Größe kann allerdings von Layout-Managern³ verändert werden.

Die `Window`-Methode

```
void pack()
```

gibt dem Fenster die Größe, die zur Darstellung der enthaltenen Komponenten nötig ist.

Graphics – paintComponent und repaint

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 so genannten *Grafikkontext* darstellt und vom Java-Laufzeitsystem übergeben wird. Der Grafikkontext enthält diverse für das Zeichnen notwendige Basisinformationen.

³ Siehe Kapitel 10.5.

`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.

Einige Methoden der Klasse `Graphics`:

Color

`void setColor(Color c)`

setzt die Farbe, die dann von allen folgenden Operationen benutzt wird. Für `c` kann z. B. eine der folgenden Konstanten der Klasse `Color` eingesetzt werden: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white` und `yellow`.

Die Klasse `java.awt.Color` stellt Farben nach dem *RGB-Farbmodell* dar. Der Konstruktor

`Color(int r, int g, int b)`

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.

`Color(int r, int g, int b, int a)`

Im Vergleich zum vorhergehenden Konstruktor wird hier zusätzlich der Alpha-Wert (0 - 255) angegeben: 0 = transparent, 255 = deckend.

`Color getColor()`

liefert die aktuelle Farbe.

Font

Mit der Klasse `java.awt.Font` wird die Schrift für die Anzeige von Texten festgelegt.

Der Konstruktor

```
Font(String name, int style, int size)
```

erzeugt ein neues Font-Objekt, wobei name den Namen der gewünschten Schriftart (z. B. "Dialog", "DialogInput", "SansSerif", "Serif", "Monospaced"), style die Schriftauszeichnung Font.BOLD (fett), Font.ITALIC (kursiv) und Font.PLAIN (normal) und size die Punktgröße angibt. Font.BOLD kann durch Addition mit Font.ITALIC kombiniert werden.

```
void setFont(Font f)
```

setzt die Schriftart, die dann von allen folgenden Operationen, die Text ausgeben, verwendet wird.

```
Font getFont()
```

liefert die aktuelle Schriftart.

setFont und getFont gibt es auch als Component-Methoden.

Zeichenmethoden drawXxx

```
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.

```
void drawLine(int x1, int y1, int x2, int y2)
```

zeichnet eine Linie von (x1,y1) bis (x2,y2).

```
void drawRect(int x, int y, int b, int h)
```

zeichnet ein Rechteck mit den Eckpunkten (x,y), (x+b,y), (x,y+h), (x+b,y+h).

```
void drawRoundRect(int x, int y, int h, int b, int arcB, int arcH)
```

zeichnet ein Rechteck mit abgerundeten Ecken. arcB ist der horizontale, arcH der vertikale Durchmesser des Bogens.

```
void drawPolygon(int[] x, int[] y, int n)
```

zeichnet ein geschlossenes Polygon mit den Eckpunktkoordinaten x[i] und y[i]. n ist die Anzahl der Eckpunkte.

```
void drawOval(int x, int y, int b, int h)
```

zeichnet ein Oval innerhalb des durch die Argumente spezifizierten Rechtecks. Ein Kreis entsteht, wenn b und h gleich sind.

```
void drawArc(int x, int y, int b, int h, int start, int arc)
```

zeichnet einen Kreis- bzw. Ellipsenbogen innerhalb des spezifizierten Rechtecks vom Startwinkel start bis zum Winkel start + arc. Positive Winkel (0 - 360) bedeuten eine Drehung gegen den Uhrzeigersinn, ein Winkel von 0 Grad bezeichnet die 3-Uhr-Position.

Zeichenmethoden fillXxx

Rechtecke, Polygone, Ovale und Kreis- bzw. Ellipsenbögen können auch im Füllmodus mit der eingestellten Farbe gezeichnet werden. Die Methodennamen beginnen dann mit `fill` statt `draw`. Die Koordinaten der Eckpunkte eines ausgefüllten Rechtecks liegen hier aber bei `x` und `x+b-1` bzw. `y` und `y+h-1`.

Vorder- und Hintergrundfarbe

Die folgenden Component-Methoden setzen bzw. liefern die Vorder- bzw. Hintergrundfarbe einer Komponente:

```
void setForeground(Color c)
void setBackground(Color c)
Color getForeground()
Color getBackground()
```

Die JComponent-Methode

```
void setOpaque(boolean b)
```

gibt der Komponente einen deckenden (`true`) oder durchsichtigen (`false`) Hintergrund. Welcher Wert voreingestellt ist, hängt vom *Look & Feel* der Komponente ab.

Programm 10.2

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class Zeichnen extends JFrame {
    public Zeichnen() {
        super("Grafik");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(new MyPanel());
        pack();
        setVisible(true);
    }

    private class MyPanel extends JPanel {
        public MyPanel() {
            setBackground(Color.white);
            setPreferredSize(new Dimension(300, 200));
        }

        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.setColor(Color.red);
            g.fillRoundRect(30, 30, 240, 140, 30, 30);
            g.setColor(Color.white);
```

```
        g.setFont(new Font("Monospaced", Font.BOLD, 48));
        g.drawString("Hallo!", 65, 110);
    }
}

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



Abbildung 10-2: Grafik im Fenster

10.4 Ereignisbehandlung

Die Kommunikation zwischen Benutzer und Anwendungsprogramm mit grafischer Oberfläche basiert auf einem so genannten Ereignismodell (*Event-Modell*).

Bisher haben wir das mit einem Fenster verbundene Programm mit Hilfe des Aufrufs

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
```

beendet. Nun wollen wir durch Ausführung des Befehls *Schließen* im Systemmenü des Fensters bzw. durch einen Mausklick auf das entsprechende 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 wesentliche Interfaces und Klassen, die die Reaktion auf derartige Ereignisse realisieren, und ist somit in jede Klasse einzubinden, die Ereignisse behandeln soll.

Eine Aktion, wie z. B. das Klicken auf einen Menüpunkt, 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 so genannten *Ereignisempfängern* (*Listener*), die bei der Ereignisquelle registriert werden müssen, wenn sie entsprechende Ereignisse mitgeteilt bekommen sollen.

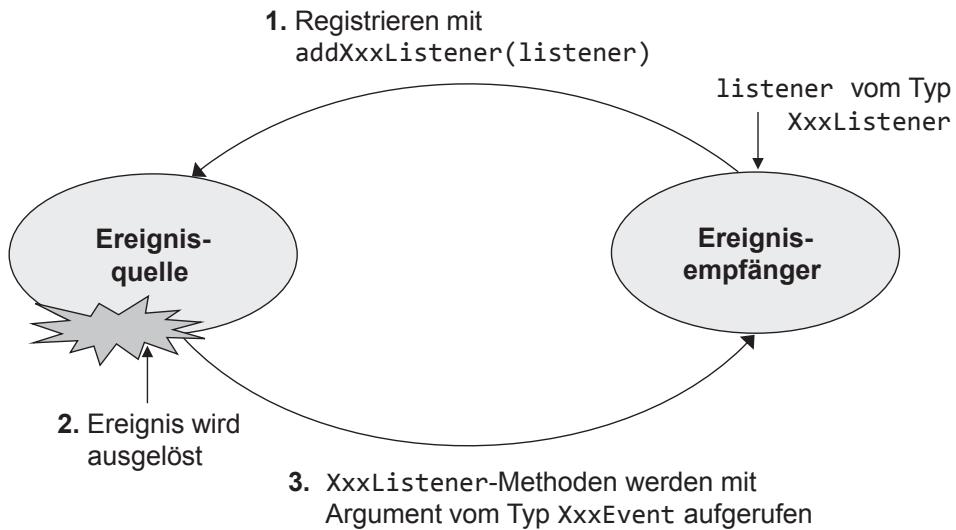


Abbildung 10-3: Ereignismodell

In den nachfolgenden Beispielen 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 1) bzw.  
void addKeyListener(KeyListener 1)
```

Die Mitteilung eines Ereignisses besteht im Aufruf der passenden Interface-Methode des Listener-Objekts, die als Argument die Referenz auf das entsprechende Ereignisobjekt enthält.

Alle Ereignisklassen besitzen eine Methode, mit deren Hilfe das Objekt ermittelt werden kann, das das Ereignis ausgelöst hat: `Object getSource()`

Im Programm 10.3 ist das Fenster sowohl Ereignisquelle als auch -empfänger. Die Klasse `Close1` implementiert die Interfaces `WindowListener` und `KeyListener`. Die einzige relevanten Methoden für die Reaktion auf die Ereignisse in diesem Beispiel sind

`windowClosing` und `keyPressed`. Die anderen Methoden der beiden Interfaces haben deshalb leere Methodenrümpfe. Die `KeyEvent`-Methode `getKeyCode` gibt den Code der gedrückten Taste zurück, der dann mit `KeyEvent.VK_ESCAPE` verglichen wird. Beim Schließen des Fensters wird die Zeit in Sekunden, die das Fenster geöffnet war, ausgegeben.

Programm 10.3

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JFrame;

public class Close1 extends JFrame implements WindowListener, KeyListener {
    private long start;

    public Close1() {
        super("Demo Ereignisbehandlung");

        addWindowListener(this);
        addKeyListener(this);

        setSize(400, 200);
        setVisible(true);
        start = System.currentTimeMillis();
    }

    private void printTime() {
        long end = System.currentTimeMillis();
        double time = (end - start) / 1000.;
        System.out.println("Das Fenster war " + time + " Sekunden geöffnet.");
    }

    public void windowClosing(WindowEvent e) {
        printTime();
        System.exit(0);
    }

    public void windowActivated(WindowEvent e) {
    }

    public void windowClosed(WindowEvent e) {
    }

    public void windowDeactivated(WindowEvent e) {
    }

    public void windowDeiconified(WindowEvent e) {
    }
```

```

public void windowIconified(WindowEvent e) {
}

public void windowOpened(WindowEvent e) {

}

public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
        printTime();
        System.exit(0);
    }
}

public void keyReleased(KeyEvent e) {

}

public void keyTyped(KeyEvent e) {

}

public static void main(String[] args) {
    new Close1();
}
}

```

Adapterklasse

Eine *Adapterklasse* ist hier eine abstrakte Klasse, die ein vorgegebenes Interface mit leeren Methodenrumpfen implementiert.

Die Adapterklassen `WindowAdapter` und `KeyAdapter` implementieren `Window-Listener` bzw. `KeyListener`. Sie werden im Programm 10.4 genutzt, um nur die gewünschten Methoden in Subklassen zu überschreiben und damit Schreibaufwand für die nicht relevanten Methoden des Listener-Interfaces zu sparen. Diese Subklassen sind hier als *Instanzklassen*⁴ realisiert.

Programm 10.4

```

import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;

public class Close2 extends JFrame {
    private long start;

    public Close2() {
        super("Demo Ereignisbehandlung");

        addWindowListener(new MyWindowListener());
        addKeyListener(new MyKeyListener());
    }
}

```

⁴ Siehe Kapitel 3.9.

```
setSize(400, 200);
setVisible(true);
start = System.currentTimeMillis();
}

private void printTime() {
    long end = System.currentTimeMillis();
    double time = (end - start) / 1000.;
    System.out.println("Das Fenster war " + time + " Sekunden geöffnet.");
}

private class MyWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        printTime();
        System.exit(0);
    }
}

private class MyKeyListener extends KeyAdapter {
    public void keyPressed(KeyEvent e) {
        if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
            printTime();
            System.exit(0);
        }
    }
}

public static void main(String[] args) {
    new Close2();
}
}
```

Wenn sehr wenig Code für die Ereignisempfänger benötigt wird, empfehlen sich auch *anonyme Klassen*.

Programm 10.5

```
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;

public class Close3 extends JFrame {
    private long start;

    public Close3() {
        super("Demo Ereignisbehandlung");

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                printTime();
                System.exit(0);
            }
        });
    }

    void printTime() {
        long end = System.currentTimeMillis();
        double time = (end - start) / 1000.;
        System.out.println("Das Fenster war " + time + " Sekunden geöffnet.");
    }
}
```

```

        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
                    printTime();
                    System.exit(0);
                }
            }
        });

        setSize(400, 200);
        setVisible(true);
        start = System.currentTimeMillis();
    }

    private void printTime() {
        long end = System.currentTimeMillis();
        double time = (end - start) / 1000.;
        System.out.println("Das Fenster war " + time + " Sekunden geöffnet.");
    }

    public static void main(String[] args) {
        new Close3();
    }
}

```

Die Tabellen 10-1 und 10-2 fassen die bisher behandelten Ereignisse zusammen.

Tabelle 10-1: WindowEvent

Ereignisklasse	<code>java.awt.event.WindowEvent</code>
Ereignisquellen	Window und Subklassen
Registrierungsmethode	<code>addWindowListener</code>
Listener-Interface	<code>java.awt.event.WindowListener</code>
Adapterklasse	<code>java.awt.event.WindowAdapter</code>

Listener-Methoden	Beschreibung
<code>windowActivated</code>	Fenster wurde aktiviert
<code>windowClosed</code>	Fenster wurde geschlossen
<code>windowClosing</code>	Fenster wird geschlossen
<code>windowDeactivated</code>	Fenster wurde deaktiviert
<code>windowDeiconified</code>	Fenster wurde wieder hergestellt
<code>windowIconified</code>	Fenster wurde auf Symbolgröße verkleinert
<code>windowOpened</code>	Fenster wurde geöffnet

Tabelle 10-2: KeyEvent

Ereignisklasse	java.awt.event.KeyEvent
Ereignisquellen	Component und Subklassen
Registrierungsmethode	<code>addKeyListener</code>
Listener-Interface	java.awt.event.KeyListener
Adapterklasse	java.awt.event.KeyAdapter

Listener-Methoden	Beschreibung
<code>keyPressed</code>	Taste wurde gedrückt
<code>keyReleased</code>	Taste wurde losgelassen
<code>keyTyped</code>	Taste wurde gedrückt und losgelassen

Allgemein gilt:

Zu allen Registrierungsmethoden `addXxxListener` existieren die dazu passenden Deregistrierungsmethoden `removeXxxListener`.

10.5 Layout-Manager

Die Anordnung von Komponenten in einem Container erfolgt mit Hilfe von so genannten *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 mgr)
```

der Klasse `Container` setzt den Layout-Manager für den Container. Alle Layout-Manager implementieren das Interface `java.awt.LayoutManager`.

Null-Layout

Ein so genanntes *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`⁵ oder einfacher mit `setBounds` positioniert:

⁵ Siehe Kapitel 10.2.

```
void setBounds(int x, int y, int width, int height)
```

Alle Beispiele dieses Kapitels nutzen Objekte der Klasse `MyPanel` als zu platzierende Komponenten.

Programm 10.6

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JPanel;

public class MyPanel extends JPanel {
    private int id;

    public MyPanel(int id) {
        this.id = id;
        setBackground(Color.lightGray);
        setPreferredSize(new Dimension(50, 50));
        setMaximumSize(new Dimension(50, 50));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawString(String.valueOf(id), 5, 15);
    }
}

import java.awt.Color;
import java.awt.Container;

import javax.swing.JFrame;

public class NullLayoutTest extends JFrame {
    public NullLayoutTest() {
        super("NullLayout");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Container c = getContentPane();
        c.setBackground(Color.white);
        c.setLayout(null);

        MyPanel p1 = new MyPanel(1);
        MyPanel p2 = new MyPanel(2);
        p1.setBounds(10, 10, 100, 100);
        p2.setBounds(120, 10, 100, 50);
        c.add(p1);
        c.add(p2);

        setSize(300, 200);
        setVisible(true);
    }
}
```

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

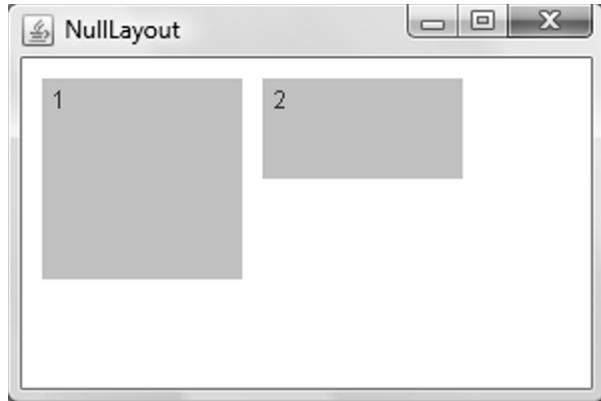


Abbildung 10-4: Null-Layout

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.

`FlowLayout()`
`FlowLayout(int align)`
`FlowLayout(int align, int hgap, int vgap)`
erzeugen jeweils das Layout.

`align` bestimmt mit Hilfe von Konstanten dieser Klasse die Anordnung in einer Zeile: `CENTER` (zentriert), `LEFT` (linksbündig), `RIGHT` (rechtsbündig). Als Voreinstellung wird `CENTER` verwendet. `hgap` bestimmt den horizontalen und `vgap` den vertikalen Komponentenabstand. Die Voreinstellung ist 5.

Programm 10.7

```
import java.awt.Color;  
import java.awt.Container;  
import java.awt.FlowLayout;  
  
import javax.swing.JFrame;  
  
public class FlowLayoutTest extends JFrame {  
    public FlowLayoutTest() {  
        super("FlowLayout");  
  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```

Container c = getContentPane();
c.setBackground(Color.white);
c.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));

for (int i = 1; i <= 5; i++)
    c.add(new MyPanel(i));

setSize(300, 200);
setVisible(true);
}

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

```

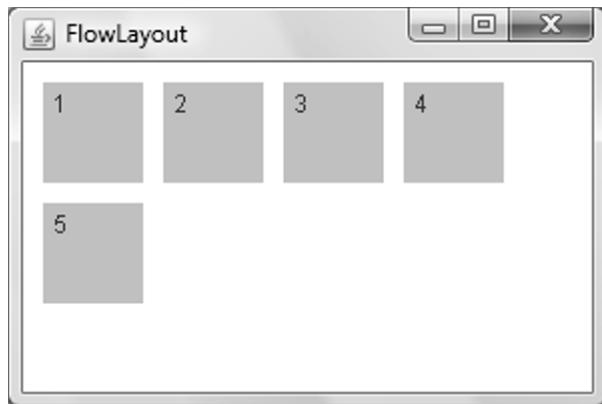


Abbildung 10-5: FlowLayout

BorderLayout

`java.awt.BorderLayout` ist der Standard-Layout-Manager für den *Content Pane*⁶ von `JFrame`, `JDialog` und `JApplet`. 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.

`BorderLayout()`

`BorderLayout(int hgap, int vgap)`

erzeugen das Layout ohne bzw. mit Komponentenabstand.

⁶ Siehe Kapitel 10.3.

Programm 10.8

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;

import javax.swing.JFrame;

public class BorderLayoutTest extends JFrame {
    public BorderLayoutTest() {
        super("BorderLayout");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setBackground(Color.white);
        c.setLayout(new BorderLayout(10, 10));

        c.add(new MyPanel(1), BorderLayout.NORTH);
        c.add(new MyPanel(2), BorderLayout.WEST);
        c.add(new MyPanel(3), BorderLayout.CENTER);
        c.add(new MyPanel(4), BorderLayout.EAST);
        c.add(new MyPanel(5), BorderLayout.SOUTH);

        setSize(300, 200);
        setVisible(true);
    }

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

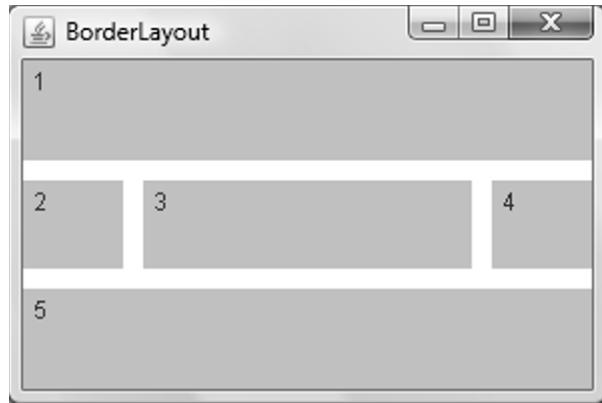


Abbildung 10-6: BorderLayout

GridLayout

`java.awt.GridLayout` ordnet die Komponenten in einem Raster aus Zeilen gleicher Höhe und Spalten gleicher Breite an.

```
GridLayout()
GridLayout(int rows, int cols)
GridLayout(int rows, int cols, int hgap, int vgap)
```

erzeugen ein Layout, bei dem alle Komponenten in einer Zeile liegen bzw. auf rows Zeilen und cols Spalten der Reihe nach verteilt sind. `hgap` und `vgap` geben den horizontalen bzw. vertikalen Komponentenabstand an.

Programm 10.9

```
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JFrame;

public class GridLayoutTest extends JFrame {
    public GridLayoutTest() {
        super("GridLayout");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setBackground(Color.white);
        c.setLayout(new GridLayout(2, 3, 10, 10));

        for (int i = 1; i <= 5; i++)
            c.add(new MyPanel(i));

        setSize(300, 200);
        setVisible(true);
    }

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

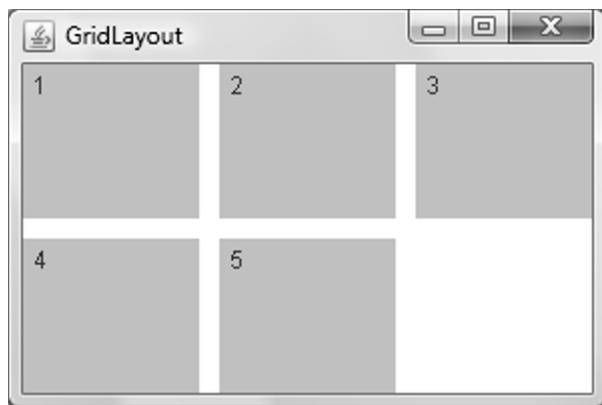


Abbildung 10-7: GridLayout

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.

`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.

Klassenmethoden von `Box`:

`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 in der angegebenen Richtung ausdehnen kann.

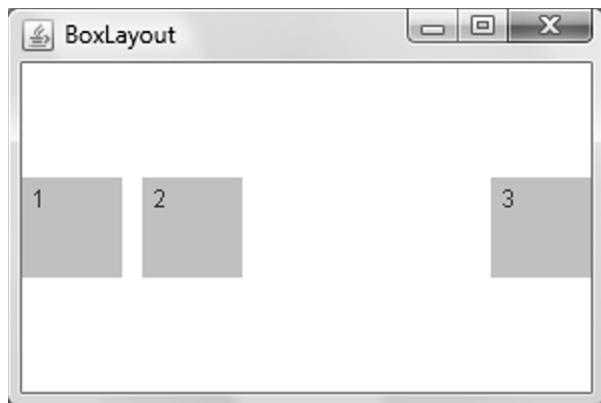


Abbildung 10-8: BoxLayout

Programm 10.10

```
import java.awt.Color;
import java.awt.Container;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JFrame;

public class BoxLayoutTest extends JFrame {
    public BoxLayoutTest() {
        super("BoxLayout");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setBackground(Color.white);
        c.setLayout(new BoxLayout(c, BoxLayout.X_AXIS));

        c.add(new MyPanel(1));
        c.add(Box.createHorizontalStrut(10));
        c.add(new MyPanel(2));
        c.add(Box.createHorizontalGlue());
        c.add(new MyPanel(3));

        setSize(300, 200);
        setVisible(true);
    }

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

Geschachtelte Layouts

Container wie z. B. JPanel können ineinander geschachtelt werden. Programm 10.11 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.

Border

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:

```
void setTitleFont(Font titleFont)
void setTitleColor(Color titleColor)
```

Die beiden Klassen implementieren das Interface `javax.swing.border.Border`.

Mit der `JComponent`-Methode

```
void setBorder(Border border)
```

kann der Rand für die Komponente festgelegt werden.

Programm 10.11

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.TitledBorder;

public class LayoutTest extends JFrame {
    public LayoutTest() {
        super("LayoutTest");

        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 LayoutTest();
    }
}

```



Abbildung 10-9: Geschachtelte Layouts

GridLayout

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.

`GridLayout` enthält die folgenden public-Instanzvariablen:

```
int[] columnWidths
int[] rowHeights
double[] columnWeights
double[] rowWeights
```

Sie enthalten Werte, die die berechneten Werte (Spaltenbreite, Zeilenhöhe, Spalten- und Zeilengewichte) am Ende überschreiben.

Programm 10.12

```
import java.awt.Color;
import java.awt.Container;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;

import javax.swing.JFrame;

public class GridBagLayoutTest extends JFrame {
    public GridBagLayoutTest() {
        super("GridBagLayout");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setBackground(Color.white);

        MyPanel[] p = new MyPanel[10];
        for (int i = 0; i < p.length; i++)
            p[i] = new MyPanel(i + 1);

        GridBagLayout gridbag = new GridBagLayout();
        gridbag.columnWidths = new int[] { 200, 100, 50 };
        gridbag.rowHeights = new int[] { 50, 50, 100, 100, 50, 50 };

        c.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);

        pack();
        setVisible(true);
    }

    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);
    }
}
```

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

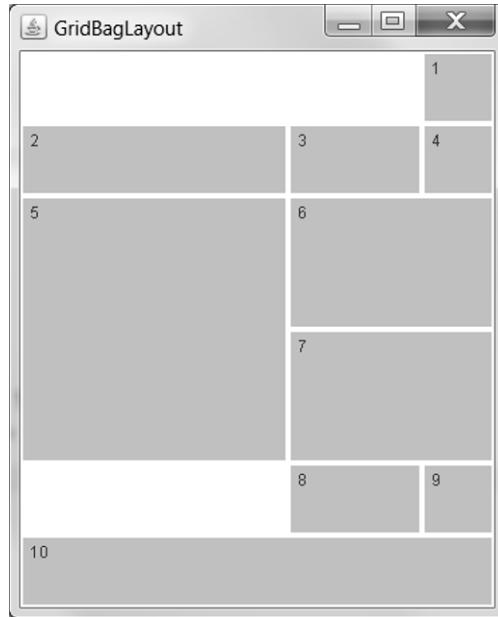


Abbildung 10-10: GridBagLayout

10.6 Buttons

JButton

Die Klasse `javax.swing.JButton` implementiert einen einfachen Button.

Konstruktoren sind:

```
JButton()  
JButton(String text)  
JButton(Icon icon)  
JButton(String text, Icon icon)
```

Sie erzeugen einen Button ohne oder mit Beschriftung bzw. Icon.

Icon

`javax.swing.Icon` ist ein Interface, das von der Klasse `javax.swing.ImageIcon` implementiert wird.

Ein `ImageIcon`-Objekt kann mit den Konstruktoren

`ImageIcon(String datei)` und
`ImageIcon(URL url)`

aus einer Bilddatei (GIF-, JPEG- oder PNG-Format) erzeugt werden. `java.net.URL` repräsentiert den *Uniform Resource Locator*⁷.

Tooltip

Mit der `JComponent`-Methode

```
void setToolTipText(String text)
```

wird eine Komponente mit einem so genannten *Tooltip* ausgestattet, der beim Bewegen des Mauszeigers über diese Komponente angezeigt wird.

AbstractButton

Die Superklasse `javax.swing.AbstractButton` bietet die folgenden Methoden:

`void setText(String text)`

legt die Beschriftung fest.

`String getText()`

liefert die Beschriftung.

`void setIcon(Icon icon)`

legt das Icon fest.

`Icon getIcon()`

liefert das Icon.

`void setHorizontalTextPosition(int pos)`

`void setVerticalTextPosition(int pos)`

legt die horizontale bzw. vertikale Position des Textes relativ zum Icon fest.

Dabei werden für die horizontale Ausrichtung die Konstanten `LEFT`, `CENTER` oder `RIGHT` und für die vertikale Ausrichtung `TOP`, `CENTER` oder `BOTTOM` übergeben.

`void setActionCommand(String cmd)`

setzt den Kommandonamen für das dem Button zugeordnete `ActionEvent`. Als Voreinstellung ist das die Beschriftung des Buttons.

`String getActionCommand()`

liefert den Kommandonamen.

Ereignisbehandlung: `ActionEvent`

Wird ein Button gedrückt, so wird ein `ActionEvent` an seine Ereignisempfänger gesendet.

Registrierungsmethode:

```
void addActionListener(ActionListener l)
```

⁷ Siehe Kapitel 13.1.

```
java.awt.event.ActionListener-Methode:  
    void actionPerformed(ActionEvent e)
```

Die Methode `String getActionCommand()` der Klasse `java.awt.event.ActionEvent` liefert den Kommandonamen. Bei mehreren Buttons kann man hiermit feststellen, welcher Button gedrückt wurde.

Komponenten deaktivieren

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.
```

Programm 10.13 demonstriert die Funktionsweise eines Buttons.

Programm 10.13

```
import java.awt.Color;  
import java.awt.Container;  
import java.awt.FlowLayout;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.util.Random;  
  
import javax.swing.Icon;  
import javax.swing.ImageIcon;  
import javax.swing.JButton;  
import javax.swing.JFrame;  
  
public class ButtonTest1 extends JFrame implements ActionListener {  
    private Container c;  
    private JButton button;  
  
    public ButtonTest1() {  
        super("Button");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        c = getContentPane();  
        c.setLayout(new FlowLayout());  
  
        Icon icon = new ImageIcon(getClass().getResource("cat.gif"));  
        button = new JButton("Bitte klicken", icon);  
        button.setToolTipText("Test");  
        button.addActionListener(this);  
        c.add(button);  
  
        setSize(300, 200);  
        setVisible(true);  
    }
```

```

public void actionPerformed(ActionEvent e) {
    Random random = new Random();
    c.setBackground(new Color(random.nextInt(256), random.nextInt(256),
        random.nextInt(256)));
}

public static void main(String[] args) {
    new ButtonTest1();
}
}

```



Abbildung 10-11: JButton

Ausführbare jar-Datei

Die gesamte Anwendung (Klassen und weitere Ressourcen wie Bilder usw.) können so in einer *jar*-Datei zusammengefasst werden, dass das Programm wie folgt aufgerufen werden kann:

```
javaw -jar jar-Datei
```

Der Aufruf mit *javaw* unterdrückt das DOS-Fenster bei Windows.

Damit im Fall von Programm 10.13 das Bild *cat.gif* gefunden wird, muss es mit Hilfe der Class-Methode *getResource* bereitgestellt werden.⁸

Manifest

Zudem muss die *jar*-Datei eine so genannte *Manifest-Datei* enthalten. Hierzu ist eine Datei *manifest.txt* mit dem folgenden Inhalt anzulegen:

```
Main-Class: ButtonTest1
```

Diese Zeile muss mit einem Zeilenvorschub abgeschlossen sein. Evtl. Klassenpfade können durch Leerzeichen getrennt in einer weiteren Zeile angegeben sein:

⁸ Siehe Kapitel 5.6.

```
Class-Path: pfade1 pfade2 ...
```

Die *jar*-Datei kann nun wie folgt im bin-Verzeichnis erzeugt werden:

```
jar cfm test.jar manifest.txt *.class *.gif
```

Ein Doppelklick auf die Datei *test.jar* führt automatisch zum Aufruf von *javaw -jar test.jar*. Die Zuordnung vom Dateityp "jar" zum Programm "javaw -jar" wurde bereits bei der Java-Installation registriert.

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.

Alle Methoden der Klasse `AbstractButton` stehen zur Verfügung. Hier zwei weitere Methoden dieser Klasse:

```
void setSelected(boolean b)
    selektiert den Button, wenn b den Wert true hat.

boolean isSelected()
    liefert true, wenn der Button selektiert ist.
```

ButtonGroup

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 Deselektieren des bisher selektierten Buttons.

Ein Objekt der Klasse `ButtonGroup` wird mit dem Standardkonstruktor erzeugt.

```
void add(AbstractButton b)
    nimmt den Button b in die Gruppe auf.
```

Ereignisbehandlung: ActionEvent, ItemEvent

Neben der Erzeugung eines `ActionEvent` löst das Selektieren und Deselektieren eines `JCheckBox`- bzw. `JRadioButton`-Objekts ein `ItemEvent` aus.

Registrierungsmethode:

```
void addItemListener(ItemListener l)
```

`java.awt.event.ItemListener`-Methode:

```
void itemStateChanged(ItemEvent e)
```

Die `java.awt.event.ItemEvent`-Methode

```
int getStateChange()
```

liefert die `ItemEvent`-Konstante `SELECTED`, wenn ausgewählt wurde, oder `DESELECTED`, wenn die Auswahl aufgehoben wurde.

Programm 10.14

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;

public class ButtonTest2 extends JFrame implements ActionListener {
    private JCheckBox oval;
    private JRadioButton red;
    private JRadioButton green;
    private JRadioButton blue;

    public ButtonTest2() {
        super("Radiobuttons und eine Checkbox");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        red = new JRadioButton("rot", true);
        green = new JRadioButton("grün");
        blue = new JRadioButton("blau");

        red.addActionListener(this);
        green.addActionListener(this);
        blue.addActionListener(this);

        ButtonGroup bg = new ButtonGroup();
        bg.add(red);
        bg.add(green);
        bg.add(blue);

        oval = new JCheckBox("oval", true);
        oval.addActionListener(this);

        JPanel panel = new JPanel();
        panel.add(red);
        panel.add(green);
        panel.add(blue);
        panel.add(oval);

        add(panel, BorderLayout.NORTH);
        add(new MyPanel(), BorderLayout.CENTER);

        pack();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == red) {
            System.out.println("Rot gewählt");
        } else if (e.getSource() == green) {
            System.out.println("Grün gewählt");
        } else if (e.getSource() == blue) {
            System.out.println("Blau gewählt");
        } else if (e.getSource() == oval) {
            System.out.println("oval gewählt");
        }
    }
}
```

```
        setVisible(true);
    }

    private class MyPanel extends JPanel {
        public MyPanel() {
            setBackground(Color.white);
            setPreferredSize(new Dimension(400, 120));
        }

        public void paintComponent(Graphics g) {
            super.paintComponent(g);

            if (red.isSelected())
                g.setColor(Color.red);
            else if (green.isSelected())
                g.setColor(Color.green);
            else if (blue.isSelected())
                g.setColor(Color.blue);
            else
                g.setColor(Color.white);

            int w = getSize().width;
            int h = getSize().height;

            if (oval.isSelected())
                g.fillOval(10, 10, w - 20, h - 20);
            else
                g.fillRect(10, 10, w - 20, h - 20);
        }
    }

    public void actionPerformed(ActionEvent e) {
        repaint();
    }

    public static void main(String[] args) {
        new ButtonTest2();
    }
}
```

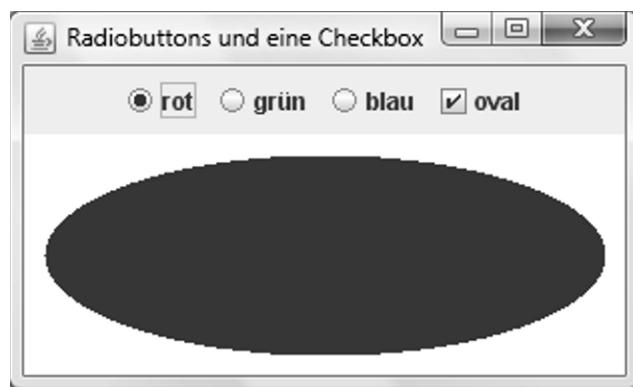


Abbildung 10-12: JCheckBox und JRadioButton

10.7 Labels

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`.

```
void setText(String text)
```

legt den Text des Labels fest.

```
String getText()
```

liefert den Text des Labels.

```
void setIcon(Icon icon)
```

legt das Icon fest.

```
Icon getIcon()
```

liefert das Icon.

```
void setIconTextGap(int gap)
```

legt den Abstand zwischen Icon und Text in Pixel fest.

```
int getIconTextGap()
```

liefert den Abstand zwischen Icon und Text in Pixel.

```
void setHorizontalAlignment(int align)
```

```
void setVerticalAlignment(int align)
```

legt die horizontale bzw. vertikale Ausrichtung des Inhalts auf der verfügbaren Fläche fest. Dabei werden für die horizontale Ausrichtung die Konstanten `LEFT`, `CENTER` oder `RIGHT` und für die vertikale Ausrichtung `TOP`, `CENTER` oder `BOTTOM` übergeben.

```
void setHorizontalTextPosition(int pos)
```

```
void setVerticalTextPosition(int pos)
```

legt die horizontale bzw. vertikale Position des Textes relativ zum Icon fest.

Dabei werden für die horizontale Positionierung die Konstanten `LEFT`, `CENTER` oder `RIGHT` und für die vertikale Positionierung `TOP`, `CENTER` oder `BOTTOM` übergeben.

Programm 10.15 zeigt, dass auch Texte mit HTML-Auszeichnung in einem Label verwendet werden können.

Programm 10.15

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Font;

import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class LabelTest extends JFrame {
    public LabelTest() {
        super("Label");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.setBackground(Color.white);

        Icon icon = new ImageIcon(getClass().getResource("javalogo.gif"));

        String text = "<html><div align='right'>Grundkurs<br/>Java</div></html>";

        JLabel label = new JLabel(text, icon, JLabel.CENTER);
        label.setIconTextGap(14);
        label.setForeground(Color.blue);
        label.setFont(new Font("SansSerif", Font.BOLD, 24));
        c.add(label);

        setSize(300, 200);
        setVisible(true);
    }

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



Abbildung 10-13: JLabel

10.8 Spezielle Container

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.

Konstruktoren:

```
JScrollPane(Component c)
JScrollPane(Component c, int v, int h)
```

`c` ist die darzustellende Komponente. Die Konstanten `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
VERTICAL_SCROLLBAR_ALWAYS
```

Gültige Werte für `h` sind die Konstanten:

```
HORIZONTAL_SCROLLBAR_AS_NEEDED (Voreinstellung)
HORIZONTAL_SCROLLBAR_NEVER
HORIZONTAL_SCROLLBAR_ALWAYS
```

Programm 10.16

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;

public class ScrollPaneTest extends JFrame {
    public ScrollPaneTest() {
        super("Scroll-Test");
        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 10-14: JScrollPane

JSplitPane

`javax.swing.JSplitPane` bietet die horizontale oder vertikale Anordnung von zwei Komponenten. Mit einem Trennbalken zwischen den beiden Komponenten kann geregelt werden, wie viel Platz für jede Komponente zur Verfügung stehen soll.

Konstruktor:

`JSplitPane(int orientation, boolean continuous, Component c1, Component c2)` erzeugt ein `JSplitPane`-Objekt für die Komponenten `c1` und `c2`. `orientation` legt die Ausrichtung des Trennbalkens fest: `JSplitPane.HORIZONTAL_SPLIT` oder `JSplitPane.VERTICAL_SPLIT`. Hat `continuous` den Wert `true`, werden die Komponenten fortlaufend neu gezeichnet, während der Trennbalken seine Position ändert.

`void setDividerLocation(double p)` positioniert den Trennbalken gemäß dem Prozentanteil ($0 \leq p \leq 1$) der `JSplitPane`-Größe.

Programm 10.17

```
import java.awt.Dimension;  
  
import javax.swing.ImageIcon;  
import javax.swing.JFrame;  
import javax.swing.JLabel;  
import javax.swing.JSplitPane;  
  
public class SplitPaneTest extends JFrame {  
    public SplitPaneTest() {  
        super("Split-Test");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

```

JLabel image1 = new JLabel(new ImageIcon(getClass().getResource(
    "tolstoi.jpg")));
JLabel image2 = new JLabel(new ImageIcon(getClass().getResource(
    "boat.png")));

// damit der Trennbalken frei bewegt werden kann
image1.setMinimumSize(new Dimension(0, 0));
image2.setMinimumSize(new Dimension(0, 0));

JSplitPane pane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, true,
    image1, image2);
add(pane);

setSize(400, 350);
setVisible(true);

// muss nach setVisible(true) stehen
pane.setDividerLocation(0.75);
}

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

```

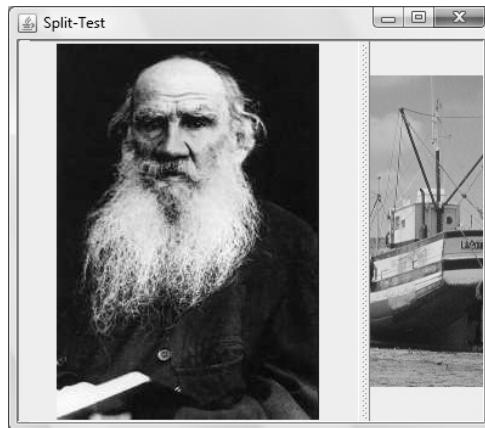


Abbildung 10-15: JSplitPane

JTabbedPane

`javax.swing.JTabbedPane` ist ein Container aus mehreren so genannten Karten (Tabs), die wie in einem Stapel übereinander gelegt und abwechselnd sichtbar gemacht werden können (siehe Abbildung 10-16).

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. Der Titel oder das Icon kann
    auch null sein.
```

```
void removeTabAt(int pos)
    löscht die Komponente an der Position pos.
```

Weitere Methoden sind:

```
void setTabLayoutPolicy(int policy)
    bestimmt die Darstellung, wenn nicht alle Registerlaschen im Fenster angezeigt
    werden können (siehe obigen Konstruktor).
```

```
void setSelectedIndex(int pos)
    selektiert die Karte an der Position pos.
```

```
int getSelectedIndex()
    liefert die Position der selektierten Karte.
```

```
void setSelectedComponent(Component c)
    selektiert die Karte mit der Komponente c.
```

```
void setTitleAt(int pos, String title)
    setzt den Titel der Karte an der Position pos.
```

```
String getTitleAt(int pos)
    liefert den Titel der Karte an der Position pos.
```

```
void setIconAt(int pos, Icon icon)
    setzt das Icon der Karte an der Position pos.
```

```
Icon getIconAt(int pos)
    liefert das Icon der Karte an der Position pos.
```

```
void setToolTipTextAt(int pos, String tip)
    setzt den Tooltip-Text der Karte an der Position pos.
```

```
void getToolTipTextAt(int pos)
    liefert den Tooltip-Text der Karte an der Position pos.
```

```
void setEnabledAt(int pos, boolean b)
    deaktiviert die Karte an der Position pos, wenn b den Wert false hat.
```

```
boolean isEnabledAt(int pos)
```

liefert den Wert true, wenn die Karte an der Position pos aktiviert ist.

Ereignisbehandlung: ChangeEvent

Die Selektion einer Karte löst ein javax.swing.event.ChangeEvent aus.

Registrierungsmethode:

```
void addChangeListener(ChangeListener l)
```

javax.swing.event.ChangeListener-Methode:

```
void stateChanged(ChangeEvent e)
```

Programm 10.18

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JTabbedPane;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class TabbedPaneTest extends JFrame implements ChangeListener {
    private JTabbedPane pane;

    public TabbedPaneTest() {
        super("Karten");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        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(this);
        add(pane);

        setSize(300, 200);
        setVisible(true);
    }

    public void stateChanged(ChangeEvent e) {
        System.out.println("Tab " + pane.getSelectedIndex());
    }

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



Abbildung 10-16: JTabbedPane

10.9 Textkomponenten

JTextComponent

Die Superklasse `javax.swing.text.JTextComponent` aller Textkomponenten bietet die folgenden Methoden:

`void setText(String text)`

setzt den Text `text`.

`String getText()`

liefert den Text des Textfeldes.

`void selectAll()`

markiert den kompletten Text im Textfeld.

`String getSelectedText()`

liefert den markierten Text.

`int getSelectionStart()`

liefert die Position des ersten markierten Zeichens.

`int getSelectionEnd()`

liefert die Position des letzten markierten Zeichens + 1.

`void setCaretPosition(int pos)`

setzt die Position der Einfügemarkie.

`int getCaretPosition()`

liefert die Position der Einfügemarkie.

`void moveCaretPosition(int pos)`

bewegt die Einfügemarkie 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 Einfügemarkie ein.

void setEditable(boolean b)
    setzt das Textfeld als editierbar, falls b den Wert true hat. Hat b den Wert false,
    so kann das Textfeld nicht editiert werden.

boolean isEditable()
    liefert true, wenn das Textfeld editiert werden kann, sonst false.

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 Eingabe-
strom. **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.

Konstruktoren:

```

JTextField()
JTextField(int cols)
JTextField(String text)
JTextField(String text, int cols)
    erzeugen jeweils ein Textfeld, das ggf. den Text text enthält und cols Zeichen
    breit ist.

```

```

void setColumns(int cols)
    setzt die Spaltenbreite des Textfeldes.

```

```

int getColumns()
    liefert die Spaltenbreite des Textfeldes.

```

```

void setHorizontalAlignment(int align)
    legt die horizontale Ausrichtung des Textes fest. Dabei werden die Konstanten
    LEFT, CENTER oder RIGHT übergeben.

```

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 '*'.

```
void setEchoChar(char c)
    setzt das Echo-Zeichen auf c.

char getEchoChar()
    liefert das eingestellte Echo-Zeichen.

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.
```

Ereignisbehandlung: ActionEvent

Wird die *Return-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.

Im Programm 10.19 wird beim Verlassen eines Feldes sein Inhalt geprüft. Das Feld "Artikel" darf nicht leer sein, das Feld "Preis" muss einen `double`-Wert enthalten. Der Eingabe-Fokus wechselt erst, wenn der Inhalt korrekt ist.

Zu diesem Zweck wird für Textfelder die `JComponent`-Methode

```
void setInputVerifier(InputVerifier verifier)
aufgerufen.
```

Die abstrakte Klasse `javax.swing.InputVerifier` enthält die Methode:

```
abstract boolean verify(JComponent input)
```

Diese Methode muss anwendungsbezogen überschrieben werden. `verify` liefert `true`, wenn die Eingabe korrekt ist, sonst `false`.

Programm 10.19

```
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.InputVerifier;
import javax.swing.JButton;
import javax.swing.JComponent;
```

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class TextTest1 extends JFrame implements ActionListener {
    private JTextField artikel, preis;
    private JLabel msg1, msg2;
    private JButton ok;

    public TextTest1() {
        super("Textfelder");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new GridLayout(7, 1, 0, 0));

        c.add(new JLabel("Artikel:"));
        artikel = new JTextField(30);
        artikel.setInputVerifier(new InputVerifier() {
            public boolean verify(JComponent input) {
                return isEmpty((JTextField) input);
            }
        });
        c.add(artikel);

        msg1 = new JLabel();
        msg1.setForeground(Color.red);
        c.add(msg1);

        c.add(new JLabel("Preis:"));
        preis = new JTextField(30);
        preis.setHorizontalAlignment(JTextField.RIGHT);
        preis.setInputVerifier(new InputVerifier() {
            public boolean verify(JComponent input) {
                return isDouble((JTextField) input);
            }
        });
        c.add(preis);

        msg2 = new JLabel();
        msg2.setForeground(Color.red);
        c.add(msg2);

        ok = new JButton("OK");
        ok.addActionListener(this);
        c.add(ok);

        pack();
        setVisible(true);
    }

    private boolean isEmpty(JTextField f) {
        String s = f.getText().trim();
        if (s.length() == 0) {
            msg1.setText("Bitte die Artikelbezeichnung eingeben.");
            return false;
        } else {
            msg1.setText("");
            return true;
        }
    }
}
```

```
private boolean isDouble(JTextField f) {
    String s = f.getText().trim();
    try {
        Double.parseDouble(s.trim());
        msg2.setText("");
        return true;
    } catch (NumberFormatException ex) {
        msg2.setText("Bitte eine Zahl eingeben.");
        return false;
    }
}

public void actionPerformed(ActionEvent e) {
    if (isNotEmpty(artikel) && isDouble(preis)) {
        System.out.println("Artikel: " + artikel.getText() + ", Preis: "
            + preis.getText());
        artikel.setText("");
        preis.setText("");
    }
}

public static void main(String[] args) {
    new TextTest1();
}
```



Abbildung 10-17: JTextField

JTextArea

Ein Objekt der Klasse `javax.swing.JTextArea` erlaubt die Eingabe mehrerer Textzeilen.

Konstruktoren:

```
JTextArea()
JTextArea(int rows, int cols)
JTextArea(String text)
JTextArea(String text, int rows, int cols)
```

erzeugen jeweils eine Textfläche, die ggf. den Text `text` enthält und `rows` sichtbare Zeilen sowie `cols` sichtbare Spalten hat, falls diese beiden Werte angegeben sind.

`void setRows(int rows)`

setzt die Anzahl Zeilen des Textbereichs.

`int getRows()`

liefert die Anzahl Zeilen des Textbereichs.

`void setColumns(int cols)`

setzt die Anzahl Spalten des Textbereichs.

`int getColumns()`

liefert die Anzahl Spalten des Textbereichs.

`void setLineWrap(boolean wrap)`

Hat `wrap` den Wert `true`, werden Zeilen, die zu lang sind, automatisch umbrochen.

`void setWrapStyleWord(boolean wrap)`

legt die Art des Zeilenumbruchs fest. Hat `wrap` den Wert `true`, werden Zeilen, die zu lang sind, am Ende eines Wortes umbrochen.

`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`.

Im Programm 10.20 können Zeilen mit Umbruch auf Wortgrenze eingegeben und gedruckt werden.

Eingabe-Fokus

Das Programm nutzt u. a. die `JComponent`-Methode

`boolean requestFocusInWindow(),`

die den *Fokus* für diese Komponente anfordert. Die Methode muss aufgerufen werden, bevor das Fenster sichtbar gemacht wird.

Programm 10.20

```
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.print.PrinterException;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class TextTest2 extends JFrame implements ActionListener {
    private JButton print;
    private JTextArea area;

    public TextTest2() {
        super("Textfläche");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel top = new JPanel();
        print = new JButton("Drucken");
        print.addActionListener(this);
        top.add(print);
        add(top, BorderLayout.NORTH);

        area = new JTextArea(10, 50);
        area.setWrapStyleWord(true);
        area.setLineWrap(true);
        area.setFont(new Font("DialogInput", Font.PLAIN, 18));
        JScrollPane pane = new JScrollPane(area,
            JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        add(pane, BorderLayout.CENTER);

        pack();
        area.requestFocusInWindow();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        Runnable printTask = new Runnable() {
            public void run() {
                try {
                    area.print();
                } catch (PrinterException ex) {
                    System.out.println(ex);
                }
            }
        };
        new Thread(printTask).start();
    }

    public static void main(String[] args) {
        TextTest2 t = new TextTest2();
        t.area.requestFocusInWindow();
    }
}
```

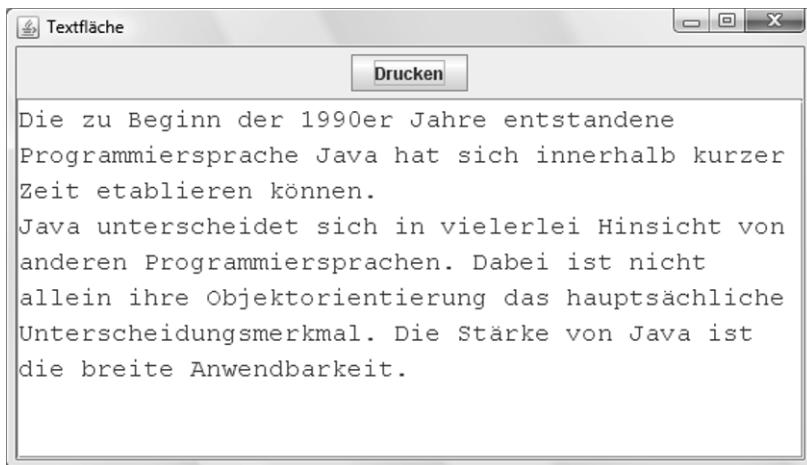


Abbildung 10-18: JTextArea

10.10 Auswahlkomponenten

Einzelige (`JComboBox`) und mehrzeilige Listenfelder (`JList`) sind Komponenten, die die Auswahl eines Eintrags aus einer Liste ermöglichen.

JComboBox

Die generische Klasse `javax.swing.JComboBox<T>` realisiert ein Feld, das durch Mausklick eine Liste aufklappt, aus der ein Eintrag ausgewählt werden kann.

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()

```

`T getItemAt(int pos)`
liefert den Eintrag an der Position pos.

`int getItemCount()`
liefert die Anzahl der Einträge.

```
int getSelectedIndex()
    liefert die Position des ausgewählten Eintrags oder -1, falls der Eintrag in der
    Liste nicht vorhanden ist.

Object getSelectedItem()
    liefert den ausgewählten Eintrag.

void setSelectedIndex(int pos)
    wählt den Eintrag an der Position pos aus.

void setSelectedItem(Object item)
    wählt den Eintrag item der Liste aus.

void setMaximumRowCount(int count)
    setzt die maximale Anzahl von sichtbaren Zeilen der Liste. Enthält die Liste
    mehr als count Zeilen, wird ein Scrollbalken erzeugt.

void setEditable(boolean b)
    Hat b den Wert true, so kann das Feld editiert werden und die Auswahl über
    Tastatur eingegeben werden.

ComboBoxEditor getEditor()
    liefert den Editor für das Feld. Mit der javax.swing.ComboBoxEditor-Methode
    Object getItem() kann dann die Eingabe abgefragt werden.
```

Ereignisbehandlung: ActionEvent, ItemEvent

Nach der Auswahl eines Eintrags wird ein `ActionEvent` ausgelöst. Beim Wechsel des selektierten Eintrags wird ein `ItemEvent` ausgelöst. Die entsprechenden Ereignisempfänger können registriert werden.⁹

Programm 10.21 präsentiert ein einzeiliges Listenfeld mit den Einträgen "rot", "gelb" und "grün". Bei Auswahl eines Eintrags wird der Hintergrund des Fensters in der ausgewählten Farbe dargestellt.

Programm 10.21

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JComboBox;
import javax.swing.JFrame;
```

⁹ Siehe Kapitel 10.6.

```

public class ComboBoxTest extends JFrame implements ActionListener {
    private JComboBox<String> auswahl;
    private String[] items = { "rot", "gelb", "grün" };
    private Color[] colors = { Color.red, Color.yellow, Color.green };
    private Container c;

    public ComboBoxTest() {
        super("Einzeiliges Listenfeld");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c = getContentPane();
        c.setLayout(new FlowLayout());

        auswahl = new JComboBox<String>(items);
        auswahl.addActionListener(this);
        auswahl.setSelectedIndex(0);
        c.add(auswahl);

        setSize(400, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        int i = auswahl.getSelectedIndex();
        c.setBackground(colors[i]);
    }
}

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

```



Abbildung 10-19: JComboBox

JList

Die generische 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.

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.

```
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 setVisibleRowCount(int count)
```

legt die Anzahl Zeilen der Liste fest, die ohne Scrollbalken angezeigt werden sollen.

```
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 setSelectionMode(int mode)
```

bestimmt, ob ein Eintrag (einfacher Mausklick) oder mehrere Einträge (Control-Taste und Mausklick bzw. Shift-Taste und Mausklick) ausgewählt werden können. Zulässige Werte von `mode` sind die `javax.swing.ListSelectionModel`-Konstanten: `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` und `MULTIPLE_INTERVAL_SELECTION` (Voreinstellung).

```
int getSelectionMode()
```

liefert den eingestellten Selektionsmodus.

```
void setSelectionBackground(Color c)
```

setzt die Hintergrundfarbe für ausgewählte Einträge.

```
void setSelectionForeground(Color c)
```

setzt die Vordergrundfarbe für ausgewählte Einträge.

```
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.

```
java.util.List<T> getSelectedValuesList()
```

liefert die ausgewählten Einträge.

```
int getSelectedIndex()
```

liefert die Position des ersten ausgewählten Eintrags oder `-1`, wenn kein Eintrag ausgewählt wurde.

```
int[] getSelectedIndices()
```

liefert die Positionen der ausgewählten Einträge.

```
void setSelectedIndex(int pos)
    wählt den Eintrag an der Position pos aus.

void setSelectedIndices(int[] pos)
    wählt die Einträge an den Positionen pos aus.
```

Programm 10.22

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;

public class ListTest extends JFrame implements ActionListener {
    private JButton ok;
    private JList<String> auswahl;
    private String[] items = { "Java", "C++", "C#", "C", "Delphi", "Ada",
        "Python", "Ruby", "PHP" };

    public ListTest() {
        super("Mehrzeiliges Listenfeld");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        auswahl = new JList<String>(items);
        auswahl.setVisibleRowCount(6);
        auswahl.setSelectionBackground(Color.yellow);
        auswahl.setSelectionForeground(Color.red);
        c.add(new JScrollPane(auswahl));

        ok = new JButton("OK");
        ok.addActionListener(this);
        c.add(ok);

        setSize(400, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (auswahl.isSelectionEmpty())
            return;
        int[] idx = auswahl.getSelectedIndices();
        for (int i = 0; i < idx.length; i++)
            System.out.println(items[idx[i]]);
        System.out.println();
        auswahl.clearSelection();
    }

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



Abbildung 10-20: JList

JSlider

`javax.swing.JSlider` ermöglicht die Auswahl einer ganzen Zahl aus einem Intervall mit Hilfe eines Schiebers.

Konstruktor:

`JSlider(int orientation, int min, int max, int value)`

orientation legt die Ausrichtung des Schiebers fest: `JSlider.HORIZONTAL` oder `JSlider.VERTICAL`. Das Intervall ist durch `min` und `max` begrenzt. `value` ist der Startwert.

`void setMajorTickSpacing(int n)`

`void setMinorTickSpacing(int n)`

definiert den Abstand zwischen großen und kleinen Strichen auf der Skala.

`void setPaintTicks(boolean b)`

legt fest, ob die Skalenstriche angezeigt werden sollen.

`void setPaintLabels(boolean b)`

legt fest, ob Beschriftungen angezeigt werden sollen.

`boolean getValueIsAdjusting()`

liefert true, wenn der Schieber bewegt wird.

`int getValue()`

liefert den eingestellten Wert.

`void setValue(int n)`

setzt den aktuellen Wert auf n.

Ereignisbehandlung: ChangeEvent

Das Bewegen des Schiebers löst ein ChangeEvent aus.¹⁰

Programm 10.23

```
import java.awt.Container;
import java.awt.FlowLayout;

import javax.swing.JFrame;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class SliderTest extends JFrame implements ChangeListener {
    private JSlider slider;

    public SliderTest() {
        super("Slider-Test");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 50);
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(10);
        slider.setPaintTicks(true);
        slider.setPaintLabels(true);
        slider.addChangeListener(this);
        c.add(slider);

        setSize(400, 200);
        setVisible(true);
    }

    public void stateChanged(ChangeEvent e) {
        if (!slider.getValueIsAdjusting())
            System.out.println(slider.getValue());
    }

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

¹⁰ Vgl. die Ausführungen zu JTabbedPane in Kapitel 10.8.

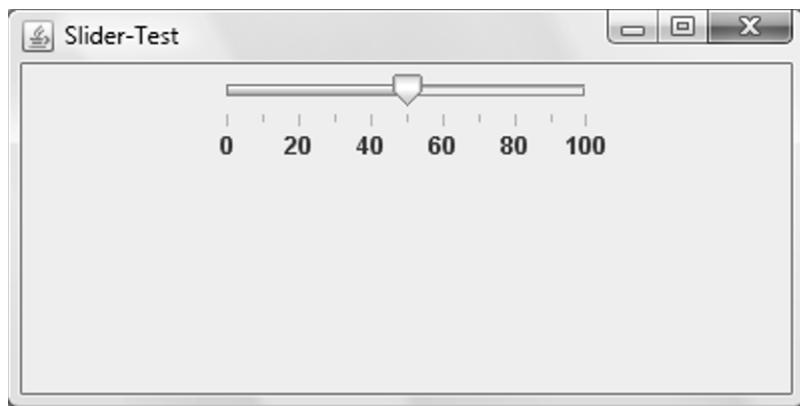


Abbildung 10-21: JSlider

10.11 Menüs und Symbolleisten

Fenster können *Menüs* enthalten. Ihre Erstellung 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 enthaltenes Menüs, die Klassen `JMenuItem`, `JCheckBoxMenuItem` und `JRadioButtonMenuItem` implementieren die Menüeinträge eines Menüs.

Die Klassen `JFrame`, `JDialog` und `JApplet` besitzen die Methode

```
void setJMenuBar(JMenuBar mb),
```

die die Menüleiste `mb` einbindet.

JMenuBar

`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

`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`.

```
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()`.

Hotkey

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` oder `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 `JCheckBoxMenuItem` analogen Konstruktoren. Mehrere dieser Menüeinträge können in einer Gruppe (Button-Group) zusammengefasst werden.¹¹

Ereignisbehandlung: ActionEvent

Wenn ein Menüeintrag ausgewählt wird, erzeugt dieser ein `ActionEvent`. Ein `ActionListener`-Objekt kann registriert werden.¹²

JToolBar

Symbolleisten werden von der Klasse `javax.swing.JToolBar` implementiert.

`JToolBar()`
`JToolBar(int orientation)`

erzeugen eine Symbolleiste mit horizontaler (`HORIZONTAL`) bzw. vertikaler (`VERTICAL`) Ausrichtung. Der parameterlose Konstruktor erzeugt eine horizontal ausgerichtete Leiste.

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.

¹¹ Siehe Kapitel 10.6.

¹² Siehe Kapitel 10.6.

Das folgende Programm zeigt ein Fenster mit Menüleiste und verschiebbbarer Symbolleiste.

Programm 10.24

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.InputEvent;
import java.awt.event.KeyEvent;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JToolBar;
import javax.swing.KeyStroke;

public class MenuTest extends JFrame implements ActionListener {
    public MenuTest() {
        super("Menü-Test");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        buildMenu();
        buildToolBar();

        setSize(300, 200);
        setVisible(true);
    }

    private void buildMenu() {
        JMenuBar bar = new JMenuBar();
        setJMenuBar(bar);

        JMenu datei = new JMenu("Datei");
        bar.add(datei);

        JMenuItem oeffnen = new JMenuItem("Öffnen");
        oeffnen.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
            InputEvent.CTRL_DOWN_MASK));
        oeffnen.setActionCommand("open");
        oeffnen.addActionListener(this);
        datei.add(oeffnen);

        JMenuItem speichern = new JMenuItem("Speichern");
        speichern.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
            InputEvent.CTRL_DOWN_MASK));
        speichern.setActionCommand("save");
        speichern.addActionListener(this);
        datei.add(speichern);

        datei.addSeparator();

        JMenuItem beenden = new JMenuItem("Beenden");
        beenden.setActionCommand("exit");
        beenden.addActionListener(this);
        datei.add(beenden);
    }
}
```

```
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.setActionCommand("cut");
    cut.addActionListener(this);
    bar.add(cut);

    JButton copy = new JButton(new ImageIcon(getClass().getResource(
        "copy.gif")));
    copy.setToolTipText("Kopieren");
    copy.setActionCommand("copy");
    copy.addActionListener(this);
    bar.add(copy);

    JButton paste = new JButton(new ImageIcon(getClass().getResource(
        "paste.gif")));
    paste.setToolTipText("Einfügen");
    paste.setActionCommand("paste");
    paste.addActionListener(this);
    bar.add(paste);

    JButton delete = new JButton(new ImageIcon(getClass().getResource(
        "delete.gif")));
    delete.setToolTipText("Löschen");
    delete.setActionCommand("delete");
    delete.addActionListener(this);
    bar.add(delete);
}

public void actionPerformed(ActionEvent e) {
    Object obj = e.getSource();
    String cmd = e.getActionCommand();
    if (obj instanceof JMenuItem) {
        System.out.println("Menue: " + cmd);
        if (cmd.equals("exit"))
            System.exit(0);
    } else if (obj instanceof JButton) {
        System.out.println("Toolbar: " + cmd);
    }
}

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

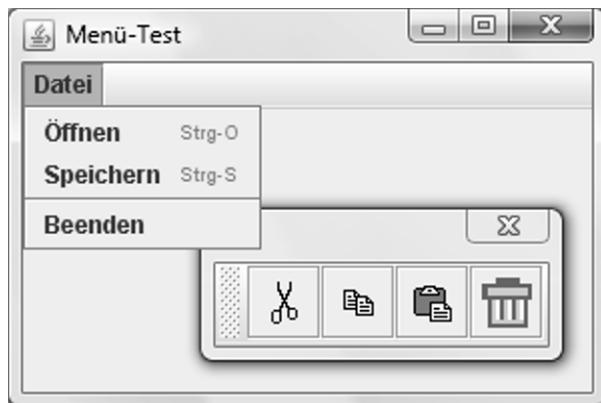


Abbildung 10-22: Menü und Symbolleiste

10.12 Mausaktionen und Kontextmenüs

Soll auf Mausaktionen, wie Drücken, Loslassen der Maustaste oder Bewegung des Mauszeigers, reagiert werden, so muss ein entsprechender Ereignisempfänger erstellt und registriert werden. Die Registrierung bei der Ereignisquelle erfolgt mit Methoden der Klasse `Component`.

Maustasten-Behandlung

Registrierungsmethode:

```
void addMouseListener(MouseListener l)
```

Der Empfänger kann dann auf Mausklicks sowie das Betreten und Verlassen der Komponente reagieren.

`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.

Die Adapterklasse `java.awt.event.MouseAdapter` implementiert das Interface `java.awt.eventMouseListener` mit leeren Methodenrumpfen.

MouseEvent

Methoden der Klasse `java.awt.event.MouseEvent` sind:

`Point getPoint()`

liefert die x- und y-Koordinate der Position des Mauszeigers als Objekt der Klasse `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.

`boolean isPopupTrigger()`

ermittelt, ob mit dem ausgelösten Maus-Ereignis ein *Kontextmenü* angezeigt werden kann. Da dies auf verschiedenen Plattformen unterschiedlich geregelt ist, sollte `isPopupTrigger` sowohl in der Methode `mousePressed` als auch `mouseReleased` aufgerufen werden. Bei Windows wird ein Kontextmenü beim Loslassen der gedrückten rechten Maustaste angezeigt.

Mausbewegungen

Registrierungsmethode:

```
void addMouseMotionListener(MouseMotionListener l)
```

Der Empfänger kann dann auf die Bewegung des Mauszeigers reagieren.

`java.awt.event.MouseMotionListener`-Methoden:

`void mouseMoved(MouseEvent e)`

wird aufgerufen, wenn die Maus bewegt wird, ohne dass dabei eine der Maustasten gedrückt wurde.

`void mouseDragged(MouseEvent e)`

wird aufgerufen, wenn die Maus bei gedrückter Maustaste bewegt wird.

Die Adapterklasse `java.awt.event.MouseMotionAdapter` implementiert das Interface `java.awt.event.MouseMotionListener` mit leeren Methodenrümpfen.

Das Interface `javax.swing.event.MouseInputListener` ist von den beiden Interfaces `MouseListener` und `MouseMotionListener` abgeleitet. Es enthält keine eigenen Methoden.

Die Adapterklasse `javax.swing.event.MouseInputAdapter` implementiert das Interface `MouseInputListener` mit leeren Methodenrümpfen.

Programm 10.25 demonstriert die Mausbehandlung. 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 (ein Objekt der Klasse `Vector`) der bereits gezeichneten Rechtecke eingetragen.

Bei jedem Aufruf von `paintComponent` bzw. `repaint` werden alle gespeicherten Rechtecke sowie das aktuelle Rechteck neu gezeichnet.

Programm 10.25

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.util.Vector;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class MausTest extends JFrame {
    public MausTest() {
        super("Maus-Test");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(new MyPanel());
        pack();
        setVisible(true);
    }

    private class MyPanel extends JPanel implements MouseListener,
        MouseMotionListener {

        // aktuelles, noch nicht gespeichertes Rechteck
        private Rect currentRect;

        // enthält alle komplett gezeichneten Rechtecke
        private Vector<Rect> rects = new Vector<Rect>();

        public MyPanel() {
            setBackground(Color.white);
            setPreferredSize(new Dimension(400, 300));
        }

        @Override
        public void mouseDown(MouseEvent e) {
            currentRect = new Rect(e.getPoint());
        }

        @Override
        public void mouseDragged(MouseEvent e) {
            currentRect.setHeight(e.getY() - currentRect.getTop());
            currentRect.setWidth(e.getX() - currentRect.getLeft());
        }

        @Override
        public void mouseUp(MouseEvent e) {
            rects.add(currentRect);
            currentRect = null;
        }

        @Override
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            for (Rect rect : rects) {
                g.drawRect(rect.getLeft(), rect.getTop(),
                    rect.getWidth(), rect.getHeight());
            }
        }
    }
}
```

```
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        // alle gespeicherten Rechtecke neu zeichnen
        for (Rect r : rects) {
            g.drawRect(r.x, r.y, r.b, r.h);
        }

        // aktuelles Rechteck zeichnen
        if (currentRect != null) {
            g.drawRect(currentRect.x, currentRect.y, currentRect.b,
                      currentRect.h);
        }
    }

    public void mousePressed(MouseEvent e) {
        // neues Rechteck erzeugen
        currentRect = new Rect(e.getX(), e.getY(), 0, 0);
    }

    public void mouseReleased(MouseEvent e) {
        // aktuelles Rechteck speichern
        if (currentRect.b > 0 && currentRect.h > 0)
            rects.add(currentRect);
    }

    public void mouseClicked(MouseEvent e) {
    }

    public void mouseEntered(MouseEvent e) {
    }

    public void mouseExited(MouseEvent e) {
    }

    public void mouseDragged(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();

        // Breite und Höhe des aktuellen Rechtecks ermitteln
        if (x > currentRect.x && y > currentRect.y) {
            currentRect.b = x - currentRect.x;
            currentRect.h = y - currentRect.y;
        }

        repaint();
    }

    public void mouseMoved(MouseEvent e) {
    }

    private class Rect {
        private int x, y, b, h;

        public Rect(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 MausTest();
}
```

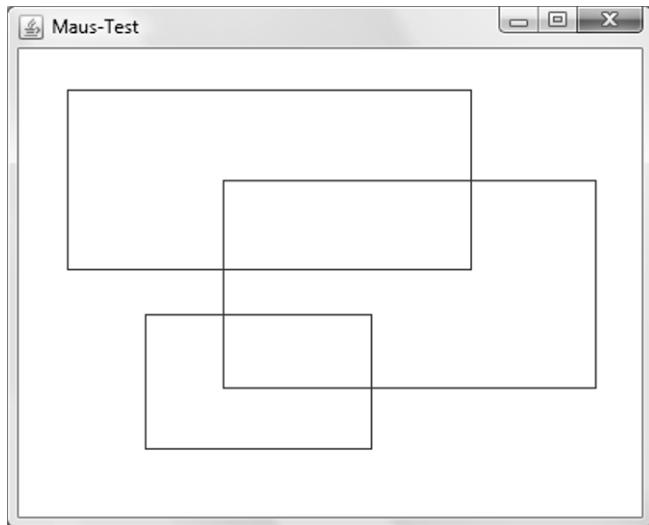


Abbildung 10-23: Rechtecke zeichnen

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()`.

`void show(Component c, int x, int y)`

zeigt das Kontextmenü in der Komponente `c`, die das Ereignis ausgelöst hat, an der Position `(x,y)` an.

Programm 10.26 demonstriert, wie ein Kontextmenü genutzt werden kann. Damit das Programm plattformunabhängig eingesetzt werden kann, sind die Implementierungen der Methoden `mousePressed` und `mouseReleased` identisch (siehe Erläuterungen zur `MouseEvent`-Methode `isPopupTrigger`).

Programm 10.26

```
import java.awt.Color;
import java.awt.Component;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JPopupMenu;

public class PopupMenuTest extends JFrame implements ActionListener,
    MouseListener {
    private JPopupMenu menu;

    public PopupMenuTest() {
        super("Kontextmenü-Test");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setBackground(Color.white);
        c.addMouseListener(this);

        buildMenu();

        setSize(400, 300);
        setVisible(true);
    }

    private void buildMenu() {
        menu = new JPopupMenu();

        JMenuItem cut = new JMenuItem("Ausschneiden", new ImageIcon(getClass()
            .getResource("cut.gif")));
        cut.setActionCommand("cut");
        cut.addActionListener(this);
        menu.add(cut);

        JMenuItem copy = new JMenuItem("Kopieren", new ImageIcon(getClass()
            .getResource("copy.gif")));
        copy.setActionCommand("copy");
        copy.addActionListener(this);
        menu.add(copy);

        JMenuItem paste = new JMenuItem("Einfügen", new ImageIcon(getClass()
            .getResource("paste.gif")));
        paste.setActionCommand("paste");
        paste.addActionListener(this);
        menu.add(paste);

        JMenuItem delete = new JMenuItem("Löschen", new ImageIcon(getClass()
            .getResource("delete.gif")));
        delete.setActionCommand("delete");
        delete.addActionListener(this);
        menu.add(delete);
    }
}
```

```
public void mousePressed(MouseEvent e) {  
    if (e.isPopupTrigger())  
        menu.show((Component) e.getSource(), e.getX(), e.getY());  
}  
  
public void mouseReleased(MouseEvent e) {  
    if (e.isPopupTrigger())  
        menu.show((Component) e.getSource(), e.getX(), e.getY());  
}  
  
public void mouseClicked(MouseEvent e) {}  
  
public void mouseEntered(MouseEvent e) {}  
  
public void mouseExited(MouseEvent e) {}  
  
public void actionPerformed(ActionEvent e) {  
    System.out.println(e.getActionCommand());  
}  
  
public static void main(String[] args) {  
    new PopupMenuTest();  
}  
}
```

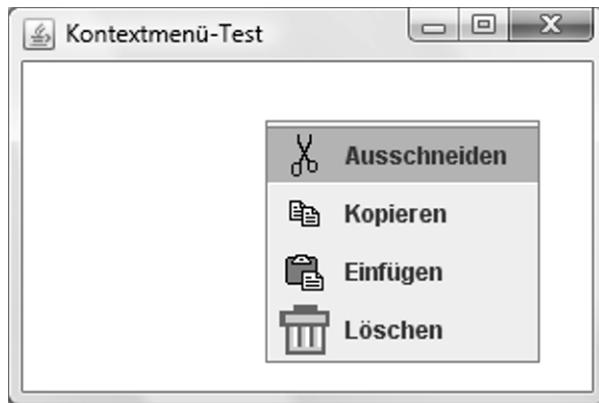


Abbildung 10-24: Ein Kontextmenü

10.13 Dialogfenster

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.

Konstruktoren:

```
JDialog(Typ owner)
JDialog(Typ owner, boolean modal)
JDialog(Typ owner, String title)
JDialog(Typ owner, String title, boolean modal)
```

erzeugen ein Dialogfenster ohne oder mit Titel. *owner* ist die übergeordnete Komponente. Für *Typ* ist entweder `java.awt.Frame` oder `java.awt.Dialog` zu lesen. 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.
```

Programm 10.27 erzeugt ein modales Dialogfenster, in dem ein Kennwort eingegeben werden kann.

Programm 10.27

```
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPasswordField;

public class DialogTest1 extends JFrame implements ActionListener {
    public DialogTest1() {
        super("Dialog-Test 1");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        JButton login = new JButton("Login");
```

```
login.addActionListener(this);
c.add(login);

setSize(300, 200);
setVisible(true);
}

public void actionPerformed(ActionEvent e) {
    LoginDialog dialog = new LoginDialog(this);

    // Wenn das Dialogfenster geschlossen wird, geht es hier weiter.
    if (dialog.isOK())
        System.out.println(dialog.getPassword());
}

private class LoginDialog extends JDialog implements ActionListener {
    private JPasswordField pw;
    private boolean ok;

    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("Kennwort: "));
        pw = new JPasswordField(15);
        c.add(pw);

        JButton ok = new JButton("OK");
        ok.addActionListener(this);
        c.add(ok);

        JButton abbr = new JButton("Abbrechen");
        abbr.addActionListener(this);
        c.add(abbr);

        setSize(250, 100);
        setResizable(false);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        ok = false;

        if (cmd.equals("OK")) {
            if (pw.getPassword().length == 0)
                return;
            ok = true;
        }

        dispose();
    }

    public boolean isOK() {
        return ok;
    }
}
```

```
public char[] getPassword() {  
    return pw.getPassword();  
}  
}  
  
public static void main(String[] args) {  
    new DialogTest1();  
}
```

Der Aufruf von `setVisible(true)` macht das modale Dialogfenster sichtbar und blockiert die Ausführung des aufrufenden Threads, bis der Dialog mit `dispose()` beendet wird.



Abbildung 10-25: 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).

Programm 10.28

```
import java.awt.Container;  
import java.awt.FlowLayout;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
  
import javax.swing.JButton;  
import javax.swing.JFrame;  
import javax.swing.JOptionPane;  
  
public class DialogTest2 extends JFrame implements ActionListener {  
    public DialogTest2() {  
        super("Dialog-Test 2");  
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);  
        Container c = getContentPane();  
        c.setLayout(new FlowLayout());  
  
        JButton ende = new JButton("Ende");  
        ende.addActionListener(this);  
        c.add(ende);  
        JButton ein = new JButton("Eingabe");  
        ein.addActionListener(this);  
        c.add(ein);  
        JButton msg = new JButton("Mitteilung");  
        msg.addActionListener(this);  
        c.add(msg);  
  
        setSize(300, 250);  
        setVisible(true);  
    }  
  
    public void actionPerformed(ActionEvent e) {  
        String cmd = e.getActionCommand();  
        if (cmd.equals("Ende")) {  
            int n = JOptionPane.showConfirmDialog(this,  
                "Anwendung wirklich beenden?", "Ende",  
                JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);  
        }  
    }  
}
```

```
if (n == JOptionPane.YES_OPTION)
    System.exit(0);
} else if (cmd.equals("Eingabe")) {
    String s = JOptionPane.showInputDialog(this,
        "Bitte Kontonummer eingaben:", "Eingabe",
        JOptionPane.PLAIN_MESSAGE);
    if (s != null)
        System.out.println(s);
} else if (cmd.equals("Mitteilung")) {
    JOptionPane.showMessageDialog(this, "Das ist ein Test!",
        "Mitteilung", JOptionPane.INFORMATION_MESSAGE);
}
}

public static void main(String[] args) {
    new DialogTest2();
}
```



Abbildung 10-26: JOptionPane

JFileChooser

Die Klasse `javax.swing.JFileChooser` implementiert einen Dialog zur Auswahl von Dateien im Dateisystem.

```
void setDialogTitle(String title)
String getDialogTitle()
    setzt bzw. liefert den Titel des Dialogfensters.

void setCurrentDirectory(File dir)
File getCurrentDirectory()
    setzt bzw. liefert das aktuelle Verzeichnis.
```

```

void setSelectedFile(File file)
File getFile()
    setzt bzw. liefert die ausgewählte Datei.

void setMultiSelectionEnabled(boolean b)
    bestimmt mit true, ob der Benutzer mehrere Dateien bzw. Verzeichnisse
    auswählen darf.

boolean isMultiSelectionEnabled()
    liefert true, wenn mehrere Dateien bzw. Verzeichnisse ausgewählt werden
    dürfen.

File[] getSelectedFiles()
    liefert die ausgewählten Dateien, falls mehrere Dateien ausgewählt werden
    dürfen.

void setSelectionMode(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.

```

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.

```

Programm 10.29 demonstriert einen Auswahldialog für Dateien mit der Endung ".java" bzw. ".txt". Die Inhalte werden in einem JTextArea-Objekt angezeigt.

Programm 10.29

```
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.filechooser.FileFilter;

public class Viewer extends JFrame implements ActionListener {
    private JTextArea text;

    public Viewer() {
        super("Viewer");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel top = new JPanel();
        JButton open = new JButton("Öffnen");
        open.addActionListener(this);
        top.add(open);
        add(top, BorderLayout.NORTH);

        text = new JTextArea(20, 80);
        text.setEditable(false);
        text.setFont(new Font("Monospaced", Font.PLAIN, 14));
        add(new JScrollPane(text), BorderLayout.CENTER);

        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        JFileChooser fc = new JFileChooser();
        fc.setCurrentDirectory(new File("."));
        fc.setFileFilter(new MyFileFilter());
        if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
            load(fc.getSelectedFile());
        }
    }

    private void load(File file) {
        try {
            text.read(new FileReader(file), null);
        } catch (IOException e) {
            text.setText(e.getMessage());
        }
    }

    private class MyFileFilter extends FileFilter {
        public boolean accept(File file) {
            if (file.isDirectory())
                return true;
```

```
String name = file.getName();
if (name.endsWith(".java"))
    return true;
else if (name.endsWith(".txt"))
    return true;
else
    return false;
}

public String getDescription() {
    return "Text file (*.java, *.txt)";
}
}

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

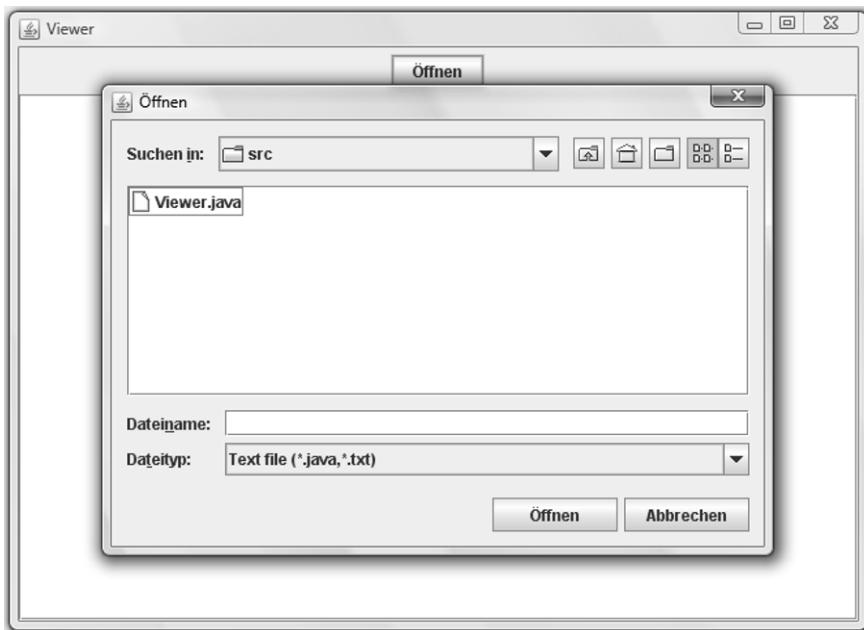


Abbildung 10-27: JFileChooser

10.14 Tabellen

Mit Hilfe der Klasse `javax.swing.JTable` können Daten tabellarisch angezeigt und einzelne Tabelleneinträge editiert werden.

Die Anzeigekomponente einer Tabelle ist von der Struktur und dem Inhalt der Daten, die angezeigt und geändert werden können, getrennt (MVC-Architektur).¹³

Aus der umfangreichen Sammlung von Methoden zur Erstellung und Präsentation von Tabellendaten zeigt dieses Kapitel nur eine Auswahl.

JTable

`JTable(TableModel m)`

erzeugt eine Tabelle auf der Basis des Datenmodells `m`.

Wird der Standardkonstruktor `JTable()` verwendet, so kann das Datenmodell mit der `JTable`-Methode

`void setModel(TableModel m)`

gesetzt werden.

Das Interface `javax.swing.table.TableModel` spezifiziert Methoden, die die `JTable`-Komponente für die Anzeige und Änderung der Daten nutzt.

AbstractTableModel

Die abstrakte Klasse `javax.swing.table.AbstractTableModel` implementiert dieses Interface 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.

Programm 10.30 zeigt, wie Daten in einer Tabelle angezeigt werden können.

¹³ Siehe Kapitel 10.1.

Programm 10.30

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;

public class TableTest1 extends JFrame {
    private JTable table;

    public TableTest1() {
        super("Tabellen-Test 1");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        table = new JTable(new MyTableModel());
        add(new JScrollPane(table));

        setSize(400, 200);
        setVisible(true);
    }

    private class MyTableModel extends AbstractTableModel {
        private static final int ROWS = 40;
        private static final int COLS = 6;
        private String[][] data = new String[ROWS][COLS];

        public MyTableModel() {
            for (int i = 0; i < ROWS; i++) {
                for (int j = 0; j < COLS; j++) {
                    data[i][j] = "Z " + i + ", S " + j;
                }
            }
        }

        public int getColumnCount() {
            return COLS;
        }

        public int getRowCount() {
            return ROWS;
        }

        public Object getValueAt(int row, int col) {
            return data[row][col];
        }
    }

    public static void main(String[] args) {
        new TableTest1();
    }
}
```

A	B	C	D	E	F
Z0,S0	Z0,S1	Z0,S2	Z0,S3	Z0,S4	Z0,S5
Z1,S0	Z1,S1	Z1,S2	Z1,S3	Z1,S4	Z1,S5
Z2,S0	Z2,S1	Z2,S2	Z2,S3	Z2,S4	Z2,S5
Z3,S0	Z3,S1	Z3,S2	Z3,S3	Z3,S4	Z3,S5
Z4,S0	Z4,S1	Z4,S2	Z4,S3	Z4,S4	Z4,S5
Z5,S0	Z5,S1	Z5,S2	Z5,S3	Z5,S4	Z5,S5
Z6,S0	Z6,S1	Z6,S2	Z6,S3	Z6,S4	Z6,S5
Z7,S0	Z7,S1	Z7,S2	Z7,S3	Z7,S4	Z7,S5

Abbildung 10-28: Eine einfache Tabelle

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.

Das `JTable`-Objekt 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 linksbündig ausgerichtet.

`boolean isCellEditable(int row, int col)`

liefert `true`, falls die Zelle editiert werden kann. Die Standardimplementierung liefert `false`.

`void setValueAt(Object value, int row, int col)`

setzt den Wert einer Zelle im Datenmodell. Diese Methode ist hier mit einem leeren Rumpf implementiert.

Ereignisbehandlung: `TableModelEvent`

Für die Benachrichtigung bei Änderung des Datenmodells wird ein Event vom Typ `javax.swing.event.TableModelEvent` erzeugt.

Registrierungsmethode:

```
void addTableModelListener(TableModelListener l)
```

`javax.swing.event.TableModelListener`-Methode:

```
void tableChanged(TableModelEvent e)
```

Das `JTable`-Objekt ist bereits standardmäßig als Listener mit seinem Datenmodell verbunden.

Die `AbstractTableModel`-Methode

```
void fireTableDataChanged()
```

benachrichtigt alle `TableModelListener`-Objekte darüber, dass Daten geändert wurden.

```
void fireTableStructureChanged()
```

benachrichtigt alle `TableModelListener`-Objekte darüber, dass sich die Tabellenstruktur (Anzahl, Überschriften oder Typen der Spalten) geändert hat.

Programm 10.31 stellte eine Tabelle mit den Spalten "Artikel", "Preis", "Menge" und "Einzelsumme" dar. Preise und Mengen können geändert werden. Einzelsummen und Gesamtsumme werden bei Änderung automatisch aktualisiert. Zur Anzeige der aktuellen Gesamtsumme wird `TableModelListener` implementiert.

Die Daten des Datenmodells werden hier der Einfachheit halber fest codiert und in einem zweidimensionalen Array gehalten. Die Daten könnten natürlich auch zu Beginn aus einer Datei geladen und in einem `Vector`-Objekt gespeichert werden.

Programm 10.31

```
import java.awt.BorderLayout;
import java.awt.Dimension;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.AbstractTableModel;

public class TableTest2 extends JFrame implements TableModelListener {
    private JTable table;
    private MyTableModel model;
    private JLabel gesamt;

    public TableTest2() {
        super("Tabellen-Test 2");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        model = new MyTableModel();
        model.addTableModelListener(this);

        table = new JTable(model);
        table.setPreferredScrollableViewportSize(new Dimension(400, 80));
        add(new JScrollPane(table), BorderLayout.CENTER);

        gesamt = new JLabel(model.update() + " ", JLabel.RIGHT);
        add(gesamt, BorderLayout.SOUTH);

        pack();
        setVisible(true);
    }
```

```
public void tableChanged(TableModelEvent e) {
    gesamt.setText(model.update() + " ");
}

private class MyTableModel extends AbstractTableModel {
    private String[] colNames = { "Artikel", "Preis", "Menge",
        "Einzelsumme" };
    private Object[][] data = { { "A4711", 100, 10, 1000 },
        { "A4721", 80, 5, 400 }, { "A4731", 10, 20, 200 },
        { "A4741", 12, 5, 60 }, { "A4751", 250, 4, 1000 } };

    public int getColumnCount() {
        return colNames.length;
    }

    public int getRowCount() {
        return data.length;
    }

    public Object getValueAt(int row, int col) {
        return data[row][col];
    }

    public String getColumnName(int col) {
        return colNames[col];
    }

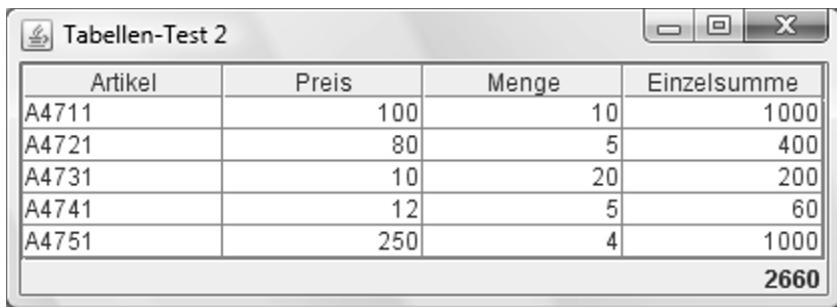
    public void setValueAt(Object value, int row, int col) {
        if (value == null)
            value = 0;
        data[row][col] = value;
        data[row][3] = (Integer) data[row][1] * (Integer) data[row][2];
        fireTableDataChanged();
    }

    public boolean isCellEditable(int row, int col) {
        return (col == 1 || col == 2) ? true : false;
    }

    public Class<?> getColumnClass(int col) {
        if (col == 0)
            return String.class;
        else
            return Integer.class;
    }

    public int update() {
        int sum = 0;
        for (int i = 0; i < data.length; i++) {
            sum += (Integer) data[i][3];
        }
        return sum;
    }
}

public static void main(String[] args) {
    new TableTest2();
}
```



The screenshot shows a Java Swing application window titled "Tabellen-Test 2". The window contains a table with five rows of data. The columns are labeled "Artikel", "Preis", "Menge", and "Einzelsumme". The data is as follows:

Artikel	Preis	Menge	Einzelsumme
A4711	100	10	1000
A4721	80	5	400
A4731	10	20	200
A4741	12	5	60
A4751	250	4	1000
			2660

Abbildung 10-29: Artikeltabelle

ListSelectionEvent

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)
```

`javax.swing.event.ListSelectionListener`-Methode:

```
void valueChanged(ListSelectionEvent e)
```

JTable-Methoden

`ListSelectionModel getSelectionModel()`

liefert das `ListSelectionModel`-Objekt. Dieses verfügt auch über die von `JList` her bekannte Methode `void setSelectionMode(int mode)`.¹⁴

`void setPreferredScrollableViewportSize(Dimension size)`

stellt die bevorzugte Größe des Anzeigebereichs ein.

`int getSelectedRow()`

liefert den Index der ersten selektierten Zeile oder -1, wenn keine Zeile selektiert wurde.

`int[] getSelectedRows()`

liefert die Indizes aller selektierten Zeilen oder ein leeres Array, wenn keine Zeile selektiert wurde.

`void setSelectionBackground(Color c)`

setzt die Hintergrundfarbe für ausgewählte Zellen.

`void setSelectionForeground(Color c)`

setzt die Vordergrundfarbe für ausgewählte Zellen.

¹⁴ Siehe Kapitel 10.10.

```
void setRowSelectionInterval(int idx0, int idx1)
    selektiert die Zeilen von idx0 bis idx1.

Object getValueAt(int row, int col)
    liefert den aktuellen Wert in Zeile row und Spalte col.

boolean print() throws java.awt.print.PrinterException
    öffnet einen Druckdialog zum Drucken der Tabelle. Der Rückgabewert ist
    false, falls der Druckvorgang vom Benutzer abgebrochen wurde.

void setAutoCreateRowSorter(boolean b)
    stellt ein, ob sortiert werden kann. Im Fall von true können die Zeilen der
    Tabelle sortiert werden, indem der Name einer Spalte angeklickt wird.
```

10.15 Aktualisierung der GUI-Oberfläche

Ereignisschlange und -Dispatcher

Für die Verarbeitung von Benutzereingaben (z. B. Mausklick auf einen Button) und die Aktualisierung der sichtbaren Komponenten eines Fensters ist ein eigener System-Thread, der so genannte *Ereignis-Dispatcher* verantwortlich. So werden vom Benutzer bzw. der Anwendung initiierte Ereignisse zunächst in eine *Ereignisschlange* (*Event Queue*) eingestellt. Die dort gesammelten Ereignisse werden dann vom Ereignis-Dispatcher der Reihen nach abgearbeitet. Beispielsweise werden die *JComponent*-Methode `paintComponent` und alle Listener-Methoden (z. B. `actionPerformed`) von diesem Dispatcher ausgeführt.

Die `java.awt.EventQueue`-Methode
 `static boolean isDispatchThread()`
liefert `true`, wenn der aufrufende Thread der Ereignis-Dispatcher ist.

Das folgende Beispiel soll mögliche Probleme bei der Aktualisierung von Ausgaben der Benutzeroberfläche demonstrieren.

Programm 10.32 zeigt in Abständen von 100 Millisekunden die aktuelle Systemzeit an. Diese Anzeige erfolgt in einem eigenen Anwendungs-Thread. Außerdem kann auf einen Button geklickt werden, dessen Beschriftung sich erst nach einer Verzögerung von einer Sekunde ändert. Mit dieser Verzögerung soll eine länger laufende Hintergrundverarbeitung simuliert werden. Während der Wartezeit wird die Zeitanzeige nicht aktualisiert. Die Oberfläche "friert ein".

Programm 10.32

```
import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Test1 extends JFrame implements Runnable, ActionListener {
    private JLabel label;
    private JButton button;
    private int count;

    public Test1() {
        super("Test 1");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        label = new JLabel(" ", JLabel.CENTER);
        add(label, BorderLayout.CENTER);

        button = new JButton("0");
        button.addActionListener(this);
        add(button, BorderLayout.SOUTH);

        setSize(300, 200);
        setVisible(true);

        Thread t = new Thread(this);
        t.start();
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("actionPerformed: "
            + Thread.currentThread().getName() + " "
            + EventQueue.isDispatchThread());

        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
        }

        button.setText(String.valueOf(++count));
    }

    public void run() {
        System.out.println("User-Thread: " + Thread.currentThread().getName()
            + " " + EventQueue.isDispatchThread());

        while (true) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
            }

            label.setText("Zeit: " + System.currentTimeMillis());
        }
    }
}
```

```
public static void main(String[] args) {
    new Test1();
}
```

In Programm 10.33 wird nun die Verzögerung um eine Sekunde vor der Aktualisierung der Buttonbeschriftung in einem Anwendungs-Thread realisiert und findet dann nicht mehr im Ereignis-Dispatcher statt.

Die Aktualisierung der GUI-Oberfläche (Zeitanzeige und Buttonbeschriftung) wird zudem komplett an den Ereignis-Dispatcher delegiert. Hierzu stehen die Klassenmethoden `invokeAndWait` und `invokeLater` der Klasse `EventQueue` zur Verfügung.

Die Ausführung des Programms zeigt, dass nunmehr keine Behinderung mehr erfolgt.

Programm 10.33

```
import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Test2 extends JFrame implements Runnable, ActionListener {

    private JLabel label;
    private JButton button;
    private int count;

    public Test2() {
        super("Test 2");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        label = new JLabel(" ", JLabel.CENTER);
        add(label, BorderLayout.CENTER);

        button = new JButton("0");
        button.addActionListener(this);
        add(button, BorderLayout.SOUTH);

        setSize(300, 200);
        setVisible(true);

        Thread t = new Thread(this);
        t.start();
    }

    public void actionPerformed(ActionEvent e) {
        final Runnable task = new Runnable() {
            public void run() {
                button.setText(String.valueOf(++count));
            }
        };
    }
}
```

```

Runnable runner = new Runnable() {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
        }

        EventQueue.invokeLater(task);
    }
};

Thread t = new Thread(runner);
t.setPriority(Thread.NORM_PRIORITY);
t.start();
}

public void run() {
    Runnable task = new Runnable() {
        public void run() {
            label.setText("Zeit: " + System.currentTimeMillis());
        }
    };

    while (true) {
        try {
            Thread.sleep(100);
            EventQueue.invokeAndWait(task);
        } catch (Exception e) {
        }
    }
}

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 Ereignis-Dispatcher ausgeführt, sobald noch nicht behandelte AWT- und Swing-Ereignisse verarbeitet sind. Die Methode `invokeAndWait` blockiert so lange, bis die `run`-Methode beendet ist. Wird der Wartevorgang unterbrochen, so wird die Ausnahme `InterruptedException` ausgelöst.

Löst die `run`-Methode eine Ausnahme im Dispatcher aus, die nicht abgefangen ist, so wird die Ausnahme `java.lang.reflect.InvocationTargetException` ausgelöst.

`invokeAndWait` darf nicht im Ereignis-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-Schlange des Dispatchers angefügt.

Fazit

Um Problemen bei der Aktualisierung der Oberfläche vorzubeugen, sollten Swing-Komponenten in der Regel nur über den Ereignis-Dispatcher manipuliert werden. Andere, länger laufende Aktionen sollten in Anwendungs-Threads ausgelagert werden.

Spätestens ab der Programmzeile mit dem Aufruf von `setVisible(true)` sollte die GUI-Aktualisierung so abgesichert werden.

Programm 10.34 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.

Konstruktor:

`JProgressBar(int min, int max)`

min und max geben den Start- und Endwert des Fortschritts an.

`void setValue(int n)`

setzt den aktuellen Fortschrittwert auf n.

`int getValue()`

liefert den aktuellen Fortschrittwert.

`void setStringPainted(boolean b)`

legt fest, ob der Fertigstellungsgrad (in Prozent) angezeigt werden soll.

Programm 10.34

```
import java.awt.Container;
import java.awt.EventQueue;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JProgressBar;

public class ProgressTest extends JFrame implements ActionListener {
    private JProgressBar pb;
    private JButton button;
    private JLabel msg;
```

public ProgressTest() {
 super("Progress-Test");
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```
Container c = getContentPane();
c.setLayout(new FlowLayout());

pb = new JProgressBar(0, 100);
pb.setStringPainted(true);
c.add(pb);

button = new JButton("Los geht's");
button.addActionListener(this);
c.add(button);

msg = new JLabel("");
msg.setFont(new Font("Monospaced", Font.BOLD, 18));
c.add(msg);

setSize(300, 200);
setVisible(true);
}

public void actionPerformed(ActionEvent e) {
    info(1);
    pb.setValue(0);
    msg.setText("");
    button.setEnabled(false);

    // Die Arbeit findet in einem eigenen Thread statt, der
    // nicht den Ereignis-Dispatcher behindert.
    Runnable longTask = new LongTask();
    Thread t = new Thread(longTask);

    // Ohne die folgende Anweisung würde Thread t die höhere
    // Priorität des Ereignis-Dispatchers erhalten.
    t.setPriority(Thread.NORM_PRIORITY);

    t.start();
}

private class LongTask implements Runnable {
    private int value;

    public void run() {
        info(2);

        Runnable task = new Runnable() {
            public void run() {
                if (value == 0)
                    info(3);
                pb.setValue(value);
            }
        };
        for (int i = 0; i <= 100; i++) {
            // Hier wird Arbeit simuliert.
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }

            value = i;
            EventQueue.invokeLater(task);
        }
    }
}
```

```
EventQueue.invokeLater(new Runnable() {
    public void run() {
        info(4);
        msg.setText("Fertig!");
        button.setEnabled(true);
    }
});

private static void info(int id) {
    System.out.println(id + " " + "Thread-Name: "
        + Thread.currentThread().getName() + ", isDispatchThread: "
        + EventQueue.isDispatchThread() + ", Priorität: "
        + Thread.currentThread().getPriority());
}

public static void main(String[] args) {
    info(0);
    new ProgressTest();
}
}
```

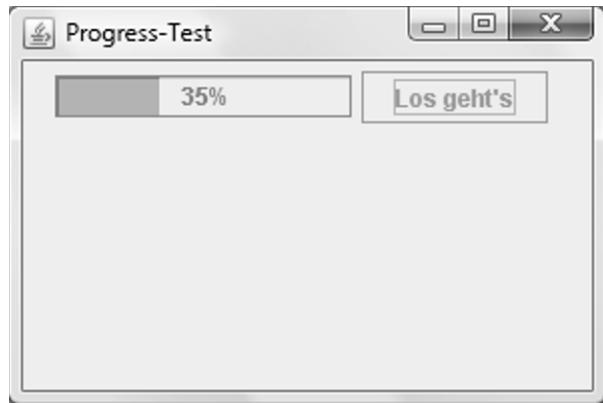
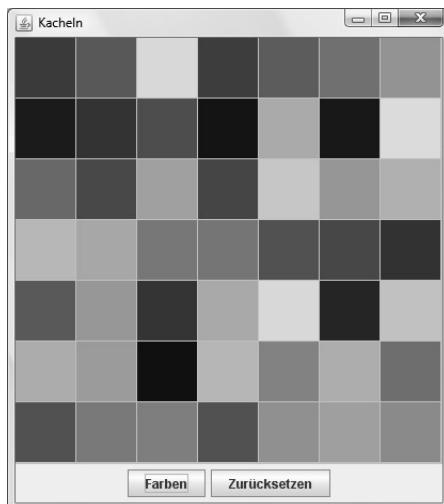


Abbildung 10-30: JProgressBar

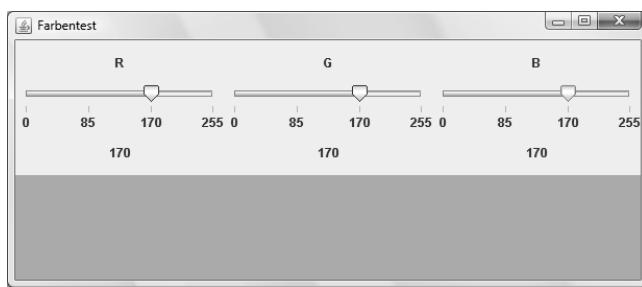
10.16 Aufgaben

1. Implementieren Sie für ein Fenster alle Methoden des Interfaces `WindowListener` und testen Sie durch entsprechende Konsolen-Ausgaben, wann welche Methode aufgerufen wird.
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.

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.
4. 7×7 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.



5. Erstellen Sie ein Programm, mit dem die drei Anteile einer Farbe nach dem RGB-Modell über JSlider-Komponenten eingestellt werden können.



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.



7. Durch Drücken der Maustaste soll ein Punkt an der Position des Mauszeigers in einem Panel gezeichnet werden. Speichern Sie die gezeichneten Punkte in einem Vektor. Die Methode `paintComponent` ist so zu implementieren, dass alle Punkte des Vektors gezeichnet werden.
8. Erweitern Sie die Funktionalität von Programm 10.25:

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 eines Markierungsfeldes (`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`).

Erweitern Sie das Programm (zweite Variante):

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`.¹⁵

9. Erstellen Sie einen Bildbetrachter zur Anzeige von Bildern im GIF-, JPEG- oder PNG-Format. Die Bilddateien sollen in einem Dialog ausgewählt werden können (`JFileChooser`). Mit Hilfe eines Filters sollen nur die geeigneten

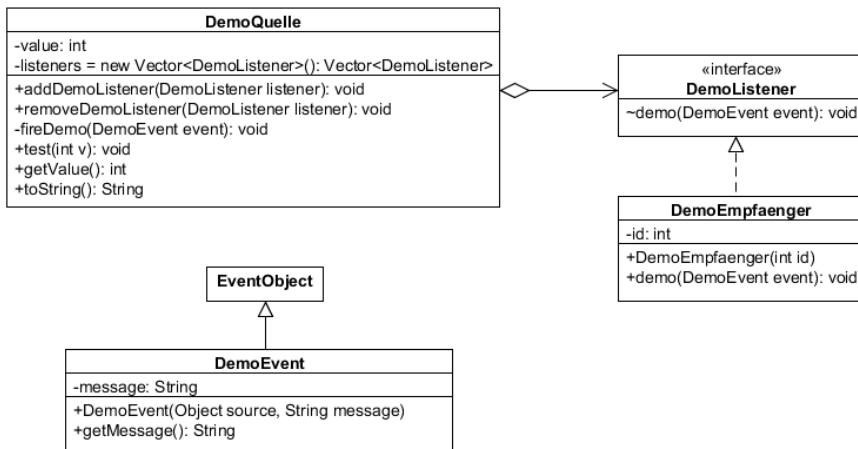
¹⁵ Siehe Aufgabe 19 aus Kapitel 8.

Dateien zur Auswahl angeboten werden. Tipp: Zeigen Sie das Bild als Icon innerhalb eines mit Scrollbalken ausgestatteten Labels an.

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.
11. Schreiben Sie ein Programm, dass eine Telefonliste (Name, Telefon) in einer Tabelle verwaltet. Bestehende Einträge können gelöscht, neue Einträge eingefügt werden. Eine Zeile kann um eine Position nach oben oder unten verschoben werden. Die Daten werden in einem Vektor gehalten. Beim Beenden des Programms werden die Elemente dieses Vektors in eine Datei geschrieben (Objektserialisierung). Beim Starten des Programms wird der Vektor aus dieser Datei geladen. Tipp: Speichern Sie einen einzelnen Eintrag bestehend aus Name und Telefonnummer in einem Objekt der Klasse `Telefon`, die die zugehörigen set- und get-Methoden implementiert.
12. In einem Fenster sollen der Reihe nach alle GIF-, JPEG- 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.
13. 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.
14. Nutzen Sie das `GridBagLayout`, um die Komponenten aus der folgenden Abbildung zu platzieren.



15. Es soll das Konzept des Event-Handlings mit eigenen Klassen veranschaulicht werden. Hierzu sind Klassen und Interface aus der folgenden Abbildung zu implementieren.



DemoEvent ist von java.util.EventObject abgeleitet. EventObject implementiert die Methoden Object getSource() und String toString(). In DemoQuelle werden registrierte DemoListener in einer Liste vom Typ Vector aufbewahrt. Das Ereignis DemoEvent wird von der Methode test durch den Aufruf von fireDemo ausgelöst. fireDemo ruft für alle registrierten Listener die DemoListener-Methode demo auf.

Testen Sie dieses Szenario.

16. Schreiben Sie ein Programm, das eine ToDo-Liste (JList) verwaltet.

Die Listeneinträge sollen beim Beenden des Programms automatisch 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.¹⁶

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.

¹⁶ Siehe Aufgabe 19 aus Kapitel 8.

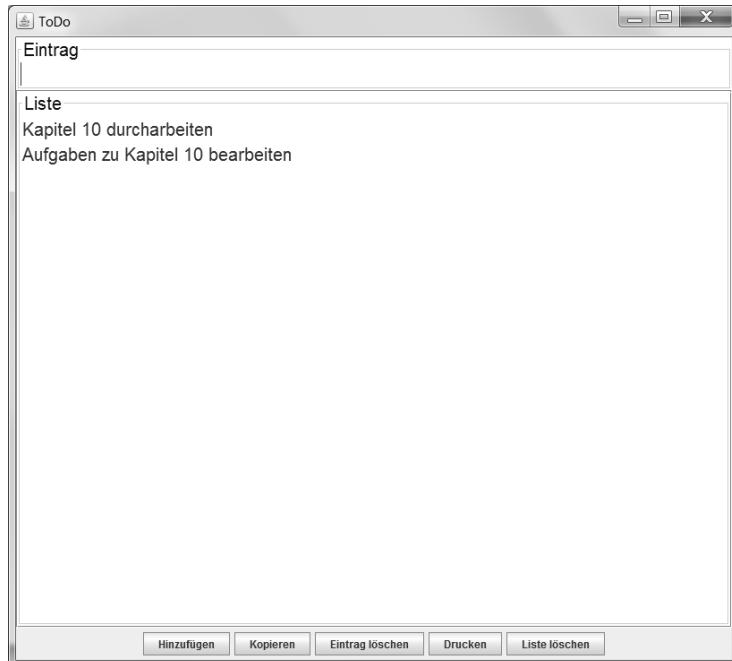
Drucken:

Die Liste wird gedruckt. Hierzu werden die Listeneinträge in eine `JTextArea` übernommen.¹⁷

Liste löschen:

Alle Einträge werden gelöscht.

Listeneinträge können mit dem Mauszeiger verschoben werden. Hierzu sind die Ereignismethoden `mousePressed` und `mouseDragged` geeignet zu implementieren.



¹⁷ Vgl. Programm 10.20.

11 Applets

Applets sind Java-Programme, die aus einer Webseite heraus aufgerufen werden. Dazu muss der Bytecode des Applets mit Hilfe von *HTML* (HyperText Markup Language) in Form eines Verweises eingebunden werden. Applets werden vom Java-Interpreter eines Browsers ausgeführt. Die Ausgabe erfolgt in einem rechteckigen Bereich des Browser-Fensters. Zum Testen kann alternativ das Programm *appletviewer* aus dem Java SE Development Kit (JDK) benutzt werden. *appletviewer* interpretiert nur die Applet-Einträge in der HTML-Datei. Alle anderen HTML-Anweisungen werden ignoriert. Das für die Ausführung im Browser benötigte *Plug-In* kann zusammen mit dem JDK installiert werden oder auch später nachinstalliert werden.

Lernziele

In diesem Kapitel lernen Sie

- wie Applets ausgeführt werden können,
- wie Anwendungen so geschrieben werden, dass sie sowohl als Applet als auch als eigenständige Applikation laufen können,
- wie mehrere Applets miteinander kommunizieren und über JavaScript gesteuert werden können,
- wie Bild- und Audiodaten eingebunden werden können,
- wie Zugriffsrechte für Applets explizit vergeben werden können.

Sicherheit

Aus Sicherheitsgründen unterliegen Applets einigen Einschränkungen:

- Applets können auf das Dateisystems des Rechners, auf dem der Browser läuft, nicht zugreifen.
- Applets können nur mit dem Server, von dem sie geladen wurden, über Netzwerk kommunizieren.
- Applets können auf dem Rechner, auf dem der Browser läuft, keine Prozesse starten.

Firefox verhindert aus Sicherheitsgründen die automatische Ausführung des Java-Plug-In. Um dennoch Applets im Browser aufrufen zu können, starten Sie das Java Control Panel (für Windows: *javacpl.exe*). Dieses Programm findet man im *bin*-Verzeichnis des für den Browser zuständigen Java-Runtime-Systems.

Wählen Sie dann den Dialog "Sicherheit". Erfassen Sie in der Liste der ausgenommenen Websites den Eintrag "<http://localhost>".

Applets können mit dem Programm *appletviewer* oder im Browser getestet werden. Beim Test mit dem Browser benutzen wir einen einfachen Webserver, der im Begleitmaterial zu diesem Buch (Online-Service) vorhanden ist.

Benötigt ein Applet weitere Ressourcen (z. B. Bilder oder Audio-Dateien), so fassen wir alles in einer jar-Datei zusammen.

11.1 Der Lebenszyklus eines Applets

Um ein Applet zu erstellen, muss eine Subklasse der Klasse `javax.swing.JApplet`, die einen speziellen Container mit *Content Pane* repräsentiert, erstellt werden. `JApplet` selbst ist Subklasse von `java.applet.Applet`.

Das folgende Beispiel zeigt ein einfaches Applet, das einen Text in seinem Anzeigebereich ausgibt.

Programm 11.1

```
import java.awt.Color;
import java.awt.Font;

import javax.swing.JApplet;
import javax.swing.JLabel;

public class Einfach extends JApplet {
    public void init() {
        System.out.println("init");
        JLabel label = new JLabel("Ein einfaches Applet", JLabel.CENTER);
        label.setForeground(Color.red);
        label.setFont(new Font("SansSerif", Font.BOLD, 24));
        add(label);
    }

    public void start() {
        System.out.println("start");
        super.start();
    }

    public void stop() {
        System.out.println("stop");
        super.stop();
    }

    public void destroy() {
        System.out.println("destroy");
        super.destroy();
    }
}
```

Die hierfür nötigen HTML-Anweisungen sind im Kapitel 11.2 beschrieben.

Einfach.html:

```
<html>
<head>
<title>Ein einfaches Applet</title>
</head>
<body>
<object type="application/x-java-applet" width="300" height="200">
<param name="code" value="Einfach" />
<param name="codebase" value="bin" />
</object>
</body>
</html>
```

Die HTML-Datei *Einfach.html* kann bei laufendem Webserver¹ von einem Browser (mit installiertem Plug-In) über

<http://localhost/P01/Einfach.html>

geladen werden oder mit dem JDK-Tool *appletviewer* getestet werden:

`appletviewer Einfach.html`



Abbildung 11-1: Anzeige mit appletviewer

Ein Applet ist kein eigenständiges Programm. Die Methode `main` fehlt. Zum Starten des Applets erzeugt der Browser bzw. das Programm *appletviewer* eine Instanz der abgeleiteten Klasse.

¹ Siehe Ausführungen zu Beginn dieses Kapitels.

Lebenszyklus

Das Applet durchläuft von dem Zeitpunkt, zu dem es geladen wurde, bis zum Zeitpunkt, zu dem kein Zugriff mehr möglich ist, einen bestimmten Zyklus. Die Übergänge zwischen den Lebensabschnitten werden durch vier Methoden der Klasse Applet realisiert, die einen leeren Rumpf besitzen, jedoch in der abgeleiteten Klasse bei Bedarf überschrieben werden können.

void init()

wird genau einmal aufgerufen, nachdem das Applet geladen wurde. Hier können Instanzvariablen initialisiert, Bilder und Audioclips geladen oder Parameter aus der HTML-Seite ausgewertet werden.

void start()

wird aufgerufen, nachdem die Initialisierung mit der Methode `init` abgeschlossen ist. Hier kann beispielsweise ein Thread gestartet werden. Im Gegensatz zur Initialisierung kann `start` vom Browser bzw. `appletviewer` mehrmals aufgerufen werden, z. B. nachdem das Applet gestoppt wurde.

void stop()

wird aufgerufen, wenn die HTML-Seite, in der das Applet eingebunden ist, verlassen wird. Ein gestopptes Applet kann durch `start` wieder zum Leben erweckt werden. `stop` kann also vom Browser bzw. `appletviewer` mehrmals aufgerufen werden.

void destroy()

wird aufgerufen, wenn ein Applet beendet wird (z. B. beim Beenden des Browsers). Hier können alle vom Applet in Anspruch genommenen Ressourcen freigegeben werden.

Beim Start eines Applets werden `init` und dann `start`, beim Beenden eines Applets `stop` und dann `destroy` aufgerufen. Das Verhalten der Browser kann unterschiedlich sein. So werden in der Regel bereits die Methoden `stop` und `destroy` ausgeführt, wenn die Seite, in der das Applet eingebunden ist, verlassen wird.

Die Ausgaben des obigen Programms erscheinen in der Java-Konsole des Java-Plug-In bzw. in der Konsole, in der der `appletviewer` gestartet wurde.

11.2 Die Appletumgebung

Applets lassen sich mit HTML-Tags in eine HTML-Seite einbinden:

```
<object Attribut="Wert" ...> ... </object>
```

Innerhalb des `Object`-Tags können mehrere Attribute auftreten. `type`, `width` und `height` müssen angegeben werden.

Zwischen `<object>` und `</object>` können mehrere *Parameter-Tags* stehen:

`<param name="Parameter" value="Wert">`
definiert einen Parameter, der an das Applet übergeben wird.

Tabelle 11-1: Attribute und Parameter des Object-Tags

Attribut	Beschreibung
type	Typ der Ressource, hier: <code>application/x-java-applet</code>
width	Breite des Anzeigebereichs in Pixel
height	Höhe des Anzeigebereichs in Pixel
name	Eindeutiger Name des Applets (zur Unterscheidung mehrerer, miteinander kommunizierender Applets)
Parameter	
code	Klassenname des auszuführenden Codes
codebase	URL des Verzeichnisses, das den Applet-Code enthält. Fehlt diese Angabe, wird das aktuelle Verzeichnis der HTML-Seite verwendet
archive	Ein mit <code>jar</code> erstelltes Archiv <code>xyz.jar</code> , das die Klassen der Applet-Anwendung und evtl. zusätzliche Ressourcen enthält

Einige Applet-Methoden:

`String getParameter(String name)`

liefert zum Parameter `name`, der im Parameter-Tag angegeben ist, den zugehörigen Wert. Wird der angegebene Parameter nicht gefunden, wird `null` zurückgegeben.

`String getAppletInfo()`

liefert einen String. Diese Methode sollte überschrieben werden und Informationen über das Applet (z. B. Name des Applets, Version, Datum, Autor, Copyright) enthalten.

`String[][] getParameterInfo()`

liefert in einem Array zu jedem Parameter den Namen, den Typ und die Beschreibung. Diese Methode sollte überschrieben werden.

`URL getDocumentBase()`

liefert den URL der HTML-Seite, in die das Applet eingebunden ist.²

`URL getCodeBase()`

liefert den URL des Verzeichnisses, das das Applet enthält.

² URL ist in Kapitel 13.1 erläutert.

Object-Tag in *Param.html*:

```
<object type="application/x-java-applet" width="300" height="200">
<param name="code" value="Param" />
<param name="codebase" value="bin" />
<param name="title" value="Parameter-Test" />
<param name="size" value="32" />
<param name="fontName" value="Monospaced" />
</object>
```

Programm 11.2

```
import java.awt.Color;
import java.awt.Font;

import javax.swing.JApplet;
import javax.swing.JLabel;

public class Param extends JApplet {
    public String getAppletInfo() {
        return "Param, (C) Abts, 2015";
    }

    public String[][] getParameterInfo() {
        return new String[][] { { "title", "String", "Titel" },
            { "size", "int", "Schriftgröße in Punkten" },
            { "fontName", "String", "Schriftart" } };
    }

    public void init() {
        String title = get("title", "Hier steht ein Text");
        int size = get("size", 12);
        String fontName = get("fontName", "SansSerif");

        JLabel label = new JLabel(title, JLabel.CENTER);
        label.setForeground(Color.red);
        label.setFont(new Font(fontName, Font.BOLD, size));
        add(label);
    }

    private String get(String key, String defaultValue) {
        String value = getParameter(key);
        return (value == null) ? defaultValue : value;
    }

    private int get(String key, int defaultValue) {
        String value = getParameter(key);
        try {
            return (value == null) ? defaultValue : Integer.parseInt(value);
        } catch (NumberFormatException e) {
            return defaultValue;
        }
    }
}
```

Der Menüpunkt *Informationen* des Programms *appletviewer* zeigt die Appletinformationen an:

Param, (C) Abts, 2015

```
title -- String -- Titel  
size -- int -- Schriftgröße in Punkten  
fontName -- String -- Schriftart
```

Kommunikation zwischen Applets und mit JavaScript

Die Methoden des Interfaces `java.applet.AppletContext` können genutzt werden, um Informationen über die Webseite zu beschaffen, in der das Applet eingebunden ist.

Die Applet-Methode

```
AppletContext getAppletContext()  
liefert den so genannten Applet-Kontext.
```

Methoden von `AppletContext` sind:

```
Applet getApplet(String name)  
liefert, sofern vorhanden, eine Referenz auf das Applet mit dem Namen name  
(Attribut name des Object-Tags) oder null.  
Enumeration<Applet> getApplets()  
liefert eine Aufzählung aller Applets der Webseite.
```

Programm 11.3 zeigt, wie ein Applet ein anderes Applet der gleichen Webseite steuern kann. Durch Drücken des Buttons in `Applet1` wird die Hintergrundfarbe im `Applet2` gewechselt.



Abbildung 11-2: Applet-Kommunikation

Programm 11.3

```
import java.applet.AppletContext;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;

public class Applet1 extends JApplet implements ActionListener {
    public void init() {
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        JLabel label = new JLabel("Applet 1");
        label.setFont(new Font("SansSerif", Font.BOLD, 24));
        c.add(label);

        JButton button = new JButton("Eine neue Farbe");
        button.addActionListener(this);
        c.add(button);
    }

    public void actionPerformed(ActionEvent e) {
        AppletContext ctx = getAppletContext();
        Applet2 a2 = (Applet2) ctx.getApplet("a2");
        Random random = new Random();
        a2.setColor(random.nextInt(256), random.nextInt(256),
                   random.nextInt(256));
    }
}

import java.awt.Color;
import java.awt.Font;

import javax.swing.JApplet;
import javax.swing.JLabel;

public class Applet2 extends JApplet {
    private JLabel label;

    public void init() {
        label = new JLabel("Applet 2", JLabel.CENTER);
        label.setFont(new Font("SansSerif", Font.BOLD, 24));
        label.setOpaque(true);
        add(label);
    }

    public void setColor(int r, int g, int b) {
        label.setBackground(new Color(r, g, b));
    }
}
```

Das Beispiel zeigt auch, dass in JavaScript Applet-Methoden aufgerufen werden können. *Komm.html* enthält eine JavaScript-Funktion und einen Button. Beim Anklicken des Button "Farbe setzen" wird diese Funktion aufgerufen. Über den Namen "a2" wird das Applet identifiziert, dessen Methode `setColor` dann ausgeführt wird.

Komm.html

```
<html>
<head>
<title>Kommunikation zwischen Applets und mit JavaScript</title>
<script type="text/javascript">
    function setColor() {
        var r = Math.floor(Math.random() * 256);
        var g = Math.floor(Math.random() * 256);
        var b = Math.floor(Math.random() * 256);
        document.a2.setColor(r, g, b);
    }
</script>
</head>
<body>
    <object type="application/x-java-applet" width="150" height="100">
        <param name="code" value="Applet1" />
        <param name="codebase" value="bin" />
    </object>

    <object type="application/x-java-applet" name="a2" width="150"
            height="100">
        <param name="code" value="Applet2" />
        <param name="codebase" value="bin" />
    </object>

    <form>
        <input type=button value="Farbe setzen" onClick="setColor();"/>
    </form>
</body>
</html>
```

11.3 Hybridanwendungen

Hybridanwendungen sind ausführbare Java-Programme, die als Applet in HTML-Seiten eingebunden und ohne Änderung auch als eigenständige Applikation gestartet werden können. Dazu muss die Klasse, die den Startcode der Anwendung enthält, von `JApplet` abgeleitet sein und sowohl die Methode `init` als auch `main` enthalten. Die Klasse `MyTabbedPane`, die die Oberfläche implementiert, ist von `JPanel` abgeleitet. `init` nimmt eine Instanz dieser Klasse in die *Content Pane* auf, `main` erzeugt einen Fensterrahmen und fügt eine Instanz in deren *Content Pane* ein. Programm 11.4 zeigt ein Beispiel.

Programm 11.4

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;

import javax.swing.ImageIcon;
import javax.swing.JApplet;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTabbedPane;

public class Bilder extends JApplet {
    public void init() {
        add(new MyTabbedPane());
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("Bilder");
        frame.add(new MyTabbedPane());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}

class MyTabbedPane extends JPanel {
    private JTabbedPane pane;

    public MyTabbedPane() {
        setLayout(new BorderLayout());
        setBackground(Color.white);
        setPreferredSize(new Dimension(300, 200));

        pane = new JTabbedPane(JTabbedPane.TOP);

        JLabel image1 = new JLabel(new ImageIcon(getClass().getResource(
            "tolstoi.jpg")));
        JLabel image2 = new JLabel(new ImageIcon(getClass().getResource(
            "duke.png")));
        JLabel image3 = new JLabel(new ImageIcon(getClass().getResource(
            "boat.png")));

        JScrollPane pane1 = new JScrollPane(image1);
        JScrollPane pane2 = new JScrollPane(image2);
        JScrollPane pane3 = new JScrollPane(image3);
        pane.addTab("Tolstoi", pane1);
        pane.addTab("Duke", pane2);
        pane.addTab("Boot", pane3);

        add(pane);
    }
}
```

Da zu dieser Anwendung mehrere Dateien (Bytecode und Bilder) gehören, fassen wir sie hier in einem Archiv mit Namen `bilder.jar` mittels

```
jar cf bilder.jar -C bin .
```

zusammen und nutzen den Parameter `archive` des Object-Tags:³

```
<object type="application/x-java-applet" width="300" height="200">
  <param name="code" value="Bilder" />
  <param name="archive" value="bilder.jar" />
</object>
```

Als Applikation kann das Programm mit

```
java -cp bilder.jar Bilder
```

aufgerufen werden.

Beim Aufruf im Browser wird nach Aufruf des Standardkonstruktors die Methode `init` der Klasse `Bilder` ausgeführt.

11.4 Wiedergabe von Bild- und Audiodaten

Bisher haben wir Bilder als Icons in speziellen GUI-Komponenten (wie `JButton` und `JLabel`) angezeigt. Bilder können aber auch mit `drawImage`-Methoden der Klasse `Graphics` gezeichnet werden.

Die Applet-Methoden

```
Image getImage(URL url)
Image getImage(URL url, String name)
```

liefern ein `java.awt.Image`-Objekt zur Darstellung eines Bildes im Grafikformat GIF, JPEG oder PNG am Bildschirm, laden aber nicht die Bilddaten. `url` ist der URL der Bilddatei bzw. der Basis-URL und `name` der zu `url` relative URL der Bilddatei.⁴

In Java-Applikationen werden entsprechende Methoden der Klasse `java.awt.Toolkit` verwendet:

```
Image getImage(URL url)
Image getImage(String filename)
```

Die Component-Methode `getToolkit()` liefert das `Toolkit`-Objekt für die aktuelle Umgebung.

³ Siehe Tabelle 11-1.

⁴ URL ist in Kapitel 13.1 erläutert.

Image

Objekte der abstrakten Klasse `java.awt.Image` repräsentieren Bilder.

```
int getHeight(ImageObserver observer)
int getWidth(ImageObserver observer)
```

liefern die Höhe bzw. Breite des Bildes oder -1, wenn diese noch nicht bekannt ist. `observer` vom Typ `java.awt.image.ImageObserver` überwacht den Ladezustand des Bildes. Da die Klasse `Component` das Interface `ImageObserver` implementiert, kann bei allen Komponenten `this` übergeben werden.

```
void flush()
gibt alle Ressourcen für dieses Bild frei.
```

Bilder anzeigen

Die folgenden `Graphics`-Methoden beginnen mit dem Zeichnen, auch wenn das gesamte Bild noch nicht komplett geladen wurde. In diesem Fall wird `false` zurückgegeben. Der Ladevorgang wird erst bei der Darstellung des Bildes ausgelöst, sodass oft noch unfertige Bilder angezeigt werden.

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
boolean drawImage(Image img, int x, int y, int width, int height,
                  ImageObserver observer)
zeichnen das Bild img mit der linken oberen Ecke an der Position (x,y). Im zweiten Fall wird das Bild noch auf die Größe width mal height skaliert.

boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1,
                  int sy1, int sx2, int sy2, ImageObserver observer)
zeichnet den rechteckigen Ausschnitt eines Bildes, der durch die linke obere Ecke (sx1,sy1) und die rechte untere Ecke (sx2,sy2) bestimmt ist, und skaliert ihn so, dass die linke obere Ecke die Position (dx1,dy1) und die rechte untere Ecke die Position (dx2,dy2) hat.
```

Vollständiges Laden von Bildern abwarten

Oft will man mit der Anzeige warten, bis das Bild vollständig geladen ist. Hierbei helfen die Methoden der Klasse `java.awt.MediaTracker`.

```
MediaTracker(Component c)
```

erzeugt eine `MediaTracker`-Instanz. `c` ist die Komponente, in der die Bilder gezeichnet werden sollen.

```
void addImage(Image img, int id)
```

fügt ein Bild `img` hinzu, dessen Ladevorgang überwacht werden soll. `id` bezeichnet die Priorität, mit der das Bild geladen wird. Niedrigere Werte bezeichnen eine höhere Priorität.

```
void waitForID(int id) throws InterruptedException
```

startet das Laden der Bilder, die die Priorität `id` haben, und kehrt erst zurück, wenn alle Bilder vorliegen oder beim Laden Fehler aufgetreten sind.

```
void waitForAll() throws InterruptedException
    startet das Laden aller Bilder und kehrt erst zurück, wenn alle Bilder vorliegen
    oder beim Laden Fehler aufgetreten sind.

boolean checkID(int id)
    liefert true, wenn die Bilder mit der Priorität id geladen wurden, sonst false.

boolean checkAll()
    liefert true, wenn alle Bilder geladen wurden, sonst false.

boolean isErrorAny()
    liefert true, wenn beim Laden eines Bildes ein Fehler aufgetreten ist, sonst
    false.

boolean isErrorID(int id)
    liefert true, wenn beim Laden eines Bildes mit der Priorität id ein Fehler
    aufgetreten ist, sonst false.
```

AudioClip

Das Interface `java.applet.AudioClip` deklariert drei Methoden zum Abspielen von Audioclips. Die Dateiformate AU, MIDI, WAVE und AIFF werden unterstützt.

```
void play()
    spielt den AudioClip einmal ab.

void loop()
    spielt den AudioClip in einer Schleife.

void stop()
    stoppt das Abspielen des Audioclips.
```

Die Applet-Methoden

```
AudioClip getAudioClip(URL url)
AudioClip getAudioClip(URL url, String name)
    erzeugen ein AudioClip-Objekt für Tondaten. Beim Aufruf einer Abspielmethode
    werden die Tondaten geladen. url ist der URL des Audioclips bzw. der
    Basis-URL und name der relative URL des Audioclips.
```

Audioclips können auch in Applikationen abgespielt werden.

Die Applet-Klassenmethode
 static AudioClip newAudioClip(URL url)
liefert ein AudioClip-Objekt.

Programm 11.5 demonstriert das Laden und die Wiedergabe von Bild und Ton. Die Namen der Medien-Dateien und die Bildtitel werden als Parameter aus der HTML-Seite gelesen.

Programm 11.5

```
import java.applet.AudioClip;
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;

import javax.swing.JApplet;
import javax.swing.JPanel;

public class Media extends JApplet {
    private AudioClip audioClip;
    private Image[] image = new Image[2];
    private String[] title = new String[2];
    private MediaTracker tracker;
    private Animation animation;
    private Thread thread;

    public void init() {
        audioClip = getAudioClip(getClass().getResource(
            getParameter("audioClip")));
        image[0] = getImage(getClass().getResource(getParameter("image1")));
        image[1] = getImage(getClass().getResource(getParameter("image2")));

        title[0] = getParameter("title1");
        title[1] = getParameter("title2");

        tracker = new MediaTracker(this);
        tracker.addImage(image[0], 0);
        tracker.addImage(image[1], 0);

        animation = new Animation();
        add(animation);
    }

    public void start() {
        if (thread == null) {
            thread = new Thread(animation);
            thread.start();
            audioClip.play();
        }
    }

    public void stop() {
        if (thread != null) {
            thread.interrupt();
            thread = null;
            audioClip.stop();
        }
    }

    private class Animation extends JPanel implements Runnable {
        private volatile int idx;

        public Animation() {
            setBackground(Color.white);
        }
    }
}
```

```
public void run() {
    try {
        tracker.waitForAll();
        if (tracker.isErrorAny())
            return;
    } catch (InterruptedException e) {
        return;
    }

    while (true) {
        repaint();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            return;
        }

        idx = ++idx % 2;
    }
}

protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    if (tracker.checkAll()) {
        int h = image[idx].getHeight(this);
        g.drawImage(image[idx], 10, 10, this);
        g.setFont(new Font("SansSerif", Font.BOLD, 12));
        g.drawString(title[idx], 10, h + 30);
    }
}
}
```

Bewegungen (Animation) in einem Applet oder einer Applikation sollten in einem eigenen Thread laufen, um den Browser nicht lahm zu legen. Somit kann das Programm auch noch auf Benutzereingaben reagieren.

Die Component-Methode `repaint` wird im Programm immer dann aufgerufen, wenn neu gezeichnet werden muss.

Double-Buffering

Ein mit dem Standardkonstruktor erzeugtes JPanel-Objekt nutzt das so genannte *Double-Buffering*: Es wird zunächst in einen *Offscreen*-Puffer gezeichnet, der anschließend, wenn alle Ausgabeoperationen abgeschlossen sind, auf den Bildschirm kopiert wird. Bei einer Animation vermeidet man so ein unschönes Flackern der Bildschirmanzeige.

Programm 11.6 implementiert ein animiertes Applet. Ein Auto (`Auto.gif`) bewegt sich schrittweise (`x += 2`) von links nach rechts über den Anzeigebereich des Applets.

Programm 11.6

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;

import javax.swing.JApplet;
import javax.swing.JPanel;

public class Auto extends JApplet {
    private Image image;
    private AutoPanel auto;
    private Thread thread;

    public void init() {
        image = getImage(getClass().getResource("Auto.gif"));
        auto = new AutoPanel();
        add(auto);
    }

    public void start() {
        if (thread == null) {
            thread = new Thread(auto);
            thread.start();
        }
    }

    public void stop() {
        if (thread != null) {
            thread.interrupt();
            thread = null;
        }
    }

    private class AutoPanel extends JPanel implements Runnable {
        private volatile int x = -111;

        public AutoPanel() {
            setBackground(Color.white);
        }

        public void run() {
            while (true) {
                if (x > 600)
                    x = -111;

                x += 2;

                repaint();

                try {
                    Thread.sleep(20);
                } catch (InterruptedException e) {
                    break;
                }
            }
        }

        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.drawLine(0, 126, 600, 126);
        }
    }
}
```

```
        g.drawImage(image, x, 80, 111, 46, this);
    }
}
```

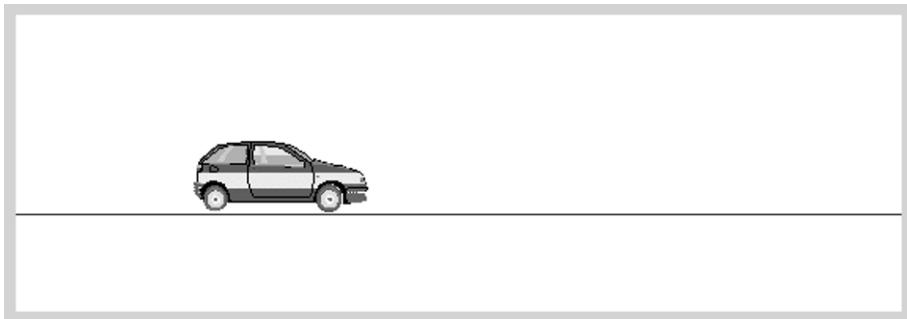


Abbildung 11-3: Ein animiertes Applet (Snapshot)

11.5 Zugriffsrechte für Applets

Das Laden von unbekanntem Java-Code über das Internet ist mit Sicherheitsrisiken verbunden. Zu Beginn dieses Kapitels wurde festgestellt, welchen Einschränkungen Applets aus Sicherheitsgründen unterliegen. In diesem Abschnitt wird gezeigt, wie solche Restriktionen gezielt gelockert werden können.

Programm 11.7 ist eine Anwendung, mit der Textdateien (mit Dateiendung .txt) gelesen und gespeichert werden können.

Programm 11.7

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Editor extends JApplet {
    public void init() {
        add(new MyPanel());
    }
}
```

```
public static void main(String[] args) {
    JFrame frame = new JFrame("Editor");
    frame.add(new MyPanel());
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
}

class MyPanel extends JPanel implements ActionListener {
    private JTextField file;
    private JButton open;
    private JButton save;
    private JTextArea text;

    public MyPanel() {
        setLayout(new BorderLayout());

        Font font = new Font("Monospaced", Font.PLAIN, 14);

        JPanel top = new JPanel();
        top.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 10));

        file = new JTextField(50);
        file.setFont(font);
        top.add(file);

        open = new JButton("Öffnen");
        open.addActionListener(this);
        top.add(open);

        save = new JButton("Speichern");
        save.addActionListener(this);
        top.add(save);

        add(top, BorderLayout.NORTH);

        text = new JTextArea(20, 80);
        text.setFont(font);
        add(new JScrollPane(text), BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e) {
        String filename = file.getText();
        if (!filename.endsWith(".txt"))
            return;

        Object obj = e.getSource();

        if (obj == open) {
            try {
                text.read(new FileReader(filename), null);
            } catch (IOException e1) {
                text.setText(e1.getMessage());
            }
        } else {
            try {
                text.write(new FileWriter(filename));
                text.setText("Datei wurde gespeichert");
            } catch (IOException e1) {
```

```
        text.setText(e1.getMessage());
    }
}
}
```

Wird die zugehörige HTML-Datei, in der das Applet eingebunden ist, geladen, so werden ohne weitere Vorkehrungen Dateizugriffe auf dem Rechner, auf dem der Browser läuft, unterbunden. Bei eingeblendeter Java-Konsole des Java Plug-In können die Fehlermeldungen mitverfolgt werden (siehe Abbildung 11-4).

Um den Zugriff auf alle Dateien eines bestimmten Verzeichnisses zu gewähren, muss eine so genannte Policy-Datei mit dem Namen *.java.policy* erstellt werden. Diese muss im Home-Verzeichnis des Users (siehe System-Eigenschaft: *user.home*) abgelegt sein.

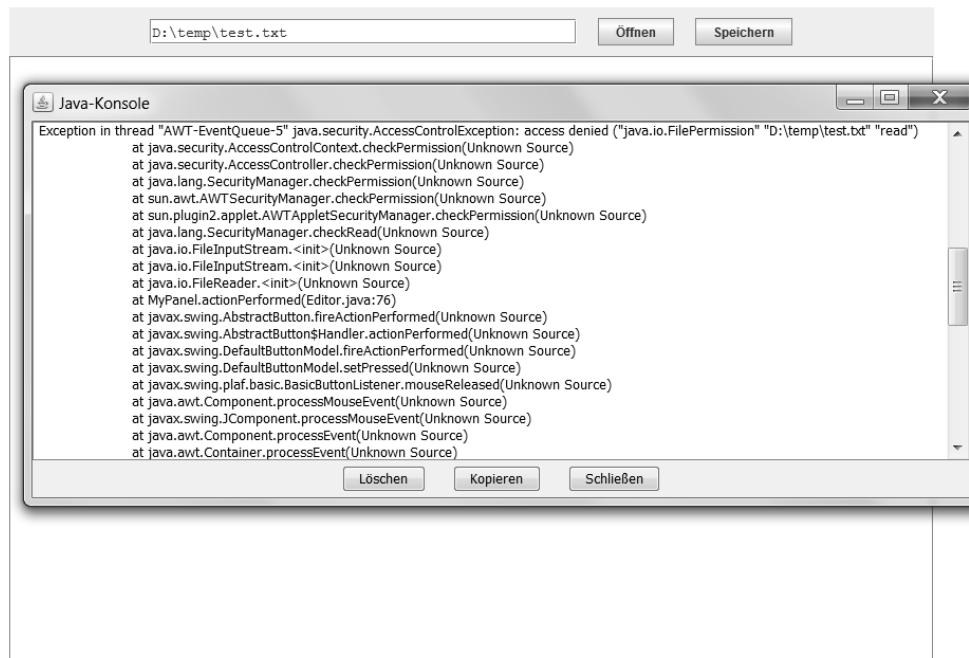


Abbildung 11-4: Die Java-Konsole zeigt Fehler an

Beispiel *.java.policy*:

```
grant codeBase "http://localhost/-" {
    permission java.io.FilePermission "D:/temp/*", "read, write";
};
```

Mit dem `codeBase`-Eintrag kann man die spezifizierten Rechte nur für den Code vergeben, der von einer bestimmten Stelle aus geladen wurde. `codeBase` gibt das Verzeichnis an, in dem der Java-Code des Applets liegt. Zugriffsrechte werden mit `permission`-Einträgen vergeben.

Für alle Dateien im Verzeichnis `D:\temp` wird hier das Lese- und Schreibrecht vergeben.

Es gibt eine Reihe anderer Typen von Zugriffsrechten. Details kann man in der Dokumentation zum JDK (unter "Security") nachlesen. In der Testphase kann auch die Java-Konsole des Plug-In genutzt werden. Die Fehlermeldungen geben Aufschluss darüber, welche Zugriffsrechte fehlen.

Das Java-Plug-In kann die Verwendung der Policy-Datei unterbinden. Sämtliche Deployment-Eigenschaften können in der Java-Konsole (Eingabe: `s`) abgefragt werden. Hier muss

```
deployment.security.use.user.home.java.policy = true
```

gesetzt sein.

Ist das nicht der Fall, kann diese Eigenschaft in die Datei `deployment.properties` eingetragen werden.

Diese Datei findet man bei Windows 7 im Verzeichnis

```
C:\Users\<username>\AppData\LocalLow\Sun\Java\Deployment5
```

11.6 Aufgaben

1. Schreiben Sie ein Applet, das eine Ampel anzeigt, die automatisch von Rot über Gelb auf Grün und wieder zurück umschaltet. Dabei sollen die Rot- und Grünphase 10 Sekunden und die Gelbphase 3 Sekunden dauern.
2. Schreiben Sie ein Applet, das farbige, kreisförmige Flecken zeichnet. Position, Größe des Radius und Farbe sollen zufällig sein. Ebenso sollen die Zeitabstände zwischen den Zeichenvorgängen zufällig verteilt sein. Bereits gezeichnete Flecken sollen sichtbar bleiben. Tipp: Lassen Sie den Aufruf von `super.paintComponent(g)` in `paintComponent` weg.
3. Schreiben Sie Programm 10.25 aus Kapitel 10 als Hybridanwendung, die sowohl als Applet als auch als Applikation laufen kann.
4. Ausgehend von einer Startposition soll ein Ball (ausgefüllter Kreis) sich diagonal über den Applet-Anzegebereich bewegen. Trifft der Ball den Rand des Bereichs, muss die Bewegungsrichtung so geändert werden, dass er wie ein Gummiball abprallt (*Bouncing Ball*).

⁵ Siehe <http://docs.oracle.com/javase/8/docs/technotes/guides/jweb/jcp/properties.html>

5. Schreiben Sie ein Applet, das auf einer schwarzen Fläche von Zeit zu Zeit Blitze vom oberen bis zum unteren Rand des Anzeigebereichs zucken lässt. Position und Form der Blitze sind mit Hilfe eines Zufallsgenerators festzulegen.
6. In einem Applet soll eine Folge von Bildern (z. B. *T1.gif* bis *T14.gif*) mit Ton abgespielt werden. Als Parameter benötigt das Applet: den Namensstamm der Bilddateien (z. B. *T*), die Erweiterung (z. B. *.gif*), die Nummer des ersten Bildes, die Nummer des letzten Bildes, die pro Sekunde anzuzeigende Anzahl Bilder und den Namen des Audioclips.

12 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. Primärschlüssele werden benutzt, um Beziehungen zwischen Tabellen herzustellen. Für eine umfassende Darstellung verweisen wir auf die am Ende des Buches aufgeführte Literatur.

JDBC

Das Paket `java.sql` bietet eine Programmierschnittstelle für den Zugriff auf relationale Datenbanken mit Hilfe der Standard-Datenbanksprache SQL (Structured Query Language). 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 Datenbanksystem geschrieben werden. Somit ist ein Wechsel des Datenbanksystems ohne Änderung des Java-Programms möglich.

Lernziele

In diesem Kapitel lernen Sie

- was JDBC ist,
- wie eine Verbindung zu einer relationalen Datenbank hergestellt wird,
- wie Daten über JDBC ausgewertet und geändert werden können.

12.1 Konfiguration und Verbindungsaufbau

Die Datenbankanbindung wird über einen datenbanksystemspezifischen *JDBC-Treiber* realisiert. Dieser Treiber versteht die JDBC-Aufrufe, übersetzt sie in datenbanksystemspezifische Befehle und leitet sie an das Datenbanksystem zur Ausführung weiter.

In den folgenden Anwendungsbeispielen werden die relationalen Datenbanksysteme *Microsoft Access* und *H2 Database*¹ eingesetzt. Selbstverständlich können auch andere Systeme wie z. B. Apache Derby und MySQL genutzt werden.

Die Open-Source-Produkte *H2* und *Derby* sind vollständig in Java realisiert. Sie können als Einzelplatzversion im eingebetteten Modus (*embedded*) oder als Server-Datenbank im Mehrbenutzerbetrieb eingesetzt werden. Im eingebetteten Modus laufen Java-Anwendung und Datenbanksystem in derselben virtuellen Maschine (JVM).

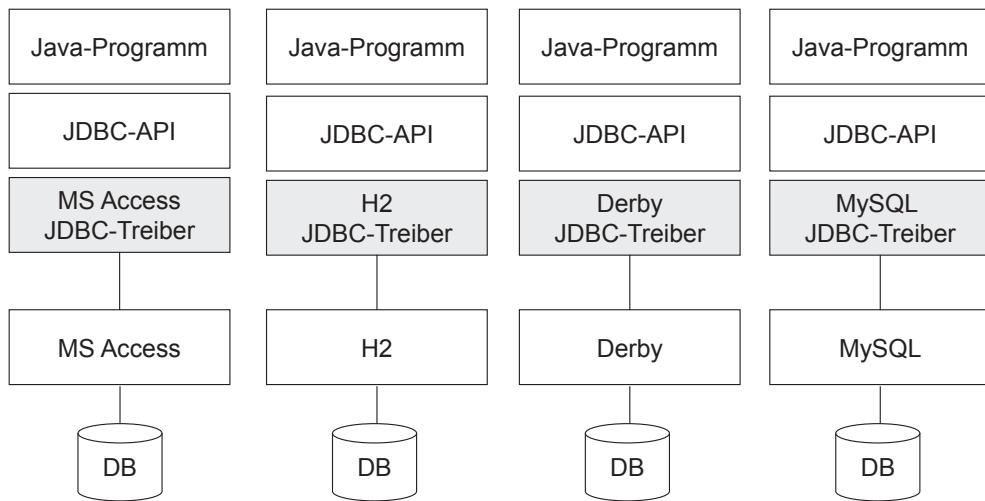


Abbildung 12-1: Passende Treiber für die jeweiligen Datenbanksysteme

Microsoft Access

Wir nutzen hier den JDBC-Treiber *UCanAccess*.² Die Datei *ucanaccess-x.x.x.jar* und die jar-Dateien aus dem lib-Verzeichnis der entpackten ZIP-Datei müssen in den CLASSPATH der JDBC-Programme aufgenommen werden.

H2

Zur Nutzung der H2-Datenbank im eingebetteten oder im Client/Server-Modus muss die jar-Datei *h2-x.x.x.jar* aus <*H2-Verzeichnis*>\bin in den CLASSPATH eingebunden werden. Der Online-Service enthält Skripte zum Starten und Herunterfahren des Servers.

¹ Siehe <http://www.h2database.com>

² Siehe <http://ucanaccess.sourceforge.net/site.html>

MySQL

Für die MySQL-Datenbank muss der JDBC-Treiber *MySQL Connector/J* (eine jar-Datei) in den Klassenpfad CLASSPATH eingebunden werden.

Derby

Wird Derby im eingebetteten Modus betrieben, muss die jar-Datei *derby.jar* aus dem Installationsverzeichnis (z. B. <JDK-Verzeichnis>\db\lib) in den CLASSPATH aufgenommen werden.

Wird Derby im Client/Server-Modus betrieben, ist die jar-Datei *derbyclient.jar* in den CLASSPATH aufzunehmen.

Die Bezugsquellen der Datenbanksysteme und Treiber sind am Ende des Buches aufgeführt.

Alle im Folgenden beschriebenen Methoden lösen die Ausnahme `java.sql.SQLException` aus, wenn beim Zugriff auf die Datenbank ein Fehler aufgetreten ist.

Verbindung herstellen

Die Methode

```
static Connection getConnection(String url, String user, String
password) throws SQLException
```

der Klasse `java.sql.DriverManager` stellt eine Verbindung zur Datenbank her und liefert ein Objekt vom Typ des Interfaces `java.sql.Connection`. `url` ist der URL (*Uniform Resource Locator*) der Datenbank. Tabelle 12-1 enthält die URL-Syntax für die verschiedenen Datenbanksysteme und Betriebsarten.

Tabelle 12-1: Datenbank-URL

Datenbanksystem	Modus	URL
MS Access		<code>jdbc:ucanaccess://pfad/dbname.accdb</code>
H2	embedded	<code>jdbc:h2:pfad/dbname</code>
H2	Client/Server	<code>jdbc:h2:tcp://localhost/dbname</code>
MySQL	Client/Server	<code>jdbc:mysql://localhost/dbname</code>
Derby	embedded	<code>jdbc:derby:pfad/dbname;create=true</code>
Derby	Client/Server	<code>jdbc:derby://localhost/dbname;create=true</code>

Läuft der Datenbankserver auf einem anderen Rechner im Netz, so muss `localhost` durch den Namen oder die IP-Nummer dieses Rechners ersetzt werden.

Die `Connection`-Methode

```
void close() throws SQLException
schließt die Verbindung zur Datenbank.
```

Programm 12.1 baut eine Verbindung zur Datenbank auf. Die Verbindungsparameter sind in der Datei `db.properties` ausgelagert:

`db.properties`:

```
# Access
#url=jdbc:ucanaccess://.../db/data/db1.accdb

# H2 embedded
url=jdbc:h2:.../db/data/db1
user=admin
password=secret

# H2 Client/Server
#url=jdbc:h2:tcp://localhost/db1
#user=admin
#password=secret
```

Durch Entfernen der Kommentarzeichen (#) werden die Einstellungen für die jeweilige Datenbank aktiviert. Hier ist vorausgesetzt, dass sich das Verzeichnis db als direktes Unterverzeichnis in GKJava befindet (siehe Abbildung 1-5).

Die Datenbanken müssen zu Beginn angelegt werden. Dies geschieht bei Derby beim erstmaligen Programmaufruf aufgrund des URL-Zusatzes `create=true` und bei H2 ohne Zusatz automatisch.

`db.properties` wird als Ressource mit der `Class`-Methode `getResourceAsStream`³ eingelesen und als Property-Liste⁴ bereitgestellt.

Programm 12.1

```
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class DBTest {
    public static void main(String[] args) {
        Connection con = null;
        try {
            Properties prop = loadDbParam(DBTest.class);
            String url = prop.getProperty("url");
```

³ Siehe Kapitel 5.6.

⁴ Siehe Kapitel 5.4.3.

```

String user = prop.getProperty("user", "");
String password = prop.getProperty("password", "");

con = DriverManager.getConnection(url, user, password);
System.out.println("Verbindung zu " + url + " wurde hergestellt.");
} catch (SQLException e) {
    System.err.println("Verbindung konnte nicht hergestellt werden.");
    System.err.println(e.getMessage());
} catch (IOException e) {
    System.err.println(e);
} finally {
    try {
        if (con != null)
            con.close();
    } catch (SQLException e) {
    }
}
}

private static Properties loadDbParam(Class<?> c) throws IOException {
    try (InputStream in = c.getResourceAsStream("db.properties")) {
        Properties prop = new Properties();
        prop.load(in);
        return prop;
    }
}
}
}

```

12.2 Daten suchen und anzeigen

Programm 12.2 ermöglicht die Suche nach Artikeln in einer Datenbanktabelle. Das Programm greift auf die Tabelle *artikel* einer Datenbank zu. Diese Tabelle enthält die Felder *Artikelnummer* (*nr*), *Artikelbezeichnung* (*bez*) und *Artikelpreis* (*preis*). Die Artikelnummer stellt den Primärschlüssel dar. In Tabelle 12-2 sind die Datentypen für die verschiedenen DBMS festgelegt.

Tabelle 12-2: Die Tabelle artikel

Spaltenname	Datentyp Access	Datentyp H2, MySQL und Derby
nr	Zahl (Integer)	integer
bez	Text (40)	varchar(40)
preis	Zahl (Double)	double

Skripte zum Aufbau der Datenbank

Das Begleitmaterial zu diesem Buch bietet die Skripte zum Erstellen der Tabelle und zum Laden der Artikeldaten.

Um nach Artikeln suchen zu können, muss nur ein Teil der Artikelbezeichnung eingegeben werden. Wird das Eingabefeld leer gelassen, so werden alle Artikel angezeigt. Die Artikelliste wird nach Artikelnummer, Artikelbezeichnung oder Artikelpreis sortiert, wenn der Benutzer auf die entsprechende Spaltenüberschrift klickt.



Abbildung 12-2: Nach Artikeln suchen

Die Klasse `ArtikelQuery` enthält die `main`-Methode, öffnet und schließt die Verbindung zur Datenbank, erzeugt das Datenmodell für die Anzeigetabelle, baut den Frame auf und führt die Datenbankabfrage aus. Die Klasse `ArtikelGUI` konstruiert die Benutzeroberfläche und behandelt die vom Benutzer initiierten Ereignisse. Die Klasse `Artikel` speichert die Daten zu einem Artikel und bietet die entsprechenden `get`-Methoden. Die Klasse `Datenmodell` verwaltet das der Tabelle zugeordnete Datenmodell. Sämtliche hier verwendeten Methoden zur Tabellenverarbeitung wurden im Kapitel 10.14 behandelt.

Programm 12.2

```

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Properties;

import javax.swing.JFrame;

```

```
public class ArtikelQuery {  
    private Connection con;  
    private Datenmodell model;  
  
    public static void main(String[] args) {  
        new ArtikelQuery();  
    }  
  
    public ArtikelQuery() {  
        try {  
            Properties prop = loadDbParam();  
            String url = prop.getProperty("url");  
            String user = prop.getProperty("user", "");  
            String password = prop.getProperty("password", "");  
  
            con = DriverManager.getConnection(url, user, password);  
        } catch (Exception e) {  
            System.err.println(e);  
            System.exit(1);  
        }  
  
        model = new Datenmodell();  
  
        JFrame frame = new JFrame("Artikel");  
        frame.getContentPane().add(new ArtikelGUI(this));  
        frame.addWindowListener(new WindowAdapter() {  
            public void windowClosing(WindowEvent e) {  
                try {  
                    if (con != null)  
                        con.close();  
                } catch (SQLException ex) {  
                }  
                System.exit(0);  
            }  
        });  
        frame.pack();  
        frame.setVisible(true);  
    }  
  
    public Datenmodell getDatenmodell() {  
        return model;  
    }  
  
    public void find(String entry) throws SQLException {  
        ArrayList<Artikel> list = model.getList();  
        list.clear();  
  
        Statement s = con.createStatement();  
        ResultSet rs = s  
            .executeQuery("select nr, bez, preis from artikel where bez like '"  
                + entry + "%' order by nr");  
        while (rs.next()) {  
            list.add(new Artikel(rs.getInt(1), rs.getString(2), rs.getDouble(3)));  
        }  
  
        rs.close();  
        s.close();  
        model.fireTableDataChanged();  
    }  
}
```

```
private Properties loadDbParam() throws IOException {
    try (InputStream in = getClass().getResourceAsStream("db.properties")) {
        Properties prop = new Properties();
        prop.load(in);
        return prop;
    }
}

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.print.PrinterException;
import java.sql.SQLException;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;

public class ArtikelGUI extends JPanel implements ActionListener {
    private ArtikelQuery query;
    private JTextField entry;
    private JLabel msg;
    private JTable table;

    public ArtikelGUI(ArtikelQuery q) {
        this.query = q;

        entry = new JTextField(25);

        JButton search = new JButton("Suche");
        search.addActionListener(this);

        table = new JTable();
        table.setAutoCreateRowSorter(true);
        table.setPreferredScrollableViewportSize(new Dimension(500, 200));
        table.setModel(query.getDatenmodell());

        msg = new JLabel(" ");
        msg.setForeground(Color.red);

        setLayout(new BorderLayout());
        JPanel p1 = new JPanel();
        p1.setLayout(new FlowLayout());
        p1.add(new JLabel("Artikelbezeichnung"));
        p1.add(entry);
        p1.add(search);

        JPanel p2 = new JPanel();
        p2.setLayout(new BorderLayout());
        p2.add(new JScrollPane(table), BorderLayout.CENTER);
        p2.add(msg, BorderLayout.SOUTH);

        JPanel p3 = new JPanel();
```

```
    JButton print = new JButton("Drucken");
    print.addActionListener(this);
    p3.add(print);

    add(p1, BorderLayout.NORTH);
    add(p2, BorderLayout.CENTER);
    add(p3, BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("Suche")) {
        doListe();
    } else {
        Runnable printTask = new Runnable() {
            public void run() {
                try {
                    table.print();
                } catch (PrinterException ex) {
                    System.out.println(ex);
                }
            }
        };
        new Thread(printTask).start();
    }
}

public void doListe() {
    try {
        query.find(entry.getText());
        int count = query.getDatenmodell().getRowCount();
        msg.setText(" " + count + " Artikel");
    } catch (SQLException ex) {
        System.err.println(ex);
    }
}

public class Artikel {
    private int nr;
    private String bez;
    private double preis;

    public Artikel(int nr, String bez, double preis) {
        this.nr = nr;
        this.bez = bez;
        this.preis = preis;
    }

    public int getNr() {
        return nr;
    }

    public String getBez() {
        return bez;
    }

    public double getPreis() {
        return preis;
    }
}
```

```
import java.util.ArrayList;
import javax.swing.table.AbstractTableModel;

public class Datenmodell extends AbstractTableModel {
    private String[] namen = { "Artikelnummer", "Artikelbezeichnung",
        "Artikelpreis" };
    private ArrayList<Artikel> list = new ArrayList<Artikel>();

    public ArrayList<Artikel> getList() {
        return list;
    }

    public int getColumnCount() {
        return namen.length;
    }

    public int getRowCount() {
        return list.size();
    }

    public Object getValueAt(int row, int col) {
        Artikel a = list.get(row);
        if (col == 0)
            return String.valueOf(a.getNr());
        else if (col == 1)
            return a.getBez();
        else
            return new Double(a.getPreis());
    }

    public String getColumnName(int col) {
        return namen[col];
    }

    public Class<?> getColumnClass(int col) {
        if (col == 2)
            return Double.class;
        else
            return String.class;
    }
}
```

Im Folgenden wird die Methode `find` der Klasse `ArtikelQuery` erläutert.

Innerhalb einer Verbindung können SQL-Anweisungen ausgeführt und Ergebnisse gelesen werden.

Connection

Die `Connection`-Methode

```
Statement createStatement() throws SQLException
```

erzeugt ein Objekt vom Typ des Interfaces `java.sql.Statement`. Dieses Objekt repräsentiert eine SQL-Anweisung.

Statement

Einige Methoden des Interfaces Statement:

```
ResultSet executeQuery(String sql) throws SQLException  
    führt die SQL-Abfrage sql aus. Das Ergebnis wird als java.sql.ResultSet-  
    Objekt geliefert.  
  
void close() throws SQLException  
    gibt die von der SQL-Anweisung benutzten Ressourcen frei.
```

select

Im Programm 12.2 wird z. B. die folgende SQL-Abfrage verwendet:

```
select * from artikel where bez like '%Signa%' order by nr
```

Diese liefert alle Datensätze der Tabelle artikel, deren Artikelbezeichnung mit dem Muster '%Signa%' übereinstimmt. % steht für beliebige Zeichen. Die Ergebnisliste ist nach Artikelnummern sortiert.

ResultSet

Einige Methoden des Interfaces ResultSet:

```
boolean next() throws SQLException  
    stellt den nächsten Datensatz zur Verfügung und liefert true oder liefert 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 geschlos-  
    sen, wenn das zugehörige Statement-Objekt geschlossen wird.
```

12.3 Daten ändern

Die Statement-Methode

```
int executeUpdate(String sql) throws SQLException  
führt die insert-, update- oder delete-Anweisung sql aus und liefert die Anzahl  
der erfolgreich bearbeiteten Datensätze.
```

Einige Beispiele:

insert

```
insert into artikel (nr, bez, preis) values (nr, bez, preis)
```

fügt einen Datensatz mit den Werten *nr*, *bez* und *preis* in die Tabelle Artikel ein. *nr*, *bez* und *preis* stehen hier für Literale. Achtung: Zeichenketten müssen durch einfache Hochkommas begrenzt werden.

Beispiel:

```
insert into artikel (nr, bez, preis) values (1120, 'Signa-Color 120-A4 weiß', 25)
```

update

```
update artikel set preis = value1 where nr = value2
```

überschreibt den Artikelpreis mit dem Wert *value1* im Datensatz mit der Artikelnummer *value2*. *value1* und *value2* stehen hier für Literale.

Beispiel:

```
update artikel set preis = 26 where nr = 1120
```

delete

```
delete from artikel where nr = value
```

löscht den Datensatz mit der Artikelnummer *value*. *value* steht hier für ein Literal.

Beispiel:

```
delete from artikel where nr = 1120
```

Programm 12.3 ändert Artikelpreise. Die Datei *update.txt* enthält die Informationen, zu welchen Artikelnummern welche neuen Preise einzutragen sind.

update.txt:

```
1120 26
3718 17
4630 16
```

Programm 12.3

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import java.util.StringTokenizer;

public class ArtikelUpdate {
    public static void main(String[] args) {
        Connection con = null;
        try {
            Properties prop = loadDbParam(ArtikelUpdate.class);
            String url = prop.getProperty("url");
            String user = prop.getProperty("user", "");
            String pass = prop.getProperty("password", "");
            // Konfiguration der Verbindung
            // ...
        } catch (SQLException e) {
            System.out.println("Fehler beim Verbinden mit der Datenbank: " + e.getMessage());
        }
    }
}
```

```
String password = prop.getProperty("password", "");  
con = DriverManager.getConnection(url, user, password);  
try (BufferedReader input = new BufferedReader(new FileReader(  
    "update.txt"))) {  
  
    String line;  
    Statement s = con.createStatement();  
    while ((line = input.readLine()) != null) {  
        StringTokenizer st = new StringTokenizer(line);  
        String nr = st.nextToken();  
        String preis = st.nextToken();  
  
        String sql = "update artikel set preis = " + preis  
            + " where nr = " + nr;  
  
        int count = s.executeUpdate(sql);  
        if (count > 0) {  
            System.out.println("Artikelnummer: " + nr  
                + ", neuer Preis: " + preis);  
        }  
        s.close();  
    }  
} catch (Exception e) {  
    System.err.println(e);  
} finally {  
    try {  
        if (con != null)  
            con.close();  
    } catch (SQLException e) {}  
}  
}  
  
private static Properties loadDbParam(Class<?> c) throws IOException {  
    try (InputStream in = c.getResourceAsStream("db.properties")) {  
        Properties prop = new Properties();  
        prop.load(in);  
        return prop;  
    }  
}
```

Die folgenden Methoden werden zur Lösung von Aufgaben und im Kapitel 13 benötigt.

Weitere Statement-Methoden

`boolean execute(String sql) throws SQLException`
führt eine SQL-Anweisung aus. Der Rückgabewert kennzeichnet das Ausführungsergebnis. Bei `true` kann das Ergebnis mit `getResultSet`, bei `false` mit `getUpdateCount` ermittelt werden.

```
ResultSet getResultSet() throws SQLException  
    liefert die Ergebnismenge der Abfrage.  
  
int getUpdateCount() throws SQLException  
    liefert das Ergebnis der SQL-Anweisung als Update-Zähler.  
  
Object getObject(int n) throws SQLException  
    liefert den Wert in der n-ten Spalte als Objekt.
```

ResultSet-Methode

```
ResultSetMetaData getMetaData() throws SQLException  
    liefert Informationen über die Ergebnismenge.
```

ResultSetMetaData-Methoden

```
int getColumnCount() throws SQLException  
    liefert die Anzahl der Spalten.  
  
String getColumnName(int col) throws SQLException  
    liefert den Namen der Spalte col.  
  
int getColumnType() throws SQLException  
    liefert den Typ der Spalte als Konstante der Klasse java.sql.Types, z. B. CHAR,  
    VARCHAR, TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL,  
    NUMERIC, DATE, TIME, TIMESTAMP.  
  
String getColumnClassName(int col) throws SQLException  
    liefert den voll qualifizierten Namen der Klasse, die von der ResultSet-  
    Methode getObject benutzt wird, um den Wert in der Spalte col zu holen.
```

12.4 Aufgaben

1. Erstellen Sie ein Programm, das alle Datensätze einer Datenbanktabelle am Bildschirm anzeigt. Die Tabelle soll Angaben zu Büchern in den Feldern *autor* und *titel* enthalten. Der Primärschlüssel *id* soll eine laufende Nummer enthalten. Die Liste soll nach Autor und Titel sortiert sein.
2. Erstellen Sie ein Programm, das neue Einträge (Autor und Titel) in der Datenbanktabelle aus Aufgabe 1 vornimmt. Die neuen Einträge sollen aus einer Datei übernommen werden, die die folgende Satzstruktur hat:

```
Autor1#Titel1  
Autor2#Titel2  
...  
...
```

Als Kommandozeilenparameter soll der id-Wert des ersten einzufügenden Satzes mitgegeben werden. Dieser Wert wird dann für die weiteren Einträge vom Programm automatisch hochgezählt.

3. Erstellen Sie ein Programm zur Abfrage von Datenbanken mit *select*-Anweisungen. Diese Anweisungen sollen über Tastatur in einer Schleife eingegeben werden. Die Spaltennamen sollen vom Programm ermittelt werden. Nutzen Sie hierzu `ResultSetMetaData`-Methoden.

Beispiel:

```
> select id, autor, titel from buch order by id
#1
id      : 1
autor   : Aitmatow, Tschingis
titel   : Aug in Auge

#2
id      : 2
autor   : Aitmatow, Tschingis
titel   : Der Richtplatz
...
```

13 Netzwerkkommunikation mit TCP/IP

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.

Lernziele

In diesem Kapitel lernen Sie

- wie Clients und Server entwickelt werden können, die über das Protokoll TCP/IP im Netzwerk miteinander kommunizieren,
- wie HTTP funktioniert,
- wie ein Webserver zur Abfrage von Datenbanken mittels SQL entwickelt werden kann.

13.1 Dateien aus dem Netz laden

Der *Uniform Resource Locator* (URL) ist die Adresse einer Ressource (Text, Bild, Sound usw.) im *World Wide Web* (WWW). Sie hat die Form

`protokoll://host[:port]/[path]`

Beispiel: `http://www.xyz.de/index.html`

`host` bezeichnet den Namen bzw. die IP-Nummer des Zielrechners. `port` und `path` sind optional. `port` steht für die Portnummer auf dem Zielrechner.¹ `path` 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` oder `default.htm` als Dateiname unterstellt. `path` muss keine physische Datei bezeichnen. Der Name kann auch für eine Ressource stehen, die erst zur Laufzeit aus diversen Quellen erzeugt wird.

URL

Objekte der Klasse `java.net.URL`, die solche Adressen repräsentieren, können mit folgenden Konstruktoren erzeugt werden:

```
URL(String url) throws java.net.MalformedURLException  
URL(URL context, String spec) throws java.net.MalformedURLException
```

¹ Portnummern werden im Kapitel 13.2 erläutert.

Im ersten Fall wird ein URL als Zeichenkette übergeben. Im zweiten Fall wird der URL aus einem Basis-URL und spec aufgebaut. In spec nicht enthaltene Adressteile werden aus context kopiert.

Beispiel:

```
URL url = new URL("http://www.xyz.de/abc/index.html");
URL url1 = new URL(url, "xxx.html");
URL url2 = new URL(url, "/xxx.html");
System.out.println(url1);
System.out.println(url2);
```

Ausgegeben wird:

```
http://www.xyz.de/abc/xxx.html
http://www.xyz.de/xxx.html
```

Die URL-Methode

`java.io.InputStream openStream() throws java.io.IOException`
stellt eine Verbindung zur Ressource her und liefert ein `InputStream`-Objekt, mit dem über diese Verbindung gelesen werden kann.

Mit dem Programm 13.1 können beliebige Dateien über das Protokoll HTTP oder FTP auf den lokalen Rechner heruntergeladen werden.

Programm 13.1

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.net.URL;

public class Download {
    public static void main(String[] args) {
        String address = args[0];
        String localFilename = args[1];

        try {
            URL url = new URL(address);
            try (BufferedInputStream input = new BufferedInputStream(
                    url.openStream());
                 BufferedOutputStream output = new BufferedOutputStream(
                     new FileOutputStream(localFilename))) {
                byte[] b = new byte[1024];
                int c;
                while ((c = input.read(b)) != -1) {
                    output.write(b, 0, c);
                }
            }
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Soll eine Datei von einem *Webserver* (HTTP-Server) über das Protokoll HTTP heruntergeladen werden, so hat die einzugebende URL-Adresse beispielsweise die Form: `http://www.xyz.de/test/xxx.zip`

Das Begleitmaterial zu diesem Buch enthält einen einfachen Webserver, mit dem getestet werden kann.

Aufrufbeispiel:

```
java -cp bin Download http://www.xyz.de/test/xxx.zip xxx.zip
```

Der zweite Kommandozeilenparameter gibt den Namen der Zielfile an. Eine Datei des lokalen Rechners kann über das Protokoll `file` gelesen werden.

Beispiel: `file:///C:/temp/info.txt`

Wird die Ressource von einem *FTP-Server* angeboten, so muss die URL-Adresse beispielsweise die folgende Form haben:

```
ftp://user:password@www.xyz.de/test/xxx.zip
```

13.2 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 Clientprogramm (z. B. ein Erfassungsformular mit Eingabeprüfung) erstellt einen Auftrag (z. B. "Ermitte den Namen des Kunden mit der Kundennummer 4711") und schickt ihn an das Serverprogramm (z. B. ein Programm, das in einer Datenbank sucht), das diesen Auftrag annimmt, bearbeitet und das Ergebnis als Antwort zurückschickt. Der Client kann sich auf demselben Rechner wie der Server oder auf einem anderen, über das Netz verbundenen Rechner befinden.

Socket

So genannte *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.

Client

Programm 13.2 implementiert einen *Client*, der eine Verbindung mit einem speziellen *Server* (Programm 13.3) 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.

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.

Methoden der Klasse Socket:

```
java.io.InputStream getInputStream() throws java.io.IOException  
    liefert den Eingabedatenstrom für diesen Socket.
```

```
java.io.OutputStream getOutputStream() throws java.io.IOException  
    liefert den Ausgabedatenstrom für diesen Socket.
```

```
void close() throws java.io.IOException  
    schließt diesen Socket.
```

Programm 13.2

```
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
import java.io.PrintWriter;  
import java.net.Socket;  
  
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);  
             BufferedReader input = new BufferedReader(  
                     new InputStreamReader(System.in))) {  
  
            System.out.println(in.readLine());  
  
            System.out.print("> ");  
            String line;  
            while ((line = input.readLine()) != null) {  
                if (line.length() == 0)  
                    break;  
  
                out.println(line);  
                System.out.println("Antwort vom Server:");  
                System.out.println(in.readLine());  
                System.out.print("> ");  
            }  
        } catch (Exception e) {  
            System.err.println(e);  
        }  
    }  
}
```

Server

Programm 13.3 implementiert den *Server*, der eine vom Client empfangene Textzeile als Echo wieder an den Client zurückschickt.

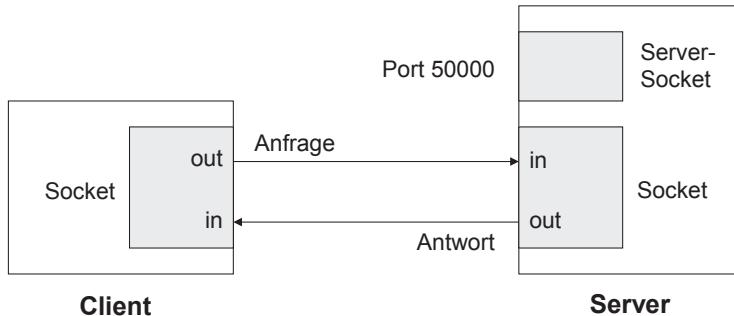


Abbildung 13-1: Kommunikation über Sockets

ServerSocket

Über ein Objekt der Klasse `java.net.ServerSocket` nimmt der Server die Verbindung zum Client auf.

`ServerSocket(int port) throws java.io.IOException`
erzeugt ein `ServerSocket`-Objekt für den Port `port`.

Die `ServerSocket`-Methode

`Socket accept() throws java.io.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.

Damit der Server mehrere Clients gleichzeitig bedienen kann, wird die eigentliche *Bearbeitung des Client-Auftrags in einem Thread* realisiert.

`void close() throws java.io.IOException`
schließt den `Server-Socket`.

Programm 13.3

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer extends Thread {
```

```
private Socket client;

public EchoServer(Socket client) {
    this.client = client;
}

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(input);
        }
    } catch (IOException e) {
        System.err.println(e);
    } finally {
        try {
            if (client != null)
                client.close();
        } catch (IOException e) {
        }
    }
}

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 EchoServer(client).start();
        }
    } catch (IOException e) {
        System.err.println(e);
    }
}
```

Zum Testen der Programme 13.2 und 13.3 muss eine freie Portnummer gewählt werden. Nummern kleiner als 1024 sind für Server des Systems reserviert und haben eine festgelegte Bedeutung.

Um Client und Server zusammen auf einem einzigen Rechner (ohne Netzverbindung) zu testen, kann beim Start des Client als Rechnername `localhost` angegeben werden.

Aufruf des Servers:

```
java -cp bin EchoServer 50000
```

Der Server meldet:

```
EchoServer auf 50000 gestartet ...
```

Aufruf des Clients (in einem zweiten DOS-Fenster):

```
java -cp EchoClient localhost 50000
```

Der Client gibt aus:

```
Hallo, ich bin der EchoServer  
>
```

Eingabe beim Client:

```
Das ist ein Test.
```

Der Client meldet:

```
Antwort vom Server:  
Das ist ein Test.  
>
```

ENTER beendet den Client. Strg + C beendet den Server.

13.3 HTTP-Transaktionen

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.xyz.de/produkte/index.html,
```

den Webserver (im Beispiel: `www.xyz.de`) und stellt eine TCP-Verbindung zum Server her (Port: 80).

Über diese Verbindung sendet er dann eine Anfrage, z. B. die Aufforderung, eine HTML-Seite zu übermitteln (im Beispiel: `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 (bis zum nächsten Update).

Eine *dynamische* Seite wird erst dann erzeugt, wenn der Browser diese Seite anfordert.

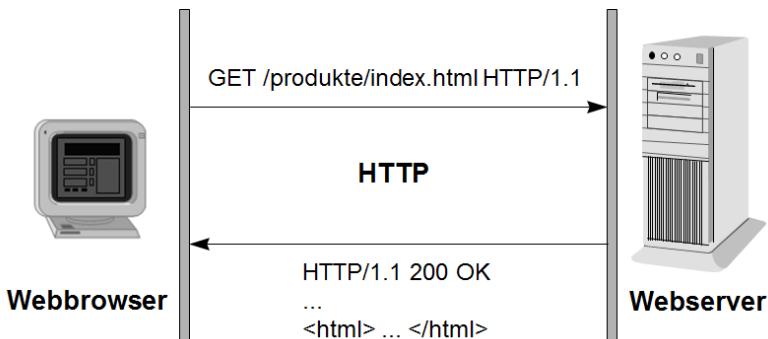


Abbildung 13-2: HTTP-Anfrage und -Antwort

Die Seitenbeschreibungssprache *HTML* bietet die Möglichkeit, Formulare zu definieren, 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.

13.3.1 Formulardaten über HTTP senden

Anwendungsbeispiel

Programm 13.6 ist ein spezieller Webserver, der mit einer Datenbank verbunden ist, beliebige SQL-Anweisungen für diese Datenbank über HTTP empfängt und die Ausführungsergebnisse (Tabellendaten, Status- oder Fehlermeldungen) zum Browser zurückschickt (siehe Abbildung 13-3). Dabei muss der Server die Anfrage (*HTTP-Request*) interpretieren und die Antwort gemäß den HTTP-Konventionen aufbereiten (*HTTP-Response*). Die clientseitige Eingabe der SQL-Anweisung erfolgt in einem HTML-Formular. Alle HTML-Seiten werden dynamisch erzeugt.

Zunächst muss der hier relevante Teil des HTTP-Protokolls verstanden werden.

Analyse des HTTP-Request

Programm 13.4 zeigt, was genau der Browser zum Server schickt, wenn Daten im Formular eingegeben und gesendet werden.

Zum Testen wird die HTML-Seite *Formular.html* benutzt, die vom Browser lokal aufgerufen wird.

SQL jdbc:h2:tcp://localhost/db1

```
select * from artikel where nr < 1600
```

NR	BEZ	PREIS
1120	Signa-Color 120-A4 weiß	26.0
1122	Signa-Color 80-A5 weiß	10.0
1515	Signa-Color 70-A4 weiß	12.0
1517	Signa-Color 70-A4 hellgrün	13.0

Abbildung 13-3: SQL-Abfrage

Die HTML-Seite *Formular.html* hat den folgenden Aufbau:

```
<html>
<head>
<title>Datenbankabfrage</title>
</head>
<body>
    <b>SQL</b>
    <p />
    <form action="http://localhost:50000" method="GET">
        <textarea cols="60" rows="4" name="sql"></textarea>
        <p />
        <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. Die `POST`-Methode wird in Aufgabe 4 dieses Kapitels behandelt. Das Formular enthält ein mehrzeiliges Textfeld (`textarea`) mit dem Namen `sql` und einen Submit-Button, der URL und Eingabedaten zum Server sendet.

Programm 13.4

```

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) {
        final 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.length() == 0)
                        break;
                }
            }

            client.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}

```

Testablauf:

1. Aufruf des Servers: `java -cp bin TestServer1`
2. Öffnen des Formulars `Formular.html` mit dem Browser
3. Eingabe der SQL-Anweisung: `select * from artikel where nr < 1600`
4. Senden

Der Server gibt folgende Daten aus (hier am Beispiel von Firefox gezeigt):

HTTP-Request

```

GET /?sql=select+*+from+artikel+where+nr+%3C+1600 HTTP/1.1
Host: localhost:50000
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:30.0) Gecko/...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive

```

Alle Zeilen des Request enden jeweils mit *Carriage Return* und *Linefeed*: `\r\n`.

Die erste Zeile enthält die Anforderung. Die Daten aus einem Formular mit mehreren Eingabefeldern werden allgemein 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.

Programm 13.5 extrahiert den Query String aus der ersten Zeile des Request und decodiert ihn mit Hilfe einer Methode der Klasse `java.net.URLDecoder`:

```
static String decode(String s, String enc)
    throws java.io.UnsupportedEncodingException
```

enc ist der Name des verwendeten Codierungsschemas, hier "ISO-8859-1".

Nach der Decodierung des Query Strings stimmt er mit der Eingabe im Formular überein.

Programm 13.5

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.URLDecoder;

public class TestServer2 {
    public static void main(String[] args) {
        final 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);
                String decodedQuery = URLDecoder.decode(query, "ISO-8859-1");
                System.out.println(decodedQuery);
            }

            client.close();
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

13.3.2 Ein spezieller HTTP-Server für SQL-Anweisungen

Programm 13.6 implementiert das oben skizzierte Anwendungsbeispiel. Wir nutzen die H2-Datenbank aus Kapitel 12.

Nach Eingabe der Adresse `http://localhost:50000` im Browser schickt der Server zunächst das von ihm erzeugte "leere" Eingabeformular. Anstelle von `localhost` kann auch der Name (IP-Adresse) des Rechners stehen, falls er ans Netz angeschlossen ist.

HTTP-Response

Der Server muss die Ergebnisdaten der SQL-Anweisung als HTML-Seite aufbereiten.

Die Rückantwort 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 Zeilen des Request enden jeweils mit *Carriage Return* und *Linefeed*: `\r\n`.

Das generierte Leer-Formular:

```
<html>
<head>
<title>Datenbankabfrage</title>
</head>
<body>
    <b>SQL jdbc:h2:tcp://localhost/db1</b>
    <p />
    <form method="GET">
        <textarea cols="60" rows="4" name="sql"></textarea>
        <p />
        <input type="submit" value="Senden" />
    </form>
    <p />
</body>
</html>
```

Hier fehlt das Attribut `action` im Tag `<form>`. Der Browser setzt automatisch die Adresse des Servers ein, der das Formular geschickt hat.

Die SQL-Anweisung

```
select * from artikel where nr = 1120
```

führt zu folgendem generierten Ergebnis:

```
<html>
<head>
<title>Datenbankabfrage</title>
</head>
<body>
    <b>SQL jdbc:h2:tcp://localhost/db1</b>
    <p />
    <form method="GET">
        <textarea cols="60" rows="4" name="sql">select * from artikel where nr =
1120</textarea>
        <p />
        <input type="submit" value="Senden" />
    </form>
    <p />
    <table border="1" cellpadding="5">
        <tr>
            <th align="right">NR</th>
            <th align="left">BEZ</th>
            <th align="right">PREIS</th>
        </tr>
        <tr>
            <td valign="top" align="right">1120</td>
            <td valign="top" align="left">Signa-Color 120-A4 weiß</td>
            <td valign="top" align="right">26.0</td>
        </tr>
    </table>
</body>
</html>
```

Programm 13.6

```
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.util.Properties;

public class HttpServer {
    public static void main(String[] args) {
        final int PORT = 50000;

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            Properties prop = loadDbParam(HttpServer.class);

            System.out.println("HttpServer auf " + PORT + " gestartet ...");
            System.out.println("Datenbank-URL: " + prop.getProperty("url"));

            while (true) {
                try {
                    new HttpConnection(serverSocket.accept(), prop).start();
                } catch (IOException e) {
                    System.err.println(e);
                }
            }
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

```
private static Properties loadDbParam(Class<?> c) throws IOException {
    try (InputStream in = c.getResourceAsStream("db.properties")) {
        Properties prop = new Properties();
        prop.load(in);
        return prop;
    }
}

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.URLDecoder;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.util.Properties;

public class HttpConnection extends Thread {
    private static final String CHAR_SET = "ISO-8859-1";
    private Socket socket;
    private Properties prop;

    public HttpConnection(Socket client, Properties prop) {
        this.socket = client;
        this.prop = prop;
    }

    public void run() {
        try (BufferedReader in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                true));
            Connection con = DriverManager.getConnection(
                prop.getProperty("url"), prop.getProperty("user", ""),
                prop.getProperty("password", ""))) {

            String sql = readRequest(in);
            if (sql.length() == 0) {
                sendForm(out, null);
            } else {
                sendForm(out, sql);
                query(con, out, sql);
            }
        } catch (Exception e) {
            System.err.println(e);
        } finally {
            try {
                if (socket != null)
                    socket.close();
            } catch (Exception e) {
            }
        }
    }
}
```

```
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 query = line.substring(x1 + 1, x2);
    return URLDecoder.decode(query, CHAR_SET).trim();
}

private String escape(String s) {
    StringBuilder sb = new StringBuilder();
    int length = s.length();
    for (int i = 0; i < length; i++) {
        char c = s.charAt(i);
        if (c == '<')
            sb.append("&lt;");
        else if (c == '>')
            sb.append("&gt;");
        else
            sb.append(c);
    }
    return sb.toString();
}

private void sendForm(PrintWriter out, String sql) throws IOException {
    out.print("HTTP/1.1 200 OK\r\n");
    out.print("Content-Type: text/html\r\n\r\n");
    out.println("<html>");
    out.println("<head><title>Datenbankabfrage</title></head>");
    out.println("<body>");
    out.println("<b>SQL " + prop.getProperty("url") + "</b><p />");
    out.println("<form method=\"GET\">");
    out.print("<textarea cols=\"60\" rows=\"4\" name=\"sql\">");

    if (sql != null)
        out.print(escape(sql));

    out.println("</textarea>");
    out.println("<p /><input type=\"submit\" value=\"Senden\" />");
    out.println("</form><p />");

    if (sql == null)
        out.println("</body></html>");
}

private void query(Connection con, PrintWriter out, String sql)
    throws SQLException {
    try (Statement stmt = con.createStatement()) {
        if (!stmt.execute(sql)) {
            out.println(stmt.getUpdateCount() + " Zeile(n)");
            return;
        }

        ResultSet rs = stmt.getResultSet();
        ResultSetMetaData rm = rs.getMetaData();
        int n = rm.getColumnCount();
    }
}
```

```

String[] align = new String[n];
out.println("<table border=\"1\" cellpadding=\"5\"><tr>");
for (int i = 1; i <= n; i++) {
    if (rm.getColumnType(i) == Types.TINYINT
        || rm.getColumnType(i) == Types.SMALLINT
        || rm.getColumnType(i) == Types.INTEGER
        || rm.getColumnType(i) == Types.BIGINT
        || rm.getColumnType(i) == Types.REAL
        || rm.getColumnType(i) == Types.FLOAT
        || rm.getColumnType(i) == Types.DOUBLE
        || rm.getColumnType(i) == Types.DECIMAL
        || rm.getColumnType(i) == Types.NUMERIC)
        align[i - 1] = "right";
    else
        align[i - 1] = "left";
    out.println("<th align=\"" + align[i - 1] + "\""
               + rm.getColumnName(i) + "</th>");
}
out.println("</tr>");
while (rs.next()) {
    out.println("<tr>");
    for (int i = 1; i <= n; i++) {
        out.println("<td valign=\"top\" align=\"" + align[i - 1]
                   + "\">" + rs.getString(i) + "</td>");
    }
    out.println("</tr>");
}
out.println("</table>");
} catch (SQLException e) {
    out.println(e.getMessage());
} finally {
    out.println("</body></html>");
}
}

```

Die Verbindungsparameter für die Datenbank sind in der Datei `db.properties` gespeichert.²

Im clientbezogenen Thread `HttpConnection` wird die Verbindung zur Datenbank aufgebaut und nach Erfüllung der Aufgabe wieder abgebaut. Die Methode `readRequest` extrahiert und decodiert die SQL-Anweisung aus dem in der ersten Zeile des HTTP-Requests enthaltenen Query String.³ Die Methode `sendForm` erzeugt eine HTML-Seite mit leerem Formular, wenn das zweite übergebene

² Siehe Kapitel 12.1.

³ Siehe Kapitel 13.3.1.

Argument `null` ist, andernfalls wird das Formular mit der "alten" SQL-Anweisung als Vorgabewert für das mehrzeilige Textfeld generiert. Zuvor werden die kritischen Zeichen '`<`' und '`>`', die für HTML eine besondere Bedeutung haben, durch so genannte *Character Entities* ersetzt. Im Anschluss daran wird die SQL-Anweisung (`select`, `insert`, `update` oder `delete`) ausgeführt und das Ergebnis als HTML-Ausgabe generiert. Hierzu werden insbesondere die am Ende von Kapitel 12.3 aufgeführten Methoden verwendet.

Der Server wird wie folgt aufgerufen:

```
java -cp bin;JDBC-Treiber HttpServer
```

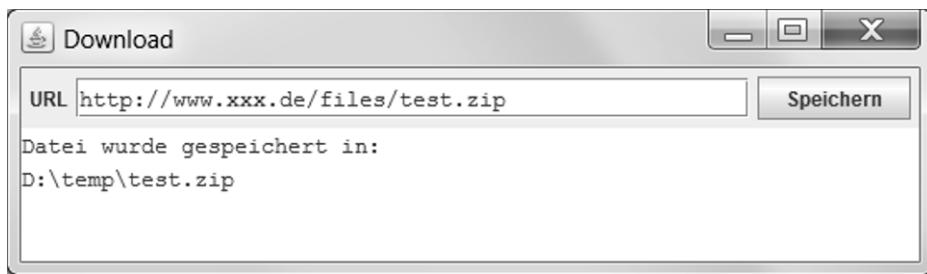
Der JDBC-Treiber für H2 muss in den `CLASSPATH` aufgenommen werden.

Selbstverständlich können auch Änderungen der Datenbank vorgenommen werden, z. B.

```
update artikel set preis = 20 where nr = 1120
```

13.4 Aufgaben

1. Entwickeln Sie eine grafische Oberfläche zum Download-Programm 13.1. Der lokale Speicherort soll über einen Dateiauswahldialog (`JFileChooser`) bestimmt werden. Die Dateiübertragung soll in einem eigenen Thread erfolgen.



2. Erstellen Sie einen Server, der nach Aufnahme einer Verbindung die aktuelle Systemzeit an den Client sendet. Implementieren Sie auch den passenden Client dazu.
3. Entwickeln Sie einen Server, der in einem HTML-Formular eingetragene Mail-Adressen entgegennimmt und in einer Datenbank speichert. Das Formular soll die Felder *nachname*, *vorname* und *email* enthalten. Die ersten beiden Felder können auch genutzt werden, um über den Namen und/oder Vornamen nach Mail-Adressen zu suchen. Über *RadioButtons* soll die gewünschte Aktion (*Suchen* oder *Einfügen*) eingestellt werden können.

Abfrage und Erfassung von Mail-Adressen

Nachname	<input type="text" value="M%er"/>	
Vorname	<input type="text" value="%"/>	
E-Mail	<input type="text"/>	
<input checked="" type="radio"/> Suchen <input type="radio"/> Einfügen		
<input type="button" value="Senden"/> <input type="button" value="Zurücksetzen"/>		
Nachname	Vorname	E-Mail
Mayer	Fritz	fritz.mayer@xyz.de
Meier	Hugo	hugo.meier@web.de

- Entwickeln Sie eine Variante zu Aufgabe 3, bei der statt der GET- die POST-Methode im Formular verwendet wird. Die gesendeten Daten werden dann nicht an den URL angehängt und sind also 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.

- Realisieren Sie einen einfachen Client SimpleHttpClient zum Laden von Webseiten (HTTP-Antwort und Rohdaten). Das Programm soll beispielsweise wie folgt aufgerufen werden können:

```
java -cp bin SimpleHttpClient http://www.google.de
```

Mittels der URL-Methoden `getProtocol`, `getHost`, `getPort` und `getFile` können der Protokollname, der Hostname, die Portnummer und der Pfadname ermittelt werden. Der Client soll die folgende Zeichenkette senden.

Passend zum obigen Beispiel also:

```
GET / HTTP/1.1\r\n
User-Agent: SimpleHttpClient\r\n
Host: www.google.de:80\r\n
Accept: */*\r\n
Connection: close\r\n\r\n
```

14 Fallbeispiel

In diesem Kapitel wird ein etwas umfangreicheres Programmbeispiel vorgestellt. Es zeigt, dass eine gut durchdachte Softwarearchitektur für das zu entwickelnde Softwareprodukt eine wesentliche Grundlage der Entwicklung ist. Zur Implementierung werden Klassen und Methoden der vorhergehenden Kapitel benutzt.¹

Lernziele

In diesem Kapitel lernen Sie

- wie ein Programm gemäß der Drei-Schichten-Architektur (Präsentations-, Anwendungs- und Persistenzschicht) strukturiert werden kann,
- wie Klassen zur Realisierung der grafischen Oberfläche, zur Umsetzung der fachlichen Logik und zur Verwaltung der Daten in einer Datenbank diesen Schichten zugeordnet werden können,
- wie die lauffähige Anwendung konfiguriert und zur Ausführung bereitgestellt werden kann (*Deployment*).

14.1 Die Anwendung

Programm 14.1

Das Programm "Adressenverwaltung" (Programm 14.1) ermöglicht die Anzeige und Pflege von Adressen (E-Mail, Webseite) über eine grafische Benutzeroberfläche. Die Adressdaten werden in einer H2-Datenbank (Datenbankname: *gkjava_db*) gespeichert.

Die Datenbanktabelle *address* wird im Programm mit der folgenden SQL-Anweisung erstellt:

```
create table if not exists address (
    id integer not null auto_increment,
    lastname varchar(40),
    firstname varchar(40),
    email varchar(40),
    email_additional varchar(40),
    homepage varchar(60),
    primary key (id)
);
```

¹ Siehe insbesondere Kapitel 10 und Kapitel 12.

Wir kommen in unserem Beispiel mit einer einzigen Tabelle aus. Bei größeren Anwendungen besteht das Datenmodell aus mehreren Tabellen, die miteinander über Schlüsselbeziehungen verknüpft sind.

Die H2-Datenbank `gkjava_db` wird im eingebetteten Modus betrieben und im User-Verzeichnis `user.home` abgelegt.²

Inhalt der Datei `db.properties`:

```
url=jdbc:h2:~/gkjava_db
```

Abbildung 14-1 zeigt das Fenster nach Start des Programms. Alle Namen zu den in der Datenbank vorhandenen Adressen werden in einer Liste aufgeführt.

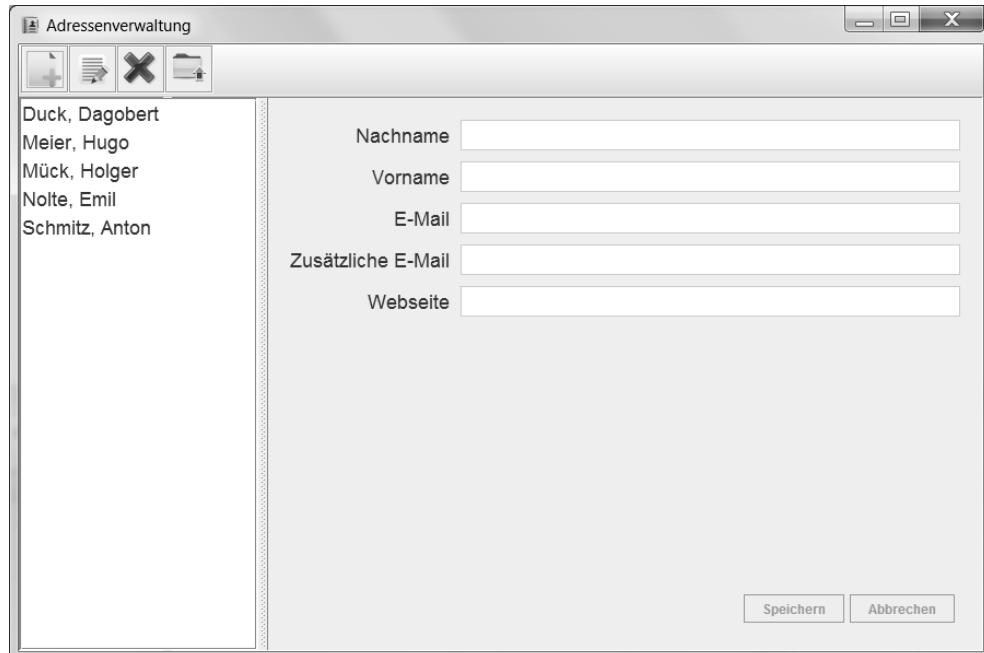


Abbildung 14-1: Nach Start des Programms

Um eine bestehende Adresse zu ändern (3), muss der entsprechende Listeneintrag selektiert (1) und anschließend der zweite Button in der Symbolleiste betätigt werden (2). Alternativ führt ein Doppelklick auf dem Listeneintrag zum selben Ergebnis (siehe Abbildung 14-2).

² Bei Windows 7 z. B. C:\Users\<username>

Die Eingabe von *ENTER* in den Feldern E-Mail, Zusätzliche E-Mail und Webseite ruft den E-Mail-Client bzw. den Webbrower auf. Bei der Datenerfassung ist zu beachten, dass die Webseite den Vorspann `http://` nicht enthalten darf.

Vor Speicherung der Daten (4) in der Datenbank erfolgt eine Prüfung der eingegebenen Daten.

Hinzufügen, Löschen, Exportieren

Bestehende Adressen können gelöscht und neue Adressen hinzugefügt werden. Alle Adressen können im *CSV-Format* (Comma Separated Values) exportiert werden, sodass diese beispielsweise in Form einer Excel-Tabelle dargestellt werden können.

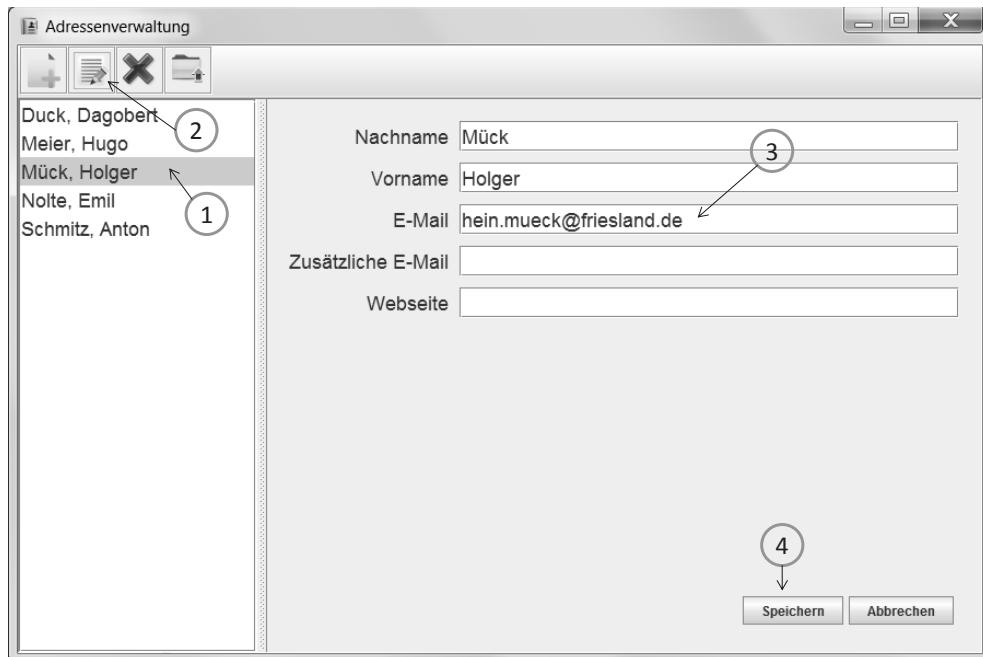


Abbildung 14-2: Eine Adresse ändern

14.2 Drei-Schichten-Architektur

Der grundlegende Ansatz zur Strukturierung von Softwaresystemen ist die *Zerlegung in Schichten*.

Jede Schicht soll ihre Funktionalität kapseln und ihre Leistungen über eine möglichst schmale Schnittstelle nach außen anbieten. Die einzelnen Schichten des Systems sollen nur lose miteinander gekoppelt sein, sodass im Prinzip die spezielle

Implementierung einer Schicht leicht durch eine andere ersetzt werden kann (z. B. Webbrower statt Swing-Client, einfache Dateiverwaltung statt Datenbankverwaltung).

Die meisten Systeme bestehen aus drei Schichten (siehe Abbildung 14-3): *Präsentationsschicht*, *Anwendungsschicht* und *Persistenzschicht*.

Präsentationsschicht

Die *Präsentationsschicht* bietet eine meist grafische Benutzeroberfläche, um Daten der Anwendung in geeigneter Form darzustellen. Bekannte Darstellungsmittel sind Textfelder, Auswahllisten, Checkboxen usw. Andererseits kann der Benutzer reagieren und durch Klick auf einen Button oder durch Auswahl eines Menüeintrags Ereignisse auslösen, die an die Anwendungsschicht zur Bearbeitung weitergeleitet werden.

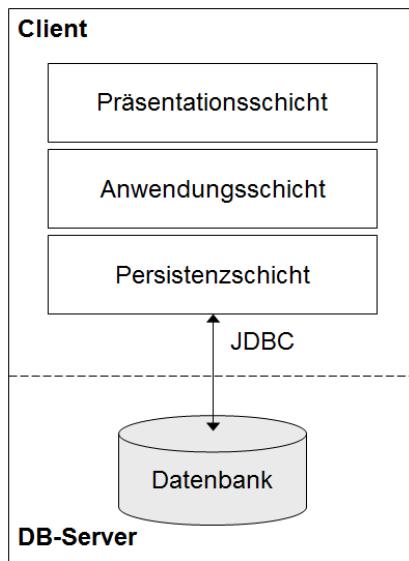


Abbildung 14-3: Drei-Schichten-Architektur

Anwendungsschicht

In der *Anwendungsschicht* werden alle fachlichen Aufgaben realisiert. Dazu gehören fachliche Objekte (in unserem Beispiel die Adresse) und die Abbildung von Geschäftsprozessen (Anwendungslogik). Diese Schicht hat kein konkretes Wissen über die Benutzeroberfläche und weiß auch nicht, wie die Daten dauerhaft (persistent) gespeichert werden.

Persistenzschicht

Die *Persistenzschicht* sorgt dafür, dass die fachlichen Objekte der Anwendung dauerhaft verwaltet werden können. Bei Nutzung eines Datenbanksystems werden hier die Datenbankzugriffe zum Lesen und Schreiben (SQL-Befehle) implementiert. Diese Schicht muss deshalb die Tabellenstrukturen der Datenbank kennen.

14.3 Klassenentwurf und Architektur

Bevor wir die Klassen genauer vorstellen, gibt das *Klassendiagramm* in Abbildung 14-4 einen Überblick über die Zusammenhänge und die Zuordnung der Pakete und Klassen zu den einzelnen Schichten. Pfeile deuten eine *Benutzt*-Beziehung an, in dem Sinne, dass die Klasse (Pfeilspitze) im Quellcode explizit verwendet wird.

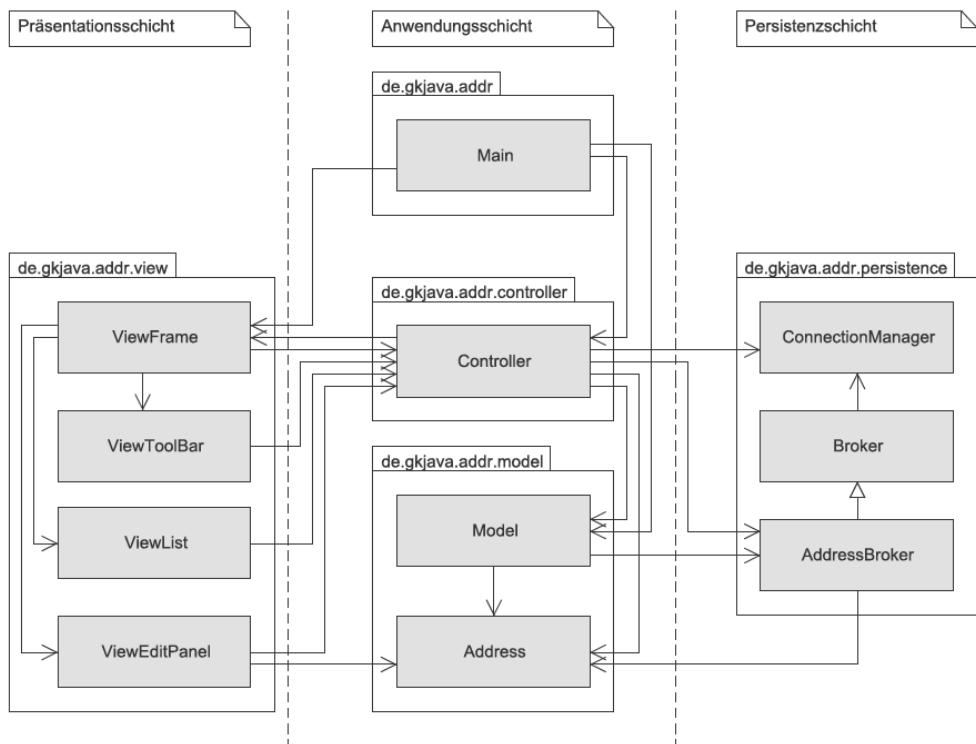


Abbildung 14-4: Klassendiagramm

Die Klassen `ConnectionManager` und `Broker<T>` sind völlig unabhängig von dem konkreten Anwendungsbeispiel und können somit in ähnlichen Fällen wieder-

verwendet werden. Die Klasse `AddressBroker` ist von `Broker<Address>` abgeleitet und enthält insbesondere die SQL-Befehle für den Datenbankzugriff.

Mit der Klasse `Main` wird die Anwendung gestartet. Objekte der Klasse `Address` nehmen jeweils eine Adresse auf und dienen als Datencontainer für den Austausch zwischen Datenbank und Benutzungsoberfläche. Die Klasse `Model` verwaltet die Liste aller Adressen im Arbeitsspeicher und bietet die hierfür nötigen Zugriffsmethoden. Die Klasse `Controller` stellt Methoden bereit, um Adressen aus der Datenbank zu holen, neue Adressen aufzunehmen, bestehende Adressen zu löschen und Adressen zu exportieren. Der `Controller` fungiert auch als Mittler zwischen Präsentations- und Persistenzschicht. Hier erfolgt die Reaktion auf durch Dialogelemente der Oberfläche ausgelöste Ereignisse.

Die Klassen `ViewFrame`, `ViewToolbar`, `ViewList` und `ViewEditPanel` enthalten die Swing-Komponenten zum Aufbau der Benutzungsoberfläche. Hier sind auch die `Listener` zur Behandlung der diversen Ereignisse implementiert.

14.4 Implementierung

Wir stellen die einzelnen Klassen gruppiert nach ihrer zugehörigen Schicht vor.

14.4.1 Persistenzschicht

Die Klassen der Persistenzschicht gehören alle zum Paket `de.gkjava.addr.persistence`. Die Klasse `ConnectionManager` baut die Verbindung zur Datenbank auf bzw. schließt diese Verbindung wieder.

ConnectionManager

```
package de.gkjava.addr.persistence;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class ConnectionManager {
    private static final String FILE = "/db.properties";
    private static Connection con;

    private ConnectionManager() {
    }

    // Verbindung herstellen
    public static Connection getConnection() throws IOException, SQLException {
        if (con == null) {
            Properties prop = loadDbParam();
```

```

        con = DriverManager.getConnection(prop.getProperty("url"),
            prop.getProperty("user", ""),
            prop.getProperty("password", ""));
    }
    return con;
}

// Verbindung schliessen
public static void closeConnection() {
    try {
        if (con != null) {
            con.close();
            con = null;
        }
    } catch (SQLException e) {
    }
}

// Verbindungseigenschaften bereitstellen
private static Properties loadDbParam() throws IOException {
    try (InputStream in = ConnectionManager.class.getResourceAsStream(FILE)) {
        Properties prop = new Properties();
        prop.load(in);
        return prop;
    }
}
}

```

Mit `getConnection` wird ein `Connection`-Objekt bereitgestellt. Nur beim ersten Aufruf wird eine Verbindung hergestellt. Die Referenz auf dieses Objekt steht dann in einer Klassenvariablen für spätere Aufrufe zur Verfügung. Die Datei `dbparam.txt` enthält die Verbindungsparameter. Sie wird mittels `getResourceAsStream`³ gefunden. Ihr Inhalt wird als `Properties`-Objekt bereitgestellt.

Broker

Die generische Klasse `Broker<T>` enthält die Methoden `query`, `update` und `insertAndReturnKey` sowie die abstrakte Methode `makeObject`, die in konkreten Subklassen implementiert werden muss. `query`, `update` und `insertAndReturnKey` nutzen einen SQL-Befehl, der als Aufrufparameter mitgegeben wird. `query` gibt ein `List`-Objekt zurück, dessen Einträge mittels `makeObject` anwendungsabhängig (siehe `AddressBroker`) erzeugt werden.⁴

Die Methode `insertAndReturnKey` enthält eine Besonderheit. Der zweite Parameter der `Statement`-Methode `executeUpdate` bestimmt, dass der von der Datenbank

³ Siehe Kapitel 5.6.

⁴ Die Einzelheiten zu JDBC-Zugriffen sind im Kapitel 12 zu finden.

automatisch generierte Schlüssel über die Statement-Methode `getGeneratedKeys` zur Verfügung gestellt wird.

```
package de.gkjava.addr.persistence;

import java.io.IOException;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public abstract class Broker<T> {
    // Diese Methode muss von Subklassen implementieren werden
    protected abstract T makeObject(ResultSet rs) throws SQLException;

    // Select
    protected List<T> query(String sql) throws IOException, SQLException {
        try (Statement stmt = ConnectionManager.getConnection()
            .createStatement()) {

            ResultSet rs = stmt.executeQuery(sql);

            List<T> result = new ArrayList<T>();
            while (rs.next()) {
                result.add(makeObject(rs));
            }

            return result;
        }
    }

    // Update, Insert oder Delete
    protected int update(String sql) throws IOException, SQLException {
        try (Statement stmt = ConnectionManager.getConnection()
            .createStatement()) {

            int count = stmt.executeUpdate(sql);
            return count;
        }
    }

    // Insert mit Rückgabe des automatisch erzeugten Schlüssels
    protected int insertAndReturnKey(String sql) throws IOException,
        SQLException {

        try (Statement stmt = ConnectionManager.getConnection()
            .createStatement()) {

            stmt.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS);
            ResultSet rs = stmt.getGeneratedKeys();
            rs.next();
            int id = rs.getInt(1);
            return id;
        }
    }
}
```

AddressBroker

AddressBroker hat das "Wissen" über die konkrete Tabellenstruktur der Datenbank. Ein AddressBroker-Objekt kann nur mittels getInstance erzeugt werden. Dies stellt sicher, dass nur ein einziges Objekt dieser Klasse erstellt wird (*Singleton*). Man beachte, dass der Konstruktor private ist. makeObject überschreibt die von Broker<Address> geerbte abstrakte Methode.⁵

```
package de.gkjava.addr.persistence;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import de.gkjava.addr.model.Address;

public class AddressBroker extends Broker<Address> {
    private static AddressBroker instance;

    private AddressBroker() {
    }

    public static AddressBroker getInstance() {
        if (instance == null)
            instance = new AddressBroker();
        return instance;
    }

    protected Address makeObject(ResultSet rs) throws SQLException {
        Address a = new Address();
        a.setId(rs.getInt(1));
        a.setLastname(rs.getString(2));
        a.setFirstname(rs.getString(3));
        a.setEmail(rs.getString(4));
        a.setEmailAdditional(rs.getString(5));
        a.setHomepage(rs.getString(6));
        return a;
    }

    // Alle Adressen holen
    public List<Address> findAll() throws Exception {
        return query("select * from address order by lastname, firstname");
    }

    // Eine neue Adresse speichern mit Rückgabe des generierten Schlüssels
    public int insert(Address a) throws Exception {
        return insertAndReturnKey("insert into address "
            + "(lastname, firstname, email, email_additional, homepage) values ('"
            + a.getLastname() + "','" + a.getFirstname() + "','" + a.getEmail() + "','" + a.getEmailAdditional() + "','" + a.getHomepage() + "')");
    }
}
```

⁵ Einzelheiten zur SQL-Syntax sind im Kapitel 12 zu finden.

```

// Eine Adresse ändern
public void update(Address a) throws Exception {
    update("update address set " + "lastname = '" + a.getLastname()
        + "', firstname = '" + a.getFirstname() + "'", email = ''
        + a.getEmail() + "', email_additional = ''"
        + a.getEmailAdditional() + "'", homepage = '' + a.getHomepage()
        + "' where id = " + a.getId());
}

// Eine Adresse löschen
public void delete(int id) throws Exception {
    update("delete from address where id = " + id);
}

// Tabelle erstellen
public void createTable() throws Exception {
    update("create table if not exists address (" +
        "id integer not null auto_increment, "
        + "lastname varchar(40), "
        + "firstname varchar(40), "
        + "email varchar(40), "
        + "email_additional varchar(40), "
        + "homepage varchar(60), "
        + "primary key (id))");
}
}

```

14.4.2 Anwendungsschicht

Die Klassen der Anwendungsschicht gehören zu drei unterschiedlichen Pakten: `de.gkjava.addr.model`, `de.gkjava.addr.controller` und `de.gkjava.addr`.

Address

Die Klasse `de.gkjava.addr.model.Address` enthält `get-` und `set-`Methoden zum Lesen und Setzen der einzelnen Attribute einer Adresse.

```

package de.gkjava.addr.model;

import java.text.Collator;

public class Address implements Comparable<Address> {
    private int id;
    private String lastname;
    private String firstname;
    private String email;
    private String emailAdditional;
    private String homepage;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}

```

```
public String getLastname() {
    return lastname;
}

public void setLastname(String lastname) {
    this.lastname = lastname;
}

public String getFirstname() {
    return firstname;
}

public void setFirstname(String firstname) {
    this.firstname = firstname;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getEmailAdditional() {
    return emailAdditional;
}

public void setEmailAdditional(String emailAdditional) {
    this.emailAdditional = emailAdditional;
}

public String getHomepage() {
    return homepage;
}

public void setHomepage(String homepage) {
    this.homepage = homepage;
}

// Wird für die Sortierung der Adressliste benutzt
public int compareTo(Address a) {
    Collator collator = Collator.getInstance();
    String s1 = lastname + ", " + firstname;
    String s2 = a.getLastname() + ", " + a.getFirstname();
    return collator.compare(s1, s2);
}
```

Address implementiert das Interface Comparable<Address>. Die Methode compareTo wird für die Sortierung der Adressen nach Namen in der Klasse Model verwendet. Mit der Methode compare der Klasse java.text.Collator ist ein sprachabhängiger lexikographischer Vergleich von Zeichenketten möglich. Damit werden dann im

Deutschen z. B. die Umlaute richtig einsortiert. `getInstance` liefert das Collator-Objekt für das standardmäßig eingestellte Gebietsschema (`Locale`).⁶

Model

Die Klasse `de.gkjava.addr.model.Model` verwaltet die Liste aller Adressen.

```
package de.gkjava.addr.model;

import java.util.Collections;
import java.util.List;
import java.util.Vector;

import de.gkjava.addr.persistence.AddressBroker;

public class Model {
    private List<Address> data;

    public void setData() throws Exception {
        data = AddressBroker.getInstance().findAll();
    }

    public List<Address> getData() {
        return data;
    }

    // Erzeugung der sortierten Namensliste für die Anzeige
    public Vector<String> getNames() {
        Collections.sort(data);
        Vector<String> names = new Vector<String>();
        for (Address address : data) {
            String firstname = address.getFirstname();
            if (firstname.length() > 0) {
                names.add(address.getLastname() + ", " + firstname);
            } else {
                names.add(address.getLastname());
            }
        }
        return names;
    }

    // Ermittlung der Position einer Adresse in der Liste
    public int getIndex(int id) {
        int size = data.size();
        for (int i = 0; i < size; i++) {
            if (data.get(i).getId() == id) {
                return i;
            }
        }
        return -1;
    }
}
```

⁶ Siehe Kapitel 5.10.

```
public Address get(int idx) {
    return data.get(idx);
}

public void set(int idx, Address address) {
    data.set(idx, address);
}

public void remove(int idx) {
    data.remove(idx);
}

public void add(Address address) {
    data.add(address);
}
```

Die Methode `getNames` extrahiert Nachname und Vorname der Adressen aus der Liste für die Darstellung an der Oberfläche (`JList`). Zu Beginn werden die Adressen mit Hilfe der `sort`-Methode der Klasse `Collections` sortiert (siehe `compare`-Methode der Klasse `Address`). Die Methode `getNames` wird immer nach einer Adressänderung und der Aufnahme einer neuen Adresse aufgerufen, damit die Namen in der Anzeige in der richtigen Sortierreihenfolge erscheinen.

`getIndex` bestimmt die Position einer mit Hilfe des Schlüssels identifizierten Adresse in der angezeigten Liste. Somit kann der Listeneintrag nach Änderung bzw. Neuaufnahme und der erfolgten Sortierung selektiert werden.

Controller

Die Methoden von `de.gkjava.addr.controller.Controller` werden hauptsächlich aus den Klassen der Präsentationsschicht aufgerufen. Sie delegieren u. a. die Aufgaben an die Persistenzschicht.

Aufgaben dieser Klasse sind:

- Erstellen der Tabelle in der Datenbank,
- Laden aller Adressen aus der Datenbank,
- Vorbereitung der Erfassung einer neuen bzw. Änderung einer bestehenden Adresse,
- Speichern einer neuen bzw. geänderten Adresse,
- Löschen einer Adresse mit Bestätigung,
- Abbrechen,
- Exportieren aller Adressen im CSV-Format,
- formale Prüfung eines Adresseintrags,

- Bereitstellung von in einem Ressourcenbündel ausgelagerten sprachabhängigen Texten (hier: deutsche und englische Texte)⁷,
- Schließen der Datenbankverbindung.

```

package de.gkjava.addr.controller;

import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.List;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
import java.util.Vector;

import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.filechooser.FileFilter;

import de.gkjava.addr.model.Address;
import de.gkjava.addr.model.Model;
import de.gkjava.addr.persistence.AddressBroker;
import de.gkjava.addr.persistence.ConnectionManager;
import de.gkjava.addr.view.ViewFrame;

public class Controller {
    private Model model;
    private ViewFrame frame;

    private boolean update;
    private int selectedIndex = -1;

    private static final String BUNDLE = "de.gkjava.addr.bundle";
    private ResourceBundle bundle;

    public Controller(Model model) {
        this.model = model;
        bundle = ResourceBundle.getBundle(BUNDLE);
    }

    public void setFrame(ViewFrame frame) {
        this.frame = frame;
    }

    // Erstellt die Tabelle, falls sie nicht existiert und lädt alle Adressen
    public void load() {
        try {
            AddressBroker.getInstance().createTable();
            model.setData();
            frame.getList().setListData(model.getNames());
        } catch (Exception e) {
            JOptionPane.showMessageDialog(frame, e.getMessage(),
                getText("message.error"), JOptionPane.ERROR_MESSAGE);
            System.exit(0);
        }
    }
}

```

⁷ Siehe Kapitel 5.10.

```
// Bereitet die Erfassung einer neuen Adresse vor
public void doNew() {
    frame.getList().clearSelection();
    frame.getEditPanel().clear();
    frame.getEditPanel().enable(true);
    update = false;
}

// Bereitet die Änderung einer Adresse vor
public void doEdit() {
    selectedIndex = frame.getList().getSelectedIndex();
    if (selectedIndex >= 0) {
        Address address = model.get(selectedIndex);
        frame.getEditPanel().setAddress(address);
        frame.getEditPanel().enable(true);
        update = true;
    }
}

// Löscht eine Adresse
public void doDelete() {
    selectedIndex = frame.getList().getSelectedIndex();
    if (selectedIndex >= 0) {
        try {
            int answer = JOptionPane
                .showConfirmDialog(frame,
                    getText("confirm.delete.message"),
                    getText("confirm.delete.title"),
                    JOptionPane.YES_NO_OPTION,
                    JOptionPane.QUESTION_MESSAGE);
            if (answer == JOptionPane.NO_OPTION)
                return;

            int id = model.get(selectedIndex).getId();
            AddressBroker.getInstance().delete(id);
            model.remove(selectedIndex);

            Vector<String> names = model.getNames();
            frame.getList().setListData(names);

            // Selektiert den Eintrag oberhalb des gelöschten Eintrags
            int idx = Math.max(0, selectedIndex - 1);
            frame.getList().setSelectedIndex(idx);

            frame.getEditPanel().clear();
            frame.getEditPanel().enable(false);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(frame, e.getMessage(),
                getText("message.error"), JOptionPane.ERROR_MESSAGE);
        }
    }
}

// Speichert eine neue bzw. geänderte Adresse
public void doSave() {
    Address address = frame.getEditPanel().getAddress();
    if (hasErrors(address)) {
        return;
    }
}
```

```
if (update) {
    if (selectedIndex >= 0) {
        try {
            AddressBroker.getInstance().update(address);
            model.set(selectedIndex, address);
            frame.getList().setListData(model.getNames());

            // Selektiert den Listeneintrag zur geänderten Adresse
            int idx = model.getIndex(address.getId());
            frame.getList().setSelectedIndex(idx);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(frame, e.getMessage(),
                getText("message.error"),
                JOptionPane.ERROR_MESSAGE);
        }
    } else {
        try {
            int newId = AddressBroker.getInstance().insert(address);
            address.setId(newId);
            model.add(address);
            frame.getList().setListData(model.getNames());

            // Selektiert den Listeneintrag zur neuen Adresse
            int idx = model.getIndex(newId);
            frame.getList().setSelectedIndex(idx);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(frame, e.getMessage(),
                getText("message.error"), JOptionPane.ERROR_MESSAGE);
        }
    }
}

frame.getEditPanel().clear();
frame.getEditPanel().enable(false);
}

public void doCancel() {
    frame.getEditPanel().clear();
    frame.getEditPanel().enable(false);
}

// Exportiert alle Adressen im CSV-Format
public void doExport() {
    JFileChooser chooser = new JFileChooser();
    chooser.setSelectedFile(new File(getText("text.filename")));
    chooser.setFileFilter(new Filefilter() {
        public boolean accept(File file) {
            if (file.isDirectory())
                return true;
            if (file.getName().endsWith(".csv"))
                return true;
            else
                return false;
        }

        public String getDescription() {
            return "CSV (*.csv)";
        }
    });
}
```

```
int opt = chooser.showSaveDialog(frame);
if (opt != JFileChooser.APPROVE_OPTION)
    return;

try {
    File file = chooser.getSelectedFile();
    if (file.exists()) {
        JOptionPane.showMessageDialog(frame, file,
            getText("message.fileexists"),
            JOptionPane.ERROR_MESSAGE);
        return;
    }

    export(file);

    JOptionPane.showMessageDialog(frame, file,
        getText("message.filesaved"),
        JOptionPane.INFORMATION_MESSAGE);
} catch (Exception e) {
    JOptionPane.showMessageDialog(frame, e.getMessage(),
        getText("message.error"), JOptionPane.ERROR_MESSAGE);
}
}

private void export(File file) throws Exception {
try (PrintWriter out = new PrintWriter(new FileWriter(file))) {
    String quote = "\"";
    String sep = ";";

    StringBuilder sb = new StringBuilder();
    sb.append(quote + getText("address.id") + quote + sep);
    sb.append(quote + getText("address.lastname") + quote + sep);
    sb.append(quote + getText("address.firstname") + quote + sep);
    sb.append(quote + getText("address.email") + quote + sep);
    sb.append(quote + getText("address.email_additional") + quote + sep);
    sb.append(quote + getText("address.homepage") + quote);
    out.println(sb);

    List<Address> data = model.getData();
    for (Address a : data) {
        sb = new StringBuilder();
        sb.append(quote + a.getId() + quote + sep);
        sb.append(quote + a.getLastname() + quote + sep);
        sb.append(quote + a.getFirstname() + quote + sep);
        sb.append(quote + a.getEmail() + quote + sep);
        sb.append(quote + a.getEmailAdditional() + quote + sep);
        sb.append(quote + a.getHomepage() + quote);
        out.println(sb);
    }
}

// Prüft einen Adresseintrag
public boolean hasErrors(Address address) {
    if (address.getLastname().length() == 0) {
        JOptionPane.showMessageDialog(frame,
            getText("message.lastname.invalid"),
            getText("message.error"), JOptionPane.ERROR_MESSAGE);
        return true;
    }
}
```

```

        return false;
    }

    // Liefert den sprachabhängigen Text
    public String getText(String key) {
        try {
            return bundle.getString(key);
        } catch (MissingResourceException e) {
            return key;
        }
    }

    // Schließt die Datenbank-Verbindung
    public void exit() {
        ConnectionManager.closeConnection();
    }
}

```

Main

Main ist die Start-Klasse der Anwendung. Damit in jedem Fall bei Beendigung des Programms durch den Benutzer oder bei Programmabbruch die Datenbankverbindung geschlossen wird, wird ein so genannter *Shutdown Hook* eingesetzt.⁸

```

package de.gkjava.addr;

import de.gkjava.addr.controller.Controller;
import de.gkjava.addr.model.Model;
import de.gkjava.addr.view.ViewFrame;

public class Main {
    public static void main(String[] args) {
        Model model = new Model();
        final Controller controller = new Controller(model);
        ViewFrame frame = new ViewFrame(controller);

        controller.setFrame(frame);
        controller.load();

        frame.setSize(900, 600);
        frame.setLocation(100, 100);
        frame.setVisible(true);
        frame.getSplitPane().setDividerLocation(0.25);

        // Wird bei Beendigung des Programms ausgeführt
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                controller.exit();
            }
        });
    }
}

```

⁸ Siehe Kapitel 9.4.

14.4.3 Präsentationsschicht

Das Paket `de.gkjava.addr.view` enthält die Klassen der Präsentationsschicht.

ViewFrame

Die Klasse `ViewFrame` bietet den Fensterrahmen für alle anzuzeigenden Komponenten: Symbolleiste, Auswahlliste und Erfassungsformular. Die beiden zuletzt genannten Komponenten sind in einem `JSplitPane` mit vertikalem Trennbalken eingebettet.

```
package de.gkjava.addr.view;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Image;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;

import de.gkjava.addr.controller.Controller;

public class ViewFrame extends JFrame {
    private Controller controller;
    private ViewToolBar toolBar;
    private ViewList list;
    private ViewEditPanel editPanel;
    private JSplitPane splitPane;

    public ViewFrame(Controller controller) {
        this.controller = controller;
        build();
    }

    private void build() {
        Image image = getToolkit().getImage(getClass().getResource("address.png"));
        setIconImage(image);

        setTitle(controller.getText("frame.title"));
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = getContentPane();

        toolBar = new ViewToolBar(controller);
        container.add(toolBar.getToolBar(), BorderLayout.NORTH);

        list = new ViewList(controller);
        editPanel = new ViewEditPanel(controller);
        editPanel.enable(false);

        splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, true,
            new JScrollPane(list.getList()), editPanel.getPanel());
        container.add(splitPane, BorderLayout.CENTER);
    }

    public ViewList getList() {
        return list;
    }
}
```

```
public ViewEditPanel getEditPanel() {
    return editPanel;
}

public JSplitPane getSplitPane() {
    return splitPane;
}

}
```

ViewToolBar

Die Klasse ViewToolBar enthält die Buttons "Neue Adresse eintragen", "Adresse anzeigen", "Adresse löschen" und "Adressen exportieren".

```
package de.gkjava.addr.view;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JToolBar;

import de.gkjava.addr.controller.Controller;

public class ViewToolBar {
    private Controller controller;
    private JToolBar toolBar;
    private JButton newButton;
    private JButton editButton;
    private JButton deleteButton;
    private JButton exportButton;

    public ViewToolBar(Controller controller) {
        this.controller = controller;
        build();
    }

    private void build() {
        toolBar = new JToolBar();
        toolBar.setFloatable(false);

        Icon icon = new ImageIcon(getClass().getResource("new.png"));
        newButton = new JButton(icon);
        newButton.setToolTipText(controller.getText("button.new"));
        newButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                controller.doNew();
            }
        });
        icon = new ImageIcon(getClass().getResource("edit.png"));
        editButton = new JButton(icon);
        editButton.setToolTipText(controller.getText("button.edit"));

    }
}
```

```
editButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        controller.doEdit();
    }
});

icon = new ImageIcon(getClass().getResource("delete.png"));
deleteButton = new JButton(icon);
deleteButton.setToolTipText(controller.getText("button.delete"));
deleteButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        controller.doDelete();
    }
});

icon = new ImageIcon(getClass().getResource("export.png"));
exportButton = new JButton(icon);
exportButton.setToolTipText(controller.getText("button.export"));
exportButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        controller.doExport();
    }
});

toolBar.add(newButton);
toolBar.add(editButton);
toolBar.add(deleteButton);
toolBar.add(exportButton);
}

public JToolBar getToolBar() {
    return toolBar;
}
}
```

ViewList

ViewList enthält die Namensliste zur Auswahl der Adressen. Mit Hilfe des registrierten MouseListener kann durch Doppelklick auf einen Eintrag die jeweilige Adresse zur Anzeige im Formular ausgewählt werden (alternativ zum Drücken des Buttons "Adresse anzeigen").

```
package de.gkjava.addr.view;

import java.awt.Font;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.Vector;

import javax.swing.JList;
import javax.swing.ListSelectionModel;

import de.gkjava.addr.controller.Controller;

public class ViewList {
```

```

private Controller controller;
private JList<String> list;

public ViewList(Controller controller) {
    this.controller = controller;
    build();
}

private void build() {
    list = new JList<String>();
    list.setFont(new Font("SansSerif", Font.PLAIN, 18));
    list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    MouseListener mouseListener = new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
            if (e.getClickCount() == 2) {
                controller.doEdit();
            }
        }
    };
    list.addMouseListener(mouseListener);
}

public JList<String> getList() {
    return list;
}

public void setListData(Vector<String> data) {
    list.setListData(data);
}

public void clearSelection() {
    list.clearSelection();
}

public int getSelectedIndex() {
    return list.getSelectedIndex();
}

public void setSelectedIndex(int idx) {
    list.setSelectedIndex(idx);
}
}

```

ViewEditPanel

ViewEditPanel enthält die Formularfelder korrespondierend zu den Attributen der Klasse Address sowie die Buttons zum Speichern ("OK") und Abbrechen der Aktion.

```

package de.gkjava.addr.view;

import java.awt.BorderLayout;
import java.awt.Desktop;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;

```

```
import java.awt.event.ActionListener;
import java.net.URI;

import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import de.gkjava.addr.controller.Controller;
import de.gkjava.addr.model.Address;

public class ViewEditPanel {
    private Controller controller;
    private int id; // Schlüssel der Adresse

    private JPanel panel;
    private JTextField lastnameField;
    private JTextField firstnameField;
    private JTextField emailField;
    private JTextField emailAdditionalField;
    private JTextField homepageField;
    private JButton saveButton;
    private JButton cancelButton;

    public ViewEditPanel(Controller controller) {
        this.controller = controller;
        build();
    }

    private void build() {
        Font font = new Font("SansSerif", Font.PLAIN, 18);

        JPanel labelPanel = new JPanel(new GridLayout(5, 1, 0, 10));

        JLabel lastnameLabel = new JLabel(
            controller.getText("address.lastname"), JLabel.RIGHT);
        lastnameLabel.setFont(font);
        labelPanel.add(lastnameLabel);

        JLabel firstnameLabel = new JLabel(
            controller.getText("address.firstname"), JLabel.RIGHT);
        firstnameLabel.setFont(font);
        labelPanel.add(firstnameLabel);

        JLabel emailLabel = new JLabel(controller.getText("address.email"),
            JLabel.RIGHT);
        emailLabel.setFont(font);
        labelPanel.add(emailLabel);

        JLabel emailAdditionalLabel = new JLabel(
            controller.getText("address.email_additional"), JLabel.RIGHT);
        emailAdditionalLabel.setFont(font);
        labelPanel.add(emailAdditionalLabel);

        JLabel homepageLabel = new JLabel(
            controller.getText("address.homepage"), JLabel.RIGHT);
        homepageLabel.setFont(font);
        labelPanel.add(homepageLabel);

        JPanel fieldPanel = new JPanel(new GridLayout(5, 1, 0, 10));
```

```
lastnameField = new JTextField();
lastnameField.setFont(font);

firstnameField = new JTextField();
firstnameField.setFont(font);

emailField = new JTextField();
emailField.setFont(font);

emailAdditionalField = new JTextField();
emailAdditionalField.setFont(font);

homepageField = new JTextField();
homepageField.setFont(font);

fieldPanel.add(lastnameField);
fieldPanel.add(firstnameField);
fieldPanel.add(emailField);
fieldPanel.add(emailAdditionalField);
fieldPanel.add(homepageField);

emailField.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        try {
            Desktop.getDesktop().mail(
                new URI("mailto:" + emailField.getText()));
        } catch (Exception e) {
        }
    }
});

emailAdditionalField.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        try {
            Desktop.getDesktop()
                .mail(new URI("mailto:"
                    + emailAdditionalField.getText()));
        } catch (Exception e) {
        }
    }
});

homepageField.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        try {
            Desktop.getDesktop().browse(
                new URI("http://" + homepageField.getText()));
        } catch (Exception e) {
        }
    }
});

JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
saveButton = new JButton(controller.getText("button.save"));
saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        controller.doSave();
    }
});
cancelButton = new JButton(controller.getText("button.cancel"));
```

```
cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        controller.doCancel();
    }
});
buttonPanel.add(saveButton);
buttonPanel.add(cancelButton);

JPanel centerPanel = new JPanel(new BorderLayout(10, 0));
centerPanel.add(labelPanel, BorderLayout.WEST);
centerPanel.add(fieldPanel, BorderLayout.CENTER);

JPanel panel1 = new JPanel(new BorderLayout());
panel1.add(Box.createVerticalStrut(20), BorderLayout.NORTH);
panel1.add(Box.createHorizontalStrut(20), BorderLayout.WEST);
panel1.add(centerPanel, BorderLayout.CENTER);
panel1.add(Box.createHorizontalStrut(20), BorderLayout.EAST);

JPanel panel2 = new JPanel(new BorderLayout());
panel2.add(buttonPanel, BorderLayout.CENTER);
panel2.add(Box.createHorizontalStrut(20), BorderLayout.EAST);
panel2.add(Box.createVerticalStrut(20), BorderLayout.SOUTH);

panel = new JPanel(new BorderLayout());
panel.add(panel1, BorderLayout.NORTH);
panel.add(panel2, BorderLayout.SOUTH);
}

public JPanel getPanel() {
    return panel;
}

// Übertragung Adress-Objekt --> Formularfelder
public void setAddress(Address address) {
    id = address.getId();
    lastnameField.setText(address.getLastname());
    firstnameField.setText(address.getFirstname());
    emailField.setText(address.getEmail());
    emailAdditionalField.setText(address.getEmailAdditional());
    homepageField.setText(address.getHomepage());
}

// Übertragung Formularfelder --> Adress-Objekt
public Address getAddress() {
    Address address = new Address();
    address.setId(id);
    address.setLastname(lastnameField.getText().trim());
    address.setFirstname(firstnameField.getText().trim());
    address.setEmail(emailField.getText().trim());
    address.setEmailAdditional(emailAdditionalField.getText().trim());
    address.setHomepage(homepageField.getText().trim());
    return address;
}

public void clear() {
    lastnameField.setText("");
    firstnameField.setText("");
    emailField.setText("");
    emailAdditionalField.setText("");
    homepageField.setText("");
}
```

```

    public void enable(boolean b) {
        lastnameField.setEnabled(b);
        firstnameField.setEnabled(b);
        emailField.setEnabled(b);
        emailAdditionalField.setEnabled(b);
        homepageField.setEnabled(b);
        saveButton.setEnabled(b);
        cancelButton.setEnabled(b);
    }
}

```

Eine Instanz von `java.awt.Desktop` kann genutzt werden, um den Standard-Mail-Client oder den Standard-Webbrowser mit der Methode `mail` bzw. `browse` aufzurufen. Übergeben wird ein Objekt vom Typ `java.net.URI` (Uniform Resource Identifier).

Mit `setAddress` werden die Attributwerte eines `Address`-Objekts mit Ausnahme des Schlüssels `id` in die Formularfelder übertragen. Der Schlüssel wird im `ViewEditPanel`-Objekt zwischengespeichert. Bei Erfassung einer neuen Adresse hat dieser Schlüssel den Wert 0. Erst bei Einfügen in die Datenbank wird der Schlüsselwert durch die Datenbank erzeugt.

`getAddress` erzeugt aus den Formularfelderinhalten ein neues `Address`-Objekt. Der zwischengespeicherte Schlüssel `id` wird gesetzt.

14.5 Bereitstellung der Anwendung

Zum Schluss werden nun alle Klassen und Ressourcen (Bilder, Ressourcenbündel und `dbparam.txt`) in einem Archiv zusammengefasst.

Voraussetzungen für das Folgende sind:

- Die Bytecodes liegen in Unterverzeichnissen (gemäß Paketstruktur) des Verzeichnisses `bin`.
- `db.properties` liegt im Verzeichnis `bin`.
- Die Button-Bilder liegen im Verzeichnis `bin\de\gkjava\addr\view`.
- Die Dateien `bundle*.properties` des Ressourcenbündels liegen im Verzeichnis `bin\de\gkjava\addr`.

1. Schritt:

Datei `manifest.txt` mit einem Texteditor erstellen. Diese muss die beiden folgenden Zeilen mit Zeilenvorschub am Ende enthalten:

```
Main-Class: de.gkjava.addr.Main
Class-Path: <relativer Pfad des JDBC-Treibers>
```

Beispiel:

Class-Path: lib/h2-1.4.181.jar

2. Schritt:

Erzeugung des Archivs im Verzeichnis P01:

jar cfm addressmanager.jar manifest.txt -C bin .

3. Schritt:

Bereitstellung des JDBC-Treibers gemäß der Angabe im 1. Schritt.

Nun kann die Anwendung durch Doppelklick auf die Datei addressmanager.jar gestartet werden. Natürlich kann hiervon auch eine Verknüpfung auf den Desktop gelegt werden.

Zu beachten ist, dass addressmanager.jar in einem Verzeichnis liegen muss, das den JDBC-Treiber gemäß der obigen Klassenpfad-Angabe enthält.

Das Programm kann in der Konsole (DOS-Fenster) auch wie folgt aufgerufen werden:

java -jar addressmanager.jar

Ist die Voreinstellung der Sprache user.language=de, so werden die englischen Texte angezeigt, wenn man das Programm wie folgt aufruft:

java -Duser.language=en -jar addressmanager.jar

15 Exkurs: Das Java Persistence API

In vorhergehenden Kapiteln wurde JDBC verwendet, um mit Hilfe von SQL auf Datenbanken zuzugreifen. Der Entwickler von JDBC-Programmen muss außer Java-Kenntnissen Know-how im Bereich relationaler Datenbanken und insbesondere SQL haben. Die Datenbanksprache SQL arbeitet mengenorientiert, d. h. die Sprache spezifiziert das Ergebnis (WAS), das man haben will, und nicht den Weg in einzelnen Ablaufschritten zum Ergebnis (WIE).

Verhältnismäßig viel Java-Code wird benötigt, um die Datensätze des Ergebnisses zu verarbeiten und in Java-Objekte zu transformieren. Es besteht ein Strukturbruch (*Impedance Mismatch*) aufgrund der unterschiedlichen Repräsentationsformen von Daten (flache Tabellenstruktur – Java-Objekte).

Lernziele

In diesem Kapitel lernen Sie

- welche Rolle JPA (Java Persistence API) in der Anwendungsentwicklung mit Java spielt,
- wie Objekte so genannter Entity-Klassen (Entities) auf Datensätze in Tabellen relationaler Datenbanken abgebildet werden,
- wie Entities grundlegend aufgebaut sind,
- wie Beziehungen zwischen Entities aufgebaut werden,
- wie Datenbankabfragen mit JPA formuliert werden,
- wie Vererbungshierarchien auf Tabellen einer Datenbank abgebildet werden.

15.1 Einleitung

Das *Java Persistence API (JPA)* ist Teil der Java-EE-Technologie (Java Enterprise Edition), kann aber auch in einer Standard-Java-Umgebung (Java SE), wie wir sie in diesem Buch nutzen, eingesetzt werden. Dieses API vereinfacht in entscheidendem Maße den Zugriff auf Daten in relationalen Datenbanken.

Daten werden aus Tabellen der Datenbank in gewöhnliche Java-Objekte (*POJO = Plain Old Java Objects*) geladen, Veränderungen der Daten in den Objekten werden in der Datenbank gespeichert. Dabei wird für die *Synchronisation zwischen den Daten der Objekte im Programm und der Datenbank* gesorgt.

Die Abbildung zwischen Objekten und Datensätzen in Tabellen einer relationalen Datenbank wird auch als *O/R-Mapping (Object Relational Mapping)* bezeichnet.

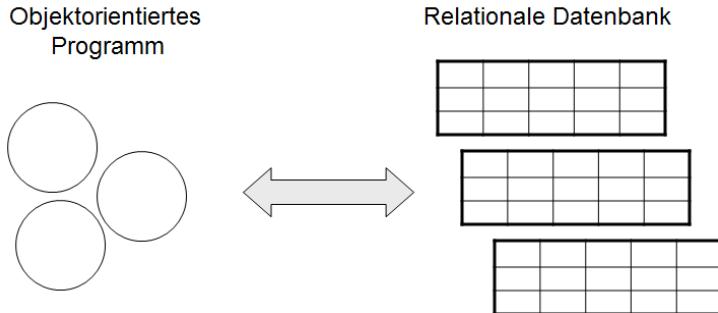
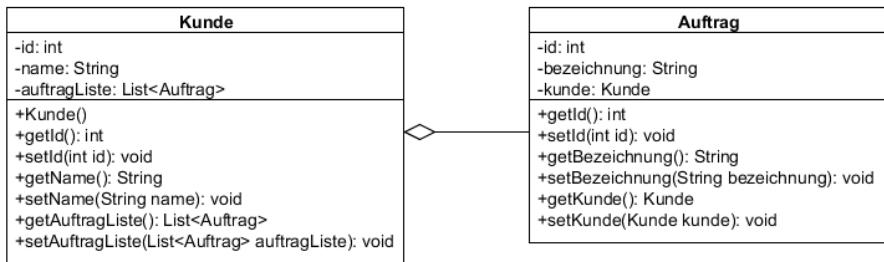


Abbildung 15-1: O/R-Mapping

Im einfachsten Fall ist jeder Klasse eine Tabelle zugeordnet. Jeder Instanz der Klasse entspricht eine Zeile in dieser Tabelle.

Klassendiagramm



Datenbanktabellen

ID	NAME	ID	BEZEICHNUNG	KUNDE_ID
4711	Meier	101	Anlage X	4711
4712	Schmitz	102	Anlage Y	4711
		103	Maschine A	4712
		104	Maschine B	4712

Abbildung 15-2: Klassendiagramm und Tabellen

Beispiel:

Ein Kunde kann mehrere Aufträge erteilen. Abbildung 15-2 zeigt das Klassendiagramm. Die Klasse Kunde verwaltet eine Liste mit Instanzen vom Typ Auftrag. Die Klasse Auftrag enthält als Attribut eine Referenz auf den Auftraggeber vom Typ Kunde.

Abbildung 15-2 zeigt auch die beiden Tabellen `kunde` und `auftrag`. Die Beziehung zwischen den Tabellen wird über die Spalte `kunde_id` hergestellt. So sind beispielsweise dem Kunden 4711 die Aufträge 101 und 102 zugeordnet.

Der folgende Ausschnitt ist Teil eines JDBC-Programms, das eine Liste der Aufträge des Kunden 4711 erzeugt.

JDBC-Programmausschnitt

```
int knr = 4711;

ResultSet rs = stmt.executeQuery("select k.id, k.name, a.id, "
    + "a.bezeichnung from kunde k join auftrag a "
    + "on k.id = a.kunde_id where k.id = " + knr);
while (rs.next()) {
    System.out.println(rs.getInt(1) + " " + rs.getString(2) + " "
        + rs.getInt(3) + " " + rs.getString(4));
}
```

Die gleiche Funktionalität wird mit folgendem Ausschnitt eines Java-Programms, das JPA verwendet, erbracht.

JPA-Programmausschnitt

```
int knr = 4711;

Kunde k = em.find(Kunde.class, knr);
System.out.println(k.getId() + " " + k.getName());

List<Auftrag> auftragListe = k.getAuftragListe();
for (Auftrag a : auftragListe) {
    System.out.println(a.getId() + " " + a.getBezeichnung());
}
```

Hier wird deutlich, dass "rein objektorientiert" gearbeitet werden kann und der oben erwähnte Strukturbrech sich nicht im Code widerspiegelt.

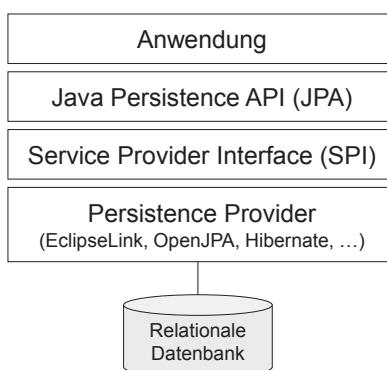


Abbildung 15-3: Persistence API und Provider

Für JPA existieren verschiedene Implementierungen (*Persistence Provider*). Wir nutzen hier die Referenzimplementierung für JPA 2.0 und 2.1: EclipseLink.

15.2 Einrichten der Entwicklungsumgebung

Was wir nutzen

Wir nutzen das relationale Datenbanksystem H2, die Entwicklungsumgebung Eclipse sowie die oben erwähnte Referenzimplementierung *EclipseLink* in der Version 2.5.2. Der grundsätzliche Umgang mit Eclipse für Java-Projekte wird vorausgesetzt.

Alternativen

Selbstverständlich können auch andere Datenbanksysteme verwendet werden. Das Begleitmaterial enthält Skripte zum Starten und Stoppen des H2-Datenbankservers und zum Erstellen und Abfragen einer Datenbank.

Die einzelnen Schritte

Zunächst muss EclipseLink heruntergeladen werden.¹ Durch Entpacken der ZIP-Datei `eclipselink-2.5.2.xxx.zip` entsteht das Verzeichnis `eclipselink`.

In Eclipse kann unter dem Menüpunkt *Preferences* für Java die *User Library* eingerichtet werden, z. B. mit dem Namen *jpa*.

Die JAR-Dateien

`eclipselink.jar` und `javax.persistence_2.1.0.xxx.jar`

aus den Unterverzeichnissen von `eclipselink` sind dann hinzuzufügen, ebenso der JDBC-Treiber für die H2-Datenbank (`h2-xxx.jar`). Der JDBC-Treiber wird nur für die Ausführung der Programme benötigt.

In jedem Java-Projekt muss nun die User Library *jpa* zum *Build Path* hinzugefügt werden.

15.3 Entity-Klassen

Eine *Entity-Klasse* ist eine gewöhnliche Java-Klasse, die Instanzvariablen und get- und set-Methoden enthält. Die Attribute repräsentieren die Spalten einer Datenbanktabelle. Die Klasse muss über einen Standardkonstruktor verfügen und ist zusätzlich mit einigen *Annotationen* ausgestattet.

¹ Siehe Internet-Quellen am Ende des Buches.

Das mit Java SE 5 eingeführte Konzept der Annotationen erlaubt 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 @.

Programm 15.1

Der folgende Code (Projekt P01) definiert die Entity-Klasse Kunde mit den Attributen id und name.

Kunde

```
package entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Kunde {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    public Kunde() {
    }

    public Kunde(String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Die Annotation @Entity kennzeichnet die Klasse Kunde als Entity-Klasse. In diesem Fall hat die zugeordnete Datenbanktabelle den Namen der Klasse Kunde.

`@Entity` hat selbst ein Attribut `name`. Fehlt diese Angabe wie im obigen Beispiel, so gilt der Klassename als Name. Im folgenden Fall wird `KundeEntity` als Tabellenname verwendet, falls keine weiteren Angaben gemacht werden:

```
@Entity(name = "KundeEntity")
```

Im Übrigen wird der Name der Entity in Abfragen mit der *Java Persistence Query Language (JPQL)* verwendet. Zur Festlegung des Namens der zugeordneten Tabelle kann auch direkt die Annotation `@Table` genutzt werden:

```
@Entity
@Table(name = "TBL_KUNDE")
```

Innerhalb der Klasse können die für JPA relevanten Annotationen *entweder alle* direkt zur Instanzvariablen *oder alle* zur entsprechenden getter-Methode vergeben werden. Wir nutzen hier die erste Möglichkeit.

Die Annotation `@Id` kennzeichnet den eindeutigen Schlüssel (Primärschlüssel der Tabelle). Schlüssel können auch mehrere Felder umfassen.²

Schlüsselwerte können automatisch generiert werden. Hierzu wird die Annotation `@GeneratedValue` verwendet. Unterschiedliche Generierungsmethoden stehen zur Verfügung. Bei der im Beispiel benutzten Strategie `IDENTITY` unterstützt die Datenbank die automatisierte Nummernvergabe.

Standardmäßig werden die Namen der Instanzvariablen als Spaltennamen in der Tabelle genutzt. Mit der Annotation `@Column` kann diese Voreinstellung überschrieben werden:

```
@Column(name = "kundenname")
private String name;
```

Sollen Werte bestimmter Instanzvariablen nicht persistent gespeichert werden, so kann das Java-Schlüsselwort `transient` oder die Annotation `@Transient` genutzt werden:

```
@Transient
private int temp;
```

Die zur Entity-Klasse `Kunde` passende Tabelle kann mit der folgenden SQL-Anweisung in der H2-Datenbank `jpa_db1` angelegt werden:

```
create table kunde (
    id integer identity not null,
    name varchar(30),
    primary key(id)
);
```

² Siehe Kapitel 15.7.

Es gibt aber auch die Möglichkeit, dass JPA die Tabelle automatisch anlegt, falls sie nicht bereits existiert (siehe den nächsten Abschnitt). Dabei werden die Java-Datentypen auf entsprechende SQL-Datentypen abgebildet. Die Länge von Zeichenketten wird auf den SQL-Typ VARCHAR mit einer datenbankspezifischen Default-Größe abgebildet. Die Länge kann aber auch mit Hilfe der Column-Annotation explizit vorgegeben werden, z. B. @Column(length = 30).

Entwickelt man eine neue JPA-Anwendung für eine sich bereits im Einsatz befindende Datenbank, kommt man nicht umhin, sich im Programm an die Spaltennamen, -längen usw. der Datenbanktabellen anzupassen. Wir nutzen in dieser Einführung die automatische Generierung.

15.4 Der Entity Manager

Der *Entity Manager* übernimmt alle Aufgaben des Datenbankzugriffs. Hierzu zählen:

- das Speichern in der Datenbank,
- das Löschen aus der Datenbank,
- das Suchen in der Datenbank,
- die Synchronisation zwischen den Objektdaten und der Datenbank.

15.4.1 Persistenzeinheit

Der Begriff *Persistenzeinheit* (*Persistence Unit*) bezeichnet die Zusammenstellung aller Entity-Klassen einer Anwendung. Momentan ist nur die Entity-Klasse `Kunde` zu verwalten. Der Entity Manager ist immer einer Persistenzeinheit zugeordnet.

Die Persistenzeinheit wird in der Datei `persistence.xml` definiert. Sie liegt im Verzeichnis `META-INF`. Zusätzlich sind hier Angaben zum Persistence Provider und zur Datenbank vorhanden.

`persistence.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemalocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="demo" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <properties>
```

```

<property name="javax.persistence.jdbc.url"
          value="jdbc:h2:../../db/data/jpa_db1" />3
<property name="javax.persistence.jdbc.user" value="" />
<property name="javax.persistence.jdbc.password" value="" />

<property name="eclipselink.logging.level" value="SEVERE" />
<property name="eclipselink.ddl-generation" value="create-tables"/>
<property name="eclipselink.create-ddl-jdbc-file-name"
          value="create.sql"/>
<property name="eclipselink.ddl-generation.output-mode" value="both"/>
</properties>
</persistence-unit>
</persistence>

```

Die letzten drei `property`-Einträge legen fest, dass die Tabellen automatisch erstellt werden (falls sie nicht bereits existieren) und dass zusätzlich die SQL-Anweisungen zur Erstellung dieser Tabellen in der Datei `create.sql` gespeichert werden.

Die Persistenzeinheit kann in ein JAR-Archiv verpackt werden. Dieses muss sich dann im Klassenpfad einer JPA-Anwendung befinden.

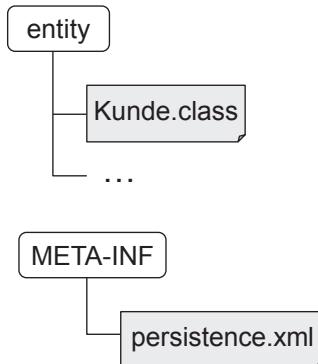


Abbildung 15-4: Persistenzeinheit

15.4.2 Persistenzkontext

Ein *Persistenzkontext* (*Persistence Context*) umfasst alle Instanzen von Entity-Klassen, die vom Entity Manager *zur Laufzeit* verwaltet werden. Ein Persistenzkontext umfasst nur die Instanzen von Entity-Klassen einer einzigen Persistenzeinheit und ist immer einem einzelnen Entity Manager und einer einzigen Datenbank zugeordnet. Innerhalb einer Anwendung kann es zur Laufzeit mehrere Persistenzkontakte geben.

³ Hier ist vorausgesetzt, dass db im Verzeichnis GKJava (siehe Kapitel 1.6) liegt. Es kann aber auch ein absoluter Pfadname oder ~ für das Home-Verzeichnis eingetragen werden.

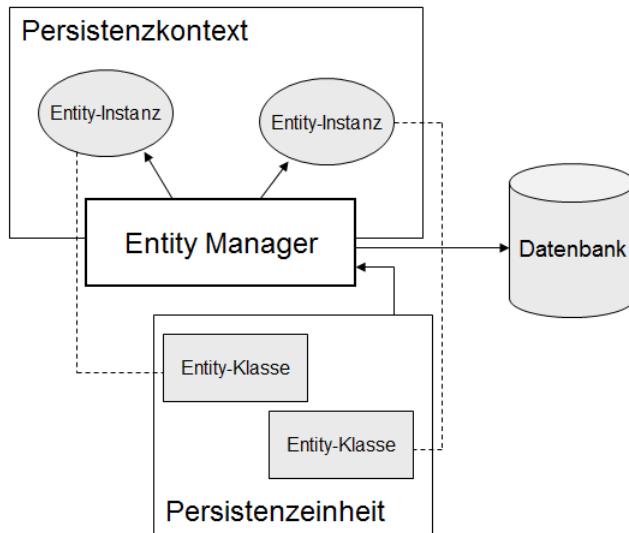


Abbildung 15-5: Entity Manager, Persistenzeinheit und -kontext

15.4.3 Der Lebenszyklus der Entity-Objekte

Der Entity Manager kontrolliert den gesamten *Lebenszyklus* der Entity-Objekte. Verschiedene Zustände können unterschieden werden:

- NEW
Ein Objekt befindet sich in diesem Zustand, wenn es über den Operator `new` erzeugt wurde und noch zu keinem Persistenzkontext gehört.
- MANAGED
Objekte in diesem Zustand befinden sich in einem Persistenzkontext und haben dort eine Identität. Es handelt sich hierbei um Objekte, die gerade erzeugt und gespeichert wurden oder die durch Rekonstruktion bestehender Daten aus der Datenbank beispielsweise über Datenbank-abfragen erzeugt wurden.
- DETACHED
Objekte in diesem Zustand haben eine Identität, aber sie werden zurzeit nicht in einem Persistenzkontext verwaltet.⁴

⁴ Objekte verlieren die Zuordnung zum Entity Manager beispielsweise dann, wenn dieser geschlossen wird oder das Objekt serialisiert und in einen anderen Prozess übertragen wird.

- REMOVED

Objekte in diesem Zustand existieren in einem Persistenzkontext, sind aber zur Löschung aus der Datenbank am Ende einer Transaktion vorgesehen.

Die Zustandsübergänge werden durch die Methoden des Entity Managers ausgelöst.

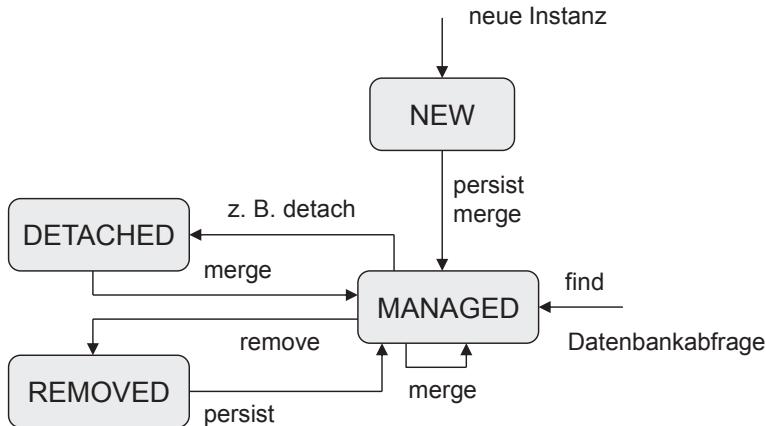


Abbildung 15-6: Lebenszyklus

Die folgenden Beispiele entstammen alle dem Projekt P01.

15.4.4 Erzeugen eines Entity-Objekts

Mit dem folgenden Programm werden drei neue Kunden in der Datenbank jpa_db1 gespeichert.

Create

```

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Kunde;

public class Create {
    private static Kunde[] kunden = new Kunde[] { new Kunde("Schmitz, Anton"),
        new Kunde("Flick, Pit"), new Kunde("Meier, Hugo") };

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();
  
```

```

        em.getTransaction().begin();
        for (Kunde k : kunden) {
            em.persist(k);
            em.flush();
        }
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}

```

Zu Beginn wird ein EntityManager mit Hilfe einer EntityManagerFactory erzeugt. Der Name `demo` ist der Name der Persistenzeinheit. Dieser Name ist in der Datei `persistence.xml` definiert.

Sämtliche Datenbankänderungen passieren in einem Transaktionskontext: `begin()`, `commit()`. Mit der EntityManager-Methode `persist` werden die neu erzeugten Kunde-Objekte in der Datenbank gespeichert:

```
void persist(Object entity)
```

`flush()` bewirkt die sofortige Synchronisation mit der Datenbank. Das wäre hier nicht nötig, da spätestens mit dem Aufruf von `commit()` die Synchronisation stattfindet. Wir nutzen das explizite `flush()`, damit die Nummern für den Schlüssel `id` in der Reihenfolge der einzelnen Speichervorgänge vergeben werden: "Schmitz" erhält die Nummer 1, "Flick" die Nummer 2 und "Meier" die Nummer 3.

Die Datenbanktabelle enthält die neuen Daten. Die ID-Werte wurden automatisch generiert.

KUNDE	
ID	NAME
1	Schmitz, Anton
2	Flick, Pit
3	Meier, Hugo

Abbildung 15-7: Kundendaten

15.4.5 Lesen eines Entity-Objekts

Kunden sollen über ihre ID gesucht werden.

Find

```

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Kunde;

```

```

public class Find {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        for (int i = 1; i <= 4; i++) {
            Kunde k = em.find(Kunde.class, i);
            if (k != null) {
                System.out.println(k.getId() + " " + k.getName());
            } else {
                System.out.println("Kunde " + i + " nicht gefunden");
            }
        }

        em.close();
        emf.close();
    }
}

```

Die `find`-Methode des Entity Managers verwendet zwei Parameter: die Klasse, von deren Typ das Objekt ist und den Primärschlüsselwert des zu suchenden Objekts. Zurückgegeben wird eine Instanz oder `null`.

```
T find(Class<T> entityClass, Object primaryKey)
```

15.4.6 Aktualisieren eines Entity-Objekts

Der Name eines Kunden soll geändert werden.

Update

```

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Kunde;

public class Update {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();
        Kunde k = em.find(Kunde.class, 1);
        if (k != null) {
            k.setName("Klein, Willi");
        } else {
            System.out.println("Kunde nicht gefunden");
        }
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}

```

Nachdem ein Entity-Objekt aus der Datenbank gelesen wurde, befindet es sich im Persistenzkontext und kann mit Hilfe der set-Methoden geändert werden. Mit dem Abschluss der Transaktion erfolgt die Änderung in der Datenbank.

15.4.7 Die Methode merge

Gehört ein Entity-Objekt zu keinem Persistenzkontext, so sorgt die Methode `merge` des Entity Managers dafür, dass das Objekt in den Kontext eingegliedert wird:

```
T merge(T entity)
```

Die von `merge` zurückgegebene Referenz verweist auf das im Persistenzkontext eingegliederte Entity-Objekt.

Gibt es in der Datenbank bereits einen Eintrag zu dem Primärschlüssel des Objekts, wird das Objekt in der Datenbank aktualisiert, andernfalls wird ein neuer Eintrag in der Datenbank aufgenommen.

Merge

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Kunde;

public class Merge {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Kunde k = em.find(Kunde.class, 1);
        k.setName("Klein, Franz");

        em.detach(k); // k ist nun im Zustand DETACHED

        em.getTransaction().begin();
        em.merge(k);

        k = new Kunde("Meier, Emil");
        k.setId(3);
        em.merge(k);

        k = new Kunde("Krause, Otto");
        em.merge(k);

        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}
```

Die Methode `detach` des Entity Managers entfernt ein Entity-Objekt aus dem Persistenzkontext.

Ergebnis der Transaktion: Die Namen der in der Datenbank bereits gespeicherten Kunden mit der ID 1 bzw. 3 wurden geändert, der Kunde "Otto Krause" wurde neu aufgenommen.

KUNDE	
ID	NAME
1	Klein, Franz
2	Flick, Pit
3	Meier, Emil
4	Krause, Otto

Abbildung 15-8: Geänderte Kundendaten

15.4.8 Löschen eines Entity-Objekts

Die Methode `remove` des Entity Managers gibt das Entity-Objekt zum Löschen aus der Datenbank frei:

```
void remove(Object entity)
```

Remove

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Kunde;

public class Remove {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();
        Kunde k = em.find(Kunde.class, 4);
        if (k != null) {
            em.remove(k);
        } else {
            System.out.println("Kunde nicht gefunden");
        }
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}
```

15.5 Entity-Beziehungen

Zwischen den Klassen einer Anwendung können Assoziationen bestehen. Grundlage für die Beispiele dieses Abschnitts ist das folgende Klassendiagramm:

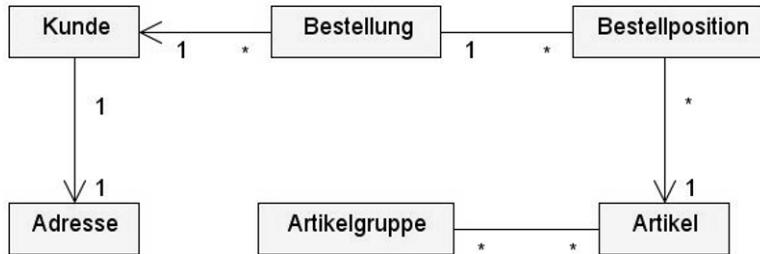


Abbildung 15-9: Entity-Klassen und ihre Beziehungen

- Jeder Kunde hat eine Adresse und jeder Adresse ist ein Kunde zugeordnet.
- Ein Kunde kann mehrere Bestellungen aufgeben. Eine Bestellung ist einem Kunden zugeordnet.
- Eine Bestellung umfasst mehrere Bestellpositionen. Eine Bestellposition ist einer Bestellung zugeordnet.
- Einer Bestellposition ist ein Artikel zugeordnet. Ein Artikel kann in mehreren Bestellpositionen vorkommen.
- Verschiedene Artikel können zu einer Artikelgruppe zusammengefasst werden. Dabei kann ein und derselbe Artikel zu mehreren Gruppen gehören.

Beziehungstypen

JPA unterscheidet bei Beziehungen die Ausprägungen *One-To-One* (1:1), *One-To-Many* (1:n), *Many-To-One* (n:1) und *Many-To-Many* (m:n).

Zusätzlich kann dann noch zwischen *unidirektionalen* und *bidirektionalen* Beziehungen unterschieden werden.

unidirektional

Eine Beziehung zwischen zwei Klassen A und B heißt *unidirektional*, wenn von Objekten nur einer Seite (der Klasse A) auf zugeordnete Objekte der anderen Seite (der Klasse B) zugegriffen werden kann (*Navigation in nur einer Richtung*). Die Klasse A muss dann eine Referenz auf ein Objekt der Klasse B verwalten.

bidirektional

Gilt auch der umgekehrte Fall, dass von Objekten der anderen Seite (Klasse B) auf Objekte der Klasse A zugegriffen werden kann, so spricht man von einer *bidirektionalen* Beziehung.

Die Pfeilspitzen in Abbildung 15-9 deuten die Richtung in einer unidirektionalen Beziehung an.

Programm 15.2

Im Folgenden werden die einzelnen Beziehungstypen am Beispiel vorgestellt (Projekt P02).

15.5.1 OneToOne

Zwischen *Kunde* und *Adresse* besteht eine 1:1-Beziehung. Diese soll ausgehend vom Kunden unidirektional sein.

Adresse

```
package entity;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Adresse {
    @Id
    private int id;
    private String strasse;
    private int plz;
    private String ort;

    public Adresse() {}

    public Adresse(int id, String strasse, int plz, String ort) {
        this.id = id;
        this.strasse = strasse;
        this.plz = plz;
        this.ort = ort;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getStrasse() {
        return strasse;
    }
}
```

```
public void setStrasse(String strasse) {
    this.strasse = strasse;
}

public int getPlz() {
    return plz;
}

public void setPlz(int plz) {
    this.plz = plz;
}

public String getOrt() {
    return ort;
}

public void setOrt(String ort) {
    this.ort = ort;
}

public String toString() {
    return "Adresse [id=" + id + ", strasse=" + strasse + ", plz=" + plz
           + ", ort=" + ort + "]";
}
}
```

Kunde

```
package entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Kunde {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    @OneToOne
    private Adresse adresse;

    public Kunde() {
    }

    public Kunde(String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Adresse getAdresse() {
    return adresse;
}

public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
}

public String toString() {
    return "Kunde [id=" + id + ", name=" + name + "]";
}
}

```

In der Klasse Kunde ist das Attribut `adresse` gegenüber dem Beispiel aus Projekt P01 neu hinzugekommen. Die 1:1-Beziehung wird durch die Annotation `@OneToOne` ausgedrückt.

Die zu den beiden Entity-Klassen passenden Tabellen werden bei Aufruf des Programms `Create1` automatisch in der H2-Datenbank `jpa_db2` angelegt (siehe `persistence.xml`):

```

create table adresse (
    id integer not null,
    strasse varchar,
    plz int not null,
    ort varchar,
    primary key(id)
);

create table kunde (
    id integer identity not null,
    name varchar,
    adresse_id integer,
    primary key(id),
    foreign key (adresse_id) references adresse (id)
);

```

Die Tabelle derjenigen Seite, von der die Beziehung ausgeht (hier `kunde`) enthält den Fremdschlüssel `adresse_id`. Dieser verweist auf den Primärschlüssel der zugeordneten Adresse.

Regel zur Bildung des Fremdschlüsselnamens

Der Fremdschlüssel wird im Allgemeinen gebildet aus:

- Name des Referenzattributs der "führenden" Seite der Beziehung,
- gefolgt von einem Unterstrich,
- gefolgt vom Namen des Primärschlüssels der anderen Seite der Beziehung.

Mit dem folgenden Programm werden Kundendaten mit Adressen in der Datenbank jpa_db2 gespeichert.

Create1

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Adresse;
import entity.Kunde;

public class Create1 {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Kunde[] kunden = new Kunde[] { new Kunde("Meier, Hugo"),
            new Kunde("Schmitz, Anton"), new Kunde("Flick, Pit") };
        Adresse[] adressen = new Adresse[] {
            new Adresse(101, "Hauptstr. 10", 12345, "Hauptdorf"),
            new Adresse(102, "Kastanienallee 1", 11223, "Musterstadt") };

        em.getTransaction().begin();

        for (Adresse a : adressen) {
            em.persist(a);
        }

        for (int i = 0; i < 3; i++) {
            if (i < 2)
                kunden[i].setAdresse(adressen[i]);
            em.persist(kunden[i]);
            em.flush();
        }

        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}
```

Die Inhalte der Tabellen der Datenbank nach Ausführung des Programms zeigt Abbildung 15-10.

KUNDE			ADRESSE			
ID	NAME	ADRESSE_ID	ID	STRASSE	PLZ	ORT
1	Meier, Hugo	101	101	Hauptstr. 10	12345	Hauptdorf
2	Schmitz, Anton	102	102	Kastanienallee 1	11223	Musterstadt
3	Flick, Pit	null				

Abbildung 15-10: Kundendaten mit Adressen

Transitive Persistenz

Wird in der Entity-Klasse `Kunde` die Annotation `@OneToOne` wie folgt ergänzt

```
@OneToOne(cascade = CascadeType.PERSIST)
```

so wird der Aufruf von `persist` an die in Beziehung stehenden Objekte automatisch durchgereicht.

Die for-Schleife im Programm `Create1`

```
for (Adresse a : adressen) {
    em.persist(a);
}
```

ist dann zu streichen.

Ein kaskadierendes Löschen von abhängigen Objekten beim Aufruf von `remove` wird durch `CascadeType.REMOVE` erreicht.

Mehrere Typen können in einer Annotation auftreten:

```
@OneToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
```

Die Annotationen zu allen Beziehungstypen erlauben dieses Durchreichen.

15.5.2 OneToMany und ManyToOne

Die 1:n-Beziehung zwischen `Bestellung` und `Bestellposition` wird bidirektional implementiert. Die n:1-Beziehungen zwischen `Bestellung` und `Kunde` bzw. zwischen `Bestellposition` und `Artikel` wird unidirektional implementiert ausgehend von `Bestellung` bzw. `Bestellposition`.

Zunächst werden Artikel in der Datenbank gespeichert.

Artikel

```
package entity;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
```

```
public class Artikel {  
    @Id  
    private int id;  
    private String bezeichnung;  
    private double preis;  
  
    public Artikel() {}  
  
    public Artikel(int id, String bezeichnung, double preis) {  
        this.id = id;  
        this.bezeichnung = bezeichnung;  
        this.preis = preis;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getBezeichnung() {  
        return bezeichnung;  
    }  
  
    public void setBezeichnung(String bezeichnung) {  
        this.bezeichnung = bezeichnung;  
    }  
  
    public double getPreis() {  
        return preis;  
    }  
  
    public void setPreis(double preis) {  
        this.preis = preis;  
    }  
  
    public String toString() {  
        return "Artikel [id=" + id + ", bezeichnung=" + bezeichnung  
               + ", preis=" + preis + "]";  
    }  
}
```

Die zugehörige Tabelle wird beim Aufruf von Create2 automatisch angelegt:

```
create table artikel (  
    id integer not null,  
    bezeichnung varchar,  
    preis double,  
    primary key(id)  
);
```

Create2

```

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Artikel;

public class Create2 {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Artikel[] artikel = new Artikel[] { new Artikel(4711, "Hammer", 2.9),
            new Artikel(4712, "Zange", 3.),
            new Artikel(4713, "Bohrer", 4.99) };

        em.getTransaction().begin();
        for (Artikel a : artikel) {
            em.persist(a);
        }
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}

```

Abbildung 15-11 zeigt den Inhalt der Tabelle artikel.

ARTIKEL

ID	BEZEICHNUNG	PREIS
4711	Hammer	2.9
4712	Zange	3.0
4713	Bohrer	4.99

Abbildung 15-11: Artikeldaten

Nun kommen wir zu den Entity-Klassen `Bestellung` und `Bestellposition`.

Führende Seite einer 1:n-Beziehung

In einer bidirektionalen 1:n-Beziehung ist die n-Seite führend. In ihrer zugehörigen Tabelle befindet sich der Fremdschlüssel. In der entsprechenden Entity-Klasse (hier `Bestellposition`) wird die Annotation `@ManyToOne` verwendet.

Bestellposition

```
package entity;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Bestellposition {
    @Id
    private int id;
    private int anzahl;
    @ManyToOne
    private Bestellung bestellung;
    @ManyToOne
    private Artikel artikel;

    public Bestellposition() {
    }

    public Bestellposition(int id, int anzahl) {
        this.id = id;
        this.anzahl = anzahl;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getAnzahl() {
        return anzahl;
    }

    public void setAnzahl(int anzahl) {
        this.anzahl = anzahl;
    }

    public Bestellung getBestellung() {
        return bestellung;
    }

    public void setBestellung(Bestellung bestellung) {
        this.bestellung = bestellung;
    }

    public Artikel getArtikel() {
        return artikel;
    }

    public void setArtikel(Artikel artikel) {
        this.artikel = artikel;
    }

    public String toString() {
        return "Bestellposition [id=" + id + ", anzahl=" + anzahl + "]";
    }
}
```

Inverse Seite einer 1:n-Beziehung

Bestellung ist die "inverse" Seite der bidirektionalen Beziehung. Diese Klasse muss eine Liste von Referenzen auf Bestellposition-Objekte verwalten. Die Annotation @OneToMany wird mit dem Attribut mappedBy versehen. Sein Wert ist der Name des mit @ManyToOne versehenen Attributs bestellung der Gegenseite.

Um welche Entity-Klasse es sich hierbei handelt, wird am Parametertyp von `List<Bestellposition>` erkannt.

Bestellung

```
package entity;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Bestellung {
    @Id
    private int id;
    @Temporal(TemporalType.DATE)
    private Date datum;
    @ManyToOne
    private Kunde kunde;
    @OneToMany(mappedBy = "bestellung")
    private List<Bestellposition> bestellpositionListe;

    public Bestellung() {
        bestellpositionListe = new ArrayList<Bestellposition>();
    }

    public Bestellung(int id, Date datum) {
        this();
        this.id = id;
        this.datum = datum;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public Date getDatum() {
        return datum;
    }
}
```

```
public void setDatum(Date datum) {
    this.datum = datum;
}

public Kunde getKunde() {
    return kunde;
}

public void setKunde(Kunde kunde) {
    this.kunde = kunde;
}

public List<Bestellposition> getBestellpositionListe() {
    return bestellpositionListe;
}

public void setBestellpositionListe(
    List<Bestellposition> bestellpositionListe) {
    this.bestellpositionListe = bestellpositionListe;
}

public void add(Bestellposition p) {
    bestellpositionListe.add(p);
    p.setBestellung(this);
}

public String toString() {
    return "Bestellung [id=" + id + ", datum=" + datum + "]";
}
}
```

Sicherstellung der Konsistenz

Das Programm zum Anlegen von Bestellungen muss dafür sorgen, dass beide Seiten der bidirektionalen Beziehung konsistent sind. Um eine neue Bestellposition aufzunehmen, muss einerseits diese in die Liste der Bestellpositionen der Bestellung eingefügt werden, andererseits muss die Bestellposition auf die zugehörige Bestellung verweisen. Dies geschieht hier mit Hilfe der Methode `add`.

Für ein `Date`-Objekt legt die Annotation `@Temporal` fest, welcher SQL-Datentyp in der Datenbank verwendet werden soll. `DATE` steht für das Tagesdatum, `TIME` für die Uhrzeit und `TIMESTAMP` für Datum und Uhrzeit zusammen.

Die folgenden Tabellen werden wiederum automatisch angelegt. Die Bildung der Fremdschlüsselnamen erfolgt nach der weiter oben beschriebenen Regel.

```
create table bestellung (
    id integer not null,
    datum date,
    kunde_id integer,
    primary key(id),
    foreign key (kunde_id) references kunde (id)
);
```

```
create table bestellposition (
    id integer not null,
    anzahl integer,
    bestellung_id integer,
    artikel_id integer,
    primary key(id),
    foreign key (bestellung_id) references bestellung (id),
    foreign key (artikel_id) references artikel (id)
);
```

Mit dem folgenden Programm wird eine Bestellung mit zwei Bestellpositionen aufgenommen.

Create3

```
import java.util.Date;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Artikel;
import entity.Bestellposition;
import entity.Bestellung;
import entity.Kunde;

public class Create3 {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Bestellung b = new Bestellung(1001, new Date());
        Kunde k = em.find(Kunde.class, 1);
        b.setKunde(k);

        Bestellposition p1 = new Bestellposition(1, 5);
        Bestellposition p2 = new Bestellposition(2, 3);

        Artikel a1 = em.find(Artikel.class, 4711);
        Artikel a2 = em.find(Artikel.class, 4712);

        p1.setArtikel(a1);
        p2.setArtikel(a2);

        b.add(p1);
        b.add(p2);

        em.getTransaction().begin();
        em.persist(b);
        em.persist(p1);
        em.persist(p2);
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}
```

Abbildung 15-12 zeigt die eingetragenen Bestelldaten.

BESTELLUNG

ID	DATUM	KUNDE_ID
1001	2014-07-03	1

BESTELLPOSITION

ID	ANZAHL	BESTELLUNG_ID	ARTIKEL_ID
1	5	1001	4711
2	3	1001	4712

Abbildung 15-12: Bestelldaten

Enthält die Entity-Klasse `Bestellung` die ergänzte Annotation

```
@OneToOne(mappedBy = "bestellung", cascade = CascadeType.PERSIST)
```

so wird der Aufruf von `persist` an die Bestellposition-Objekte durchgereicht.

Die beiden Anweisungen

```
em.persist(p1);
em.persist(p2);
```

müssen dann gestrichen werden.

Mit dem folgenden Programm werden alle Daten zur Bestellung 1001 ausgelesen.

Retrieve

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Adresse;
import entity.Bestellposition;
import entity.Bestellung;
import entity.Kunde;

public class Retrieve {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Bestellung b = em.find(Bestellung.class, 1001);
        System.out.println(b);

        Kunde k = b.getKunde();
        System.out.println(k);

        Adresse a = k.getAdresse();
        System.out.println(a);
    }
}
```

```

        List<Bestellposition> bestellpositionListe = b
            .getBestellpositionListe();
        for (Bestellposition p : bestellpositionListe) {
            System.out.println(p);
            System.out.println(p.getArtikel());
        }

        em.close();
        emf.close();
    }
}

```

Ausgabe:

```

Bestellung [id=1001, datum=Thu Jul 03 00:00:00 CEST 2014]
Kunde [id=1, name=Meier, Hugo]
Adresse [id=101, strasse=Hauptstr. 10, plz=12345, ort=Hauptdorf]
Bestellposition [id=1, anzahl=5]
Artikel [id=4711, bezeichnung=Hammer, preis=2.9]
Bestellposition [id=2, anzahl=3]
Artikel [id=4712, bezeichnung=Zange, preis=3.0]

```

15.5.3 ManyToMany

Zwischen Artikel und Artikelgruppe besteht eine m:n-Beziehung. Diese wollen wir bidirektional implementieren.

Für die Bildung von m:n-Beziehungen wird in der Datenbank eine zusätzliche Tabelle benötigt. Diese Tabelle enthält in unserem Beispiel einen Fremdschlüssel zum Verweis auf Artikel und einen Fremdschlüssel zum Verweis auf Artikelgruppen. Wir bestimmen die Klasse `Artikel` als diejenige, die die Beziehung definiert (führende Seite).

Programm 15.3

Der Quellcode dieses Beispiels befindet sich im Projekt P03.

Artikel

```

package entity;

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

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Artikel {
    @Id
    private int id;
}

```

```
private String bezeichnung;
private double preis;
@ManyToMany
private List<Artikelgruppe> artikelgruppeListe;

public Artikel() {
    artikelgruppeListe = new ArrayList<Artikelgruppe>();
}

public Artikel(int id, String bezeichnung, double preis) {
    this();
    this.id = id;
    this.bezeichnung = bezeichnung;
    this.preis = preis;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getBezeichnung() {
    return bezeichnung;
}

public void setBezeichnung(String bezeichnung) {
    this.bezeichnung = bezeichnung;
}

public double getPreis() {
    return preis;
}

public void setPreis(double preis) {
    this.preis = preis;
}

public List<Artikelgruppe> getArtikelgruppeListe() {
    return artikelgruppeListe;
}

public void setArtikelgruppeListe(List<Artikelgruppe> artikelgruppeListe) {
    this.artikelgruppeListe = artikelgruppeListe;
}

public void add(Artikelgruppe ag) {
    artikelgruppeListe.add(ag);
    ag.getArtikelListe().add(this);
}

public String toString() {
    return "Artikel [id=" + id + ", bezeichnung=" + bezeichnung
           + ", preis=" + preis + "]";
}
}
```

Die Klasse Artikelgruppe stellt dann die inverse Seite dar und beinhaltet das Attribut mappedBy der Annotation @ManyToMany.

Artikelgruppe

```
package entity;

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

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Artikelgruppe {
    @Id
    private int id;
    private String name;
    @ManyToMany(mappedBy = "artikelgruppeListe")
    private List<Artikel> artikelListe;

    public Artikelgruppe() {
        artikelListe = new ArrayList<Artikel>();
    }

    public Artikelgruppe(int id, String name) {
        this();
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Artikel> getArtikelListe() {
        return artikelListe;
    }

    public void setArtikelListe(List<Artikel> artikelListe) {
        this.artikelListe = artikelListe;
    }

    public String toString() {
        return "Artikelgruppe [id=" + id + ", name=" + name + "]";
    }
}
```

Die folgende Tabelle wird in der Datenbank jpa_db2 automatisch angelegt:

```
create table artikelgruppe (
    id integer not null,
    name varchar,
    primary key(id)
);
```

Verbindungstabelle

Der Name der Verbindungstabelle ergibt sich aus dem Namen der führenden Seite, gefolgt von einem Unterstrich und dem Namen der inversen Seite. Die Bildung der Fremdschlüsselnamen erfolgt analog zur bekannten Regel.

```
create table artikel_artikelgruppe (
    artikelliste_id integer not null,
    artikelgruppeliste_id integer not null,
    primary key (artikelliste_id, artikelgruppeliste_id),
    foreign key (artikelliste_id) references artikel (id),
    foreign key (artikelgruppeliste_id) references
        artikelgruppe (id)
);
```

Mit dem folgenden Programm werden einige Artikelgruppen angelegt und die bereits vorhandenen Artikel diesen Gruppen zugeordnet.

Create

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Artikel;
import entity.Artikelgruppe;

public class Create {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Artikelgruppe[] gruppen = new Artikelgruppe[] {
            new Artikelgruppe(1, "Gruppe 1"),
            new Artikelgruppe(2, "Gruppe 2") };

        em.getTransaction().begin();

        for (Artikelgruppe ag : gruppen) {
            em.persist(ag);
        }

        Artikel artikel = em.find(Artikel.class, 4711);
        artikel.add(gruppen[0]);
```

```

        artikel = em.find(Artikel.class, 4712);
        artikel.add(gruppen[0]);
        artikel.add(gruppen[1]);

        artikel = em.find(Artikel.class, 4713);
        artikel.add(gruppen[1]);

        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}

```

Abbildung 15-13 zeigt den Inhalt der neuen Tabellen.

ARTIKELGRUPPE		ARTIKEL_ARTIKELGRUPPE	
ID	NAME	ARTIKELLISTE_ID	ARTIKELGRUPPELISTE_ID
1	Gruppe 1	4711	1
2	Gruppe 2	4712	2

ARTIKELLISTE_ID	ARTIKELGRUPPELISTE_ID
4711	1
4712	2
4712	1
4713	2

Abbildung 15-13: Artikelgruppen

Der Name der Verbindungstabelle sowie ihre Spaltennamen werden – wie oben geschildert – nach einer bestimmten Regel gebildet. Diese Namen kann man aber auch selbst vorgeben. Dazu wird die Annotation `@JoinTable` benutzt.

Diese Annotation wird in der Entity-Klasse der führenden Seite eingetragen:

```

@ManyToMany
@JoinTable(name = "artikel_artikelgruppe",
           joinColumns = @JoinColumn(name = "artikel_id"),
           inverseJoinColumns = @JoinColumn(name = "artikelgruppe_id"))
private List<Artikelgruppe> artikelgruppeListe;

```

Mit dem folgenden Programm werden alle Daten zu Artikeln und ihren Gruppen ausgelesen.

Retrieve

```

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Artikel;
import entity.Artikelgruppe;

```

```
public class Retrieve {  
    public static void main(String[] args) {  
        EntityManagerFactory emf = Persistence  
            .createEntityManagerFactory("demo");  
        EntityManager em = emf.createEntityManager();  
  
        int[] agIDs = new int[] { 1, 2 };  
        for (int id : agIDs) {  
            Artikelgruppe ag = em.find(Artikelgruppe.class, id);  
            System.out.println(ag);  
            List<Artikel> artikelliste = ag.getArtikelListe();  
            for (Artikel a : artikelliste) {  
                System.out.println("\t" + a);  
            }  
        }  
  
        System.out.println();  
        int[] artIDs = new int[] { 4711, 4712, 4713 };  
        for (int id : artIDs) {  
            Artikel a = em.find(Artikel.class, id);  
            System.out.println(a);  
            List<Artikelgruppe> artikelgruppeListe = a.getArtikelgruppeListe();  
            for (Artikelgruppe g : artikelgruppeListe) {  
                System.out.println("\t" + g);  
            }  
        }  
  
        em.close();  
        emf.close();  
    }  
}
```

Ausgabe:

```
Artikelgruppe [id=1, name=Gruppe 1]  
    Artikel [id=4711, bezeichnung=Hammer, preis=2.9]  
    Artikel [id=4712, bezeichnung=Zange, preis=3.0]  
Artikelgruppe [id=2, name=Gruppe 2]  
    Artikel [id=4712, bezeichnung=Zange, preis=3.0]  
    Artikel [id=4713, bezeichnung=Bohrer, preis=4.99]  
  
Artikel [id=4711, bezeichnung=Hammer, preis=2.9]  
    Artikelgruppe [id=1, name=Gruppe 1]  
Artikel [id=4712, bezeichnung=Zange, preis=3.0]  
    Artikelgruppe [id=2, name=Gruppe 2]  
    Artikelgruppe [id=1, name=Gruppe 1]  
Artikel [id=4713, bezeichnung=Bohrer, preis=4.99]  
    Artikelgruppe [id=2, name=Gruppe 2]
```

15.6 Abfragen

Mit der *Java Persistence Query Language (JPQL)* liefert JPA eine eigene Abfrage-sprache. Sie ist der Datenbankabfragesprache SQL in vielen Belangen sehr ähnlich. JPQL arbeitet im Gegensatz zu SQL aber nicht auf dem Datenbankschema, sondern auf dem durch die Entity-Klassen und ihren Beziehungen definierten Modell.

Der Quellcode der folgenden Beispiele stammt aus der Klasse `Abfragen` aus Projekt P02.

Der folgende Eintrag in der Datei `persistence.xml` sorgt dafür, dass die von JPA genutzten SQL-Anweisungen protokolliert werden:

```
<property name="eclipselink.logging.level.sql" value="FINE"/>
```

Alle Kunden

Im einfachsten Fall sollen alle Entity-Objekte zu einer Entity-Klasse gelesen werden.

```
String jpql = "select k from Kunde k order by k.name";
Query query = em.createQuery(jpql);
List<Kunde> kunden = query.getResultList();
for (Kunde k : kunden) {
    System.out.println(k);
    Adresse a = k.getAdresse();
    if (a != null)
        System.out.println("\t" + a);
}
```

Der Typ der zurückgegebenen Objekte wird über die Identifikationsvariable `k` bestimmt.

Das Ergebnis ist nach Namen sortiert:

```
Kunde [id=3, name=Flick, Pit]
Kunde [id=1, name=Meier, Hugo]
    Adresse [id=101, strasse=Hauptstr. 10, plz=12345, ort=Hauptdorf]
Kunde [id=2, name=Schmitz, Anton]
    Adresse [id=102, strasse=Kastanienallee 1, plz=11223, ort=Musterstadt]
```

Alle Artikelbezeichnungen

Abfragen können auch auf Attribute eines Entity-Objekts beschränkt werden:

```
String jpql = "select a.bezeichnung from Artikel a "
    + "order by a.bezeichnung";
Query query = em.createQuery(jpql);
List<String> bezeichnungen = query.getResultList();
for (String bez : bezeichnungen) {
    System.out.println(bez);
}
```

Ergebnis:

Bohrer
Hammer
Zange

Alle Artikelnummern und Preise

```
String jpql = "select a.id, a.preis from Artikel a " + "order by a.id";
Query query = em.createQuery(jpql);
List<Object[]> list = query.getResultList();
for (Object[] objs : list) {
    System.out.println(objs[0] + " " + objs[1]);
}
```

Hier liegt jedes Element der Ergebnisliste als Array `Object[]` vor.

Ergebnis:

4711 2.9
4712 3.0
4713 4.99

Alle Kunden mit Adresse

Zwischen Kunde und Adresse besteht eine 1:1-Beziehung.

Version 1:

```
String jpql = "select k from Kunde k join k.adresse a order by k.id";
Query query = em.createQuery(jpql);
List<Kunde> list = query.getResultList();
for (Kunde k : list) {
    System.out.println(k + " " + k.getAdresse());
}
```

Der auf `join` folgende Pfadausdruck `k.adresse` verweist auf das in Beziehung zu `k` stehende Objekt.

Version 2:

```
String jpql = "select k from Kunde k join fetch k.adresse a order by k.id";
Query query = em.createQuery(jpql);
List<Kunde> list = query.getResultList();
for (Kunde k : list) {
    System.out.println(k + " " + k.getAdresse());
}
```

In Version 1 erfordert der Aufruf von `k.getAdresse()` weitere SQL-Anfragen, um die Adressdaten nachzuladen.

In Version 2 werden durch den "Fetch Join" alle benötigten Daten mit der Anfrage sofort geladen (vgl. die protokollierten SQL-Anweisungen).

Das Ergebnis ist in beiden Fällen:

Kunde [id=1, name=Meier, Hugo] Adresse [id=101, strasse=Hauptstr. 10, plz=12345, ort=Hauptdorf]

Kunde [id=2, name=Schmitz, Anton] Adresse [id=102, strasse=Kastanienallee 1, plz=11223, ort=Musterstadt]

Bei beiden Versionen werden nur die Kunden, die eine Adresse haben, ausgegeben.

Alle Kunden ohne Adresse

Möchte man auch Kunden anzeigen, denen keine Adresse zugeordnet ist, muss man `left join` verwenden.

Die folgende Abfrage liefert alle Kunden, die keine Adresse haben (`k.adresse == null`).

```
String jpql = "select k from Kunde k left join k.adresse a "
    + "where a is null order by k.id";
Query query = em.createQuery(jpql);
List<Kunde> list = query.getResultList();
for (Kunde k : list) {
    System.out.println(k);
}
```

Ergebnis:

Kunde [id=3, name=Flick, Pit]

Alle Kunden ohne Bestellung

Mit JPQL können auch Unterabfragen formuliert werden. Mit `exists` wird geprüft, ob die Ergebnismenge einer Unterabfrage leer ist oder nicht. Die folgende Abfrage liefert alle Kunden, für die keine Bestellung existiert.

Version 1:

```
String jpql = "select k from Kunde k where not exists "
    + "(select b from Bestellung b where b.kunde = k) order by k.id";
Query query = em.createQuery(jpql);
List<Kunde> list = query.getResultList();
for (Kunde k : list) {
    System.out.println(k);
}
```

Version 2:

```
String jpql = "select k from Kunde k where k not in "
    + "(select b.kunde from Bestellung b) order by k.id";
Query query = em.createQuery(jpql);
List<Kunde> list = query.getResultList();
for (Kunde k : list) {
    System.out.println(k);
}
```

Mit `in` wird geprüft, ob Objekte in der Ausgabe der Unterabfrage vorkommen.

Das Ergebnis ist in beiden Fällen:

```
Kunde [id=2, name=Schmitz, Anton]
Kunde [id=3, name=Flick, Pit]
```

Preiswerte Artikel

Abfragen können Parameter haben. Die folgende Abfrage liefert alle Artikel, deren Preis einen vorgegebenen Grenzwert nicht übersteigt.

```
String jpql = "select a from Artikel a where "
    + "a.preis <= :maxPreis order by a.bezeichnung";
Query query = em.createQuery(jpql);
query.setParameter("maxPreis", maxPreis);
List<Artikel> artikel = query.getResultList();
for (Artikel a : artikel) {
    System.out.println(a);
}
```

Der Platzhalter `maxPreis` wird zur Laufzeit mittels der Methode `setParameter` mit einem Wert belegt.

Ergebnis:

```
Artikel [id=4711, bezeichnung=Hammer, preis=2.9]
Artikel [id=4712, bezeichnung=Zange, preis=3.0]
```

15.7 Eingebettete Klassen

In den vorhergehenden Beispielen ist der Datentyp des Schlüssels einer Entity-Klasse immer ein einfacher Java-Datentyp. Es können aber auch zusammengesetzte Schlüssel verwendet werden.

Im folgenden Beispiel besteht der Schlüssel der Kundentabelle aus der Kundennummer (`int id`) und der Mandantennummer (`String mandant`).

In der hier unterstellten Anwendung können gleiche Kundennummern bei unterschiedlichen Mandanten vorkommen. Deshalb reicht die Kundennummer alleine nicht zur Identifizierung eines Kunden aus.

Der zusammengesetzte Schlüssel wird in einer eigenen Klasse `KundePK` definiert, die mit der Annotation `@Embeddable` gekennzeichnet ist. Es handelt sich hierbei also um eine Komponente, die in ein Entity eingebettet werden kann. In der Entity-Klasse `Kunde` wird der zusammengesetzte Schlüssel `pk` mit der Annotation `@EmbeddedId` versehen.

Im selben Beispiel wird die Adresse des Kunden nicht als eigenständiges Entity modelliert (wie in den vorhergehenden Beispielen), sondern als einbettbare Komponente (`@Embeddable`).

`@Embedded` in der Klasse `Kunde` legt fest, dass das Attribut `adresse` eine Referenz auf diese Komponente darstellt.

Mit `@AttributeOverrides` und `@AttributeOverride` können die Tabellenspaltennamen zu den Attributen der eingebetteten Klasse abweichend vom Attributnamen vorgegeben werden. Damit kann diese Klasse unverändert in unterschiedlichen Kontexten wiederverwendet werden. Ebenso können hierdurch Namenskonflikte beim Einbetten verhindert werden.

Programm 15.4

Der Quellcode dieses Beispiels befindet sich im Projekt P04.

Kunde

```
package entity;

import javax.persistence.AttributeOverride;
import javax.persistence.AttributeOverrides;
import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
```

```
@Entity
public class Kunde {
    @EmbeddedId
    private KundePK pk;
    private String name;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="strasse", column=@Column(name="adr_strasse")),
        @AttributeOverride(name="plz", column=@Column(name="adr_plz")),
        @AttributeOverride(name="ort", column=@Column(name="adr_ort"))})
    private Adresse adresse;

    public Kunde() {
    }

    public Kunde(KundePK pk, String name, Adresse adresse) {
        this.pk = pk;
        this.name = name;
        this.adresse = adresse;
    }

    public KundePK getPk() {
        return pk;
    }

    public void setPk(KundePK pk) {
        this.pk = pk;
    }
```

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Adresse getAdresse() {
    return adresse;
}

public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
}

public String toString() {
    return "Kunde [pk=" + pk + ", name=" + name + ", adresse=" + adresse
           + "]";
}
}
```

KundePK

```
package entity;

import javax.persistence.Embeddable;

@Embeddable
public class KundePK {
    private int id;
    private String mandant;

    public KundePK() {
    }

    public KundePK(int id, String mandant) {
        this.id = id;
        this.mandant = mandant;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getMandant() {
        return mandant;
    }

    public void setMandant(String mandant) {
        this.mandant = mandant;
    }

    @Override
    public String toString() {
```

```
        return "KundePK [id=" + id + ", mandant=" + mandant + "]";  
    }  
}
```

Adresse

```
package entity;  
  
import javax.persistence.Embeddable;  
  
@Embeddable  
public class Adresse {  
    private String strasse;  
    private int plz;  
    private String ort;  
  
    public Adresse() {}  
  
    public Adresse(String strasse, int plz, String ort) {  
        this.strasse = strasse;  
        this.plz = plz;  
        this.ort = ort;  
    }  
  
    public String getStrasse() {  
        return strasse;  
    }  
  
    public void setStrasse(String strasse) {  
        this.strasse = strasse;  
    }  
  
    public int getPlz() {  
        return plz;  
    }  
  
    public void setPlz(int plz) {  
        this.plz = plz;  
    }  
  
    public String getOrt() {  
        return ort;  
    }  
  
    public void setOrt(String ort) {  
        this.ort = ort;  
    }  
  
    public String toString() {  
        return "Adresse [strasse=" + strasse + ", plz=" + plz + ", ort=" + ort  
               + "]";  
    }  
}
```

Die Entity-Klasse Kunde wird auf die wie folgt definierte Tabelle in der Datenbank jpa_db3 abgebildet:

```
create table kunde (
    id integer not null,
    mandant varchar not null,
    name varchar,
    adr_strasse varchar,
    adr_plz integer,
    adr_ort varchar,
    primary key (id, mandant)
);
```

Create

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Adresse;
import entity.Kunde;
import entity.KundePK;

public class Create {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();

        KundePK pk = new KundePK(4711, "M01");
        Adresse adr = new Adresse("Hauptstr. 10", 12345, "Hauptdorf");
        Kunde k = new Kunde(pk, "Meier", adr);

        em.persist(k);

        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}
```

Read

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Kunde;
import entity.KundePK;

public class Read {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
```

```

EntityManager em = emf.createEntityManager();

KundePK pk = new KundePK(4711, "M01");
Kunde k = em.find(Kunde.class, pk);
System.out.println(k);

em.close();
emf.close();
}
}

```

15.8 Vererbung

JPA kann auch Vererbungshierarchien auf Tabellen der Datenbank abbilden. Dabei können verschiedene Strategien angewendet werden:

- SINGLE_TABLE
Für alle Klassen der Hierarchie wird eine einzige Tabelle erzeugt.
- TABLE_PER_CLASS
Für jede konkrete (d. h. nicht abstrakte Klasse) wird eine eigene Tabelle erzeugt.
- JOINED
Für jede Klasse (abstrakt oder nicht) wird eine eigene Tabelle erzeugt.

In den folgenden Programmbeispielen werden diese Strategien der Reihe nach vorgestellt.

Dabei wird die folgende Vererbungshierarchie verwendet:

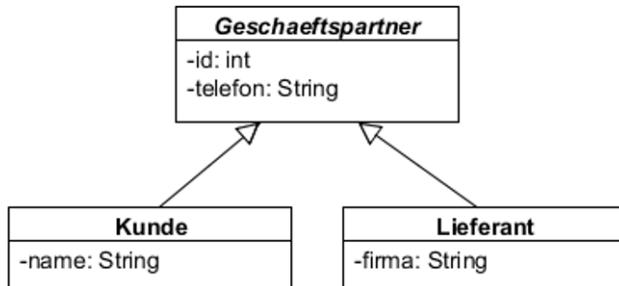


Abbildung 15-14: Vererbungshierarchie

Das Beispiel ist bewusst einfach gehalten, um die wesentlichen Aspekte deutlich hervorheben zu können.

15.8.1 SINGLE_TABLE

Programm 15.5

Der Quellcode dieses Beispiels befindet sich im Projekt P05. Alle Klassen werden auf eine einzige Tabelle abgebildet. Die Superklasse `Geschaeftpartner` hat die Annotation

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
```

Geschaeftpartner

```
package entity;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Geschaeftpartner {
    @Id
    private int id;
    private String telefon;

    public Geschaeftpartner() {
    }

    public Geschaeftpartner(int id, String telefon) {
        this.id = id;
        this.telefon = telefon;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getTelefon() {
        return telefon;
    }

    public void setTelefon(String telefon) {
        this.telefon = telefon;
    }
}
```

Die abgeleiteten Klassen sind jeweils mit einem so genannten *Diskriminator*-Wert gekennzeichnet. Hierbei handelt es sich um einen zusätzlichen Spaltenwert der Tabelle (Spaltenname: `DTYPE`). Anhand dieses Wertes kann der O/R-Mapper entscheiden, zu welchem Entity-Typ eine Zeile der Tabelle gehört. Dieser Wert vom Typ `String` kann für jede konkrete Klasse explizit vorgegeben werden:

```
@DiscriminatorValue("Kunde") bzw. @DiscriminatorValue("Lieferant")
```

Kunde

```
package entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("Kunde")
public class Kunde extends Geschaeftspartner {
    private String name;

    public Kunde() {
    }

    public Kunde(int id, String telefon, String name) {
        super(id, telefon);
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString() {
        return "Kunde [id=" + getId() + ", telefon=" + getTelefon() + ", name="
            + name + "]";
    }
}
```

Lieferant

```
package entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("Lieferant")
public class Lieferant extends Geschaeftspartner {
    private String firma;

    public Lieferant() {
    }

    public Lieferant(int id, String telefon, String firma) {
        super(id, telefon);
        this.firma = firma;
    }

    public String getFirma() {
        return firma;
    }
}
```

```
public void setFirma(String firma) {
    this.firma = firma;
}

public String toString() {
    return "Lieferant [id=" + getId() + ", telefon=" + getTelefon()
           + ", firma=" + firma + "]";
}
}
```

Die Vererbungshierarchie wird auf die folgende Tabelle in der Datenbank jpa_db4 abgebildet:

```
create table geschaeftspartner (
    id integer not null,
    dtype varchar(31),
    telefon varchar,
    name varchar,
    firma varchar,
    primary key (id)
);
```

Create

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Kunde;
import entity.Lieferant;

public class Create {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();

        em.persist(new Kunde(1, "1234", "Mayer"));
        em.persist(new Lieferant(2, "5678", "Schnell & Gut"));
        em.persist(new Kunde(3, "1122", "Schmitz"));
        em.persist(new Lieferant(4, "3344", "ABC GmbH"));

        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}
```

GESCHAEFTSPARTNER

ID	DTYPE	TELEFON	NAME	FIRMA
1	Kunde	1234	Mayer	<i>null</i>
2	Lieferant	5678	<i>null</i>	Schnell & Gut
3	Kunde	1122	Schmitz	<i>null</i>
4	Lieferant	3344	<i>null</i>	ABC GmbH

Abbildung 15-15: Inhalt der Tabelle GESCHAEFTSPARTNER**Read**

```

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

import entity.Geschaeftspartner;
import entity.Kunde;
import entity.Lieferant;

@SuppressWarnings("unchecked")
public class Read {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        System.out.println("\n--- Geschäftspartner ---");
        String jpql = "select g from Geschaeftspartner g order by g.id";
        Query query = em.createQuery(jpql);
        List<Geschaeftspartner> gList = query.getResultList();
        for (Geschaeftspartner g : gList) {
            System.out.println(g);
        }

        System.out.println("\n--- Kunden ---");
        jpql = "select k from Kunde k order by k.id";
        query = em.createQuery(jpql);
        List<Kunde> kList = query.getResultList();
        for (Kunde k : kList) {
            System.out.println(k);
        }

        System.out.println("\n--- Lieferanten ---");
        jpql = "select l from Lieferant l order by l.id";
        query = em.createQuery(jpql);
        List<Lieferant> lList = query.getResultList();
        for (Lieferant l : lList) {
            System.out.println(l);
        }
    }

    em.close();
    emf.close();
}
}

```

Ausgabe des Programms:

```
--- Geschäftspartner ---
Kunde [id=1, telefon=1234, name=Mayer]
Lieferant [id=2, telefon=5678, firma=Schnell & Gut]
Kunde [id=3, telefon=1122, name=Schmitz]
Lieferant [id=4, telefon=3344, firma=ABC GmbH]

--- Kunden ---
Kunde [id=1, telefon=1234, name=Mayer]
Kunde [id=3, telefon=1122, name=Schmitz]

--- Lieferanten ---
Lieferant [id=2, telefon=5678, firma=Schnell & Gut]
Lieferant [id=4, telefon=3344, firma=ABC GmbH]
```

Vorteil dieser Strategie: Will man alle Geschäftspartner auswerten, muss nur eine einzige Abfrage abgesetzt werden.⁵

Nachteil: Bei vielen Attributen sind auch viele Spalten der Tabelle nicht besetzt (NULL-Werte).

15.8.2 TABLE_PER_CLASS

Programm 15.6

Der Quellcode dieses Beispiels befindet sich im Projekt P06. Für jede konkrete Klasse wird eine Tabelle angelegt. Die Superklasse `Geschaeftspartner` hat die Annotation

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
```

Die abgeleiteten Klassen benötigen keine spezielle Annotation. Die Vererbungshierarchie wird auf die folgenden Tabellen in der Datenbank `jpa_db5` abgebildet:

```
create table kunde (
    id integer not null,
    name varchar,
    telefon varchar,
    primary key (id)
);

create table lieferant (
    id integer not null,
    firma varchar,
    telefon varchar,
    primary key (id)
);
```

⁵ Vgl. die protokollierten SQL-Anweisungen.

Die Programme Create und Read entsprechen denen aus dem vorigen Projekt.

KUNDE			LIEFERANT		
ID	NAME	TELEFON	ID	FIRMA	TELEFON
1	Mayer	1234	2	Schnell & Gut	5678
3	Schmitz	1122	4	ABC GmbH	3344

Abbildung 15-16: Inhalt der Tabellen KUNDE und LIEFERANT

Ausgabe des Programms Read:

```
--- Geschäftspartner ---
Kunde [id=1, telefon=1234, name=Mayer]
Kunde [id=3, telefon=1122, name=Schmitz]
Lieferant [id=2, telefon=5678, firma=Schnell & Gut]
Lieferant [id=4, telefon=3344, firma=ABC GmbH]

--- Kunden ---
Kunde [id=1, telefon=1234, name=Mayer]
Kunde [id=3, telefon=1122, name=Schmitz]

--- Lieferanten ---
Lieferant [id=2, telefon=5678, firma=Schnell & Gut]
Lieferant [id=4, telefon=3344, firma=ABC GmbH]
```

Hier fällt auf, dass die Liste der Geschäftspartner nur jeweils innerhalb der Kunden und Lieferanten sortiert ist.

Vorteil dieser Strategie: Die für konkrete Klassen formulierte Abfrage ist performant.

Nachteil: Abfragen über alle Geschäftspartner machen im Hintergrund mehrere SELECT-Anweisungen erforderlich (hier für KUNDE und für LIEFERANT). Wird ein Attribut der Superklasse geändert, müssen alle Tabellen entsprechend angepasst werden.

15.8.3 JOINED

Programm 15.7

Der Quellcode dieses Beispiels befindet sich im Projekt P07. Für jede abstrakte und konkrete Klasse wird eine Tabelle angelegt. Die Superklasse `Geschaeftspartner` hat die Annotation

```
@Inheritance(strategy = InheritanceType.JOINED)
```

Die abgeleiteten Klassen benötigen keine spezielle Annotation. Die Vererbungshierarchie wird auf die folgenden Tabellen in der Datenbank jpa_db6 abgebildet:

```
create table geschaeftpartner (
    id integer not null,
    dtype varchar(31),
    telefon varchar,
    primary key (id)
);

create table kunde (
    id integer not null,
    name varchar,
    primary key (id),
    foreign key (id) references geschaeftpartner (id)
);

create table lieferant (
    id integer not null,
    firma varchar,
    primary key (id),
    foreign key (id) references geschaeftpartner (id)
);
```

Die Programme Create und Read entsprechen denen aus dem vorigen Projekt.

GESCHAEFTSPARTNER

ID	DTYPE	TELEFON
1	Kunde	1234
2	Lieferant	5678
3	Kunde	1122
4	Lieferant	3344

KUNDE

ID	NAME
1	Mayer
3	Schmitz

LIEFERANT

ID	FIRMA
2	Schnell & Gut
4	ABC GmbH

Abbildung 15-17: Inhalt der Tabellen GESCHAEFTSPARTNER, KUNDE und LIEFERANT

Ausgabe des Programms Read:

```
--- Geschäftspartner ---
Kunde [id=1, telefon=1234, name=Mayer]
Lieferant [id=2, telefon=5678, firma=Schnell & Gut]
Kunde [id=3, telefon=1122, name=Schmitz]
Lieferant [id=4, telefon=3344, firma=ABC GmbH]

--- Kunden ---
Kunde [id=1, telefon=1234, name=Mayer]
Kunde [id=3, telefon=1122, name=Schmitz]

--- Lieferanten ---
Lieferant [id=2, telefon=5678, firma=Schnell & Gut]
Lieferant [id=4, telefon=3344, firma=ABC GmbH]
```

Vorteil dieser Strategie: Die Daten der Tabellen sind redundanzfrei gespeichert. Im Gegensatz zur Strategie SINGLE_TABLE können die Tabellenspalten auch Pflichteingaben verlangen (NOT NULL).

Nachteil: Abfragen über alle Geschäftspartner sind in der Regel weitaus komplexer als bei den anderen Strategien (vgl. die protokollierten SQL-Anweisungen) und wirken sich bei größeren Hierarchien ungünstig auf die Performance aus.

15.9 Lebenszyklusmethoden

In Entity-Klassen können so genannte Callback-Methoden definiert werden, die jedesmal dann automatisch aufgerufen werden, wenn sich der Zustand der Entität im Lebenszyklus ändert.

Hierzu müssen diese Methoden mit Annotationen versehen werden:

```
@PrePersist
@PostPersist
@PostLoad
@PreUpdate
@PostUpdate
@PreRemove
@PostRemove
```

Die Bezeichnungen sind selbsterklärend.

So wird beispielsweise die mit @PrePersist annotierte Methode aufgerufen, bevor die neue Entität in der Datenbank angelegt wird. Die mit @PostPersist annotierte Methode wird aufgerufen, wenn die neue Entität in der Datenbank gespeichert wurde.

Die Callback-Methoden haben einen frei wählbaren Namen, den Rückgabetyp void und keine Parameter.

Programm 15.8 (Projekt P08) zeigt ein Beispiel. Die Callback-Methoden protokollieren ihren Aufruf. Bei @PrePersist und @PreUpdate wird der Zeitstempel (changedDate) aktualisiert.

Programm 15.8

```
package entity;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.PostLoad;
import javax.persistence.PostPersist;
import javax.persistence.PostRemove;
import javax.persistence.PostUpdate;
import javax.persistence.PrePersist;
import javax.persistence.PreRemove;
import javax.persistence.PreUpdate;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class User {
    @Id
    private String name;
    private String password;
    @Temporal(TemporalType.TIMESTAMP)
    private Date changedDate;

    public User() {
    }

    public User(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

```
public Date getChangedDate() {
    return changedDate;
}

public void setChangedDate(Date changedDate) {
    this.changedDate = changedDate;
}

@Override
public String toString() {
    return "User [name=" + name + ", password=" + password
           + ", changedDate=" + changedDate + "]";
}

@PrePersist
public void onPrePersist() {
    System.out.println("onPrePersist");
    changedDate = new Date();
}

@PostPersist
public void onPostPersist() {
    System.out.println("onPostPersist");
}

@PostLoad
public void onPostLoad() {
    System.out.println("onPostLoad");
}

@PreUpdate
public void onPreUpdate() {
    System.out.println("onPreUpdate");
    changedDate = new Date();
}

@PostUpdate
public void onPostUpdate() {
    System.out.println("onPostUpdate");
}

@PreRemove
public void onPreRemove() {
    System.out.println("onPreRemove");
}

@PostRemove
public void onPostRemove() {
    System.out.println("onPostRemove");
}
}

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.User;

public class Create {
```

```
public static void main(String[] args) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("demo");

    EntityManager em = emf.createEntityManager();

    em.getTransaction().begin();
    em.persist(new User("hugo", "oguh"));
    em.getTransaction().commit();

    em.close();
    emf.close();
}

}

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.User;

public class Update {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");

        EntityManager em = emf.createEntityManager();

        User user = em.find(User.class, "hugo");
        System.out.println(user);

        em.getTransaction().begin();
        user.setPassword("hugo123");
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.User;

public class Remove {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");

        EntityManager em = emf.createEntityManager();

        User user = em.find(User.class, "hugo");
        System.out.println(user);

        em.getTransaction().begin();
```

```
    em.remove(user);
    em.getTransaction().commit();

    em.close();
    emf.close();
}
}
```

Programm 15.9

Programm 15.9 (Projekt P09) zeigt, wie der Zeitstempel in verschiedenen Entity-Klassen aktualisiert werden kann, ohne dieselben Callback-Methoden in allen Klassen zu implementieren. Hierzu wird ein spezieller Listener implementiert.

```
package entity;

import java.util.Date;

import javax.persistence.PrePersist;
import javax.persistence.PreUpdate;

public class DateEntityListener {
    @PrePersist
    public void onPrePersist(DateEntity entity) {
        entity.changedDate = new Date();
    }

    @PreUpdate
    public void onPreUpdate(DateEntity entity) {
        entity.changedDate = new Date();
    }
}
```

Jede Methode erhält die Entität als Parameter vom Typ `DateEntity`.

Um diesen Listener beispielsweise für die Klasse `User` einsetzen zu können, muss diese von der Klasse `DateEntity` abgeleitet sein.

```
package entity;

import java.util.Date;

import javax.persistence.MappedSuperclass;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@MappedSuperclass
public abstract class DateEntity {
    @Temporal(TemporalType.TIMESTAMP)
    protected Date changedDate;
```

```
public Date getChangedDate() {
    return changedDate;
}

public void setChangedDate(Date changedDate) {
    this.changedDate = changedDate;
}
```

DateEntity ist mit @MappedSuperclass annotiert.

Im Unterschied zu den Vererbungsstrategien in Kapitel 15.8 wird eine so annotierte Klasse selbst nicht in der Datenbank repräsentiert. Attribute werden in den Tabellen ihrer Subklassen abgebildet.

In der Klasse User ist der Listener mithilfe der Annotation

```
@EntityListeners(DateEntityListener.class)
```

registriert.

```
package entity;

import javax.persistence.Entity;
import javax.persistence.EntityListeners;
import javax.persistence.Id;

@Entity
@EntityListeners(DateEntityListener.class)
public class User extends DateEntity {
    @Id
    private String name;
    private String password;

    public User() {}

    public User(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

```
    @Override
    public String toString() {
        return "User [name=" + name + ", password=" + password
               + ", changedDate=" + changedDate + "]";
    }
}
```

Das Attribut `changedDate` wird nun bei jeder Änderung aktualisiert.

15.10 Optimistisches Sperren in Multi-User-Anwendungen

Arbeiten mehrere Benutzer mit derselben Datenbank, kann es vorkommen, dass zwei Benutzer gleichzeitig dasselbe Objekt verändern und speichern. Das kann zu ungewollten Reaktionen führen.

Programm 15.10

Programm 15.10 (UpdateTest im Projekt P10) ist ein Dialogprogramm, mit dem der Benutzer für einen Artikel einen neuen Preis speichern kann.

```
package entity;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Artikel {
    @Id
    private int id;
    private String bezeichnung;
    private double preis;

    public Artikel() {
    }

    public Artikel(int id, String bezeichnung, double preis) {
        this.id = id;
        this.bezeichnung = bezeichnung;
        this.preis = preis;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getBezeichnung() {
        return bezeichnung;
    }
```

```
public void setBezeichnung(String bezeichnung) {
    this.bezeichnung = bezeichnung;
}

public double getPreis() {
    return preis;
}

public void setPreis(double preis) {
    this.preis = preis;
}

public String toString() {
    return "Artikel [id=" + id + ", bezeichnung=" + bezeichnung
           + ", preis=" + preis + "]";
}
}

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Artikel;

public class Create {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Artikel[] artikel = new Artikel[] { new Artikel(4711, "Hammer", 2.9),
                                             new Artikel(4712, "Zange", 3.),
                                             new Artikel(4713, "Bohrer", 4.99) };

        em.getTransaction().begin();
        for (Artikel a : artikel) {
            em.persist(a);
        }
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Artikel;

public class UpdateTest {
    public static void main(String[] args) throws IOException {
        EntityManagerFactory emf = Persistence
```

```
.createEntityManagerFactory("demo");
EntityManager em = emf.createEntityManager();

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String input;

while (true) {
    input = read("Artikelnummer: ", in);
    if (input.equals("q")) {
        break;
    }

    int id = 0;
    try {
        id = Integer.parseInt(input);
    } catch (NumberFormatException e) {
        System.out.println("Die Artikelnummer muss ganzzahlig sein.");
        continue;
    }

    Artikel artikel = em.find(Artikel.class, id);
    if (artikel == null) {
        System.out.println("Artikel nicht gefunden.");
        continue;
    }
    System.out.println(artikel);

    input = read("Neuer Preis: ", in);
    if (input.equals("q")) {
        break;
    }

    double preis = 0;
    try {
        preis = Double.parseDouble(input);
    } catch (NumberFormatException e) {
        System.out.println("Hier muss eine Zahl eingegeben werden.");
        continue;
    }

    input = read("Update (j/n): ", in);
    if (!input.equals("j"))
        continue;

    em.getTransaction().begin();
    artikel.setPreis(preis);
    em.getTransaction().commit();
}

em.close();
emf.close();
}

private static String read(String prompt, BufferedReader in)
    throws IOException {
    System.out.print(prompt);
    return in.readLine();
}
```

Beispiel:

```
Artikelnummer: 4711
Artikel [id=4711, bezeichnung=Hammer, preis=2.9]
Neuer Preis: 3.
Update (j/n): j
Artikelnummer: q
```

Das Programm läuft in einer Schleife, die durch Eingabe von "q" beendet werden kann.

Dieses Programm soll nun von zwei Benutzern zur selben Zeit gestartet werden. Hierzu muss das Datenbanksystem H2 im Server-Modus laufen.⁶

Der Eintrag zum URL der Datenbank in der Datei persistence.xml ist entsprechend angepasst:

```
<property name="javax.persistence.jdbc.url"
          value="jdbc:h2:tcp://localhost/jpa_db7" />
```

Folgendes Ablaufszenario zeigt, dass die Änderung von Benutzer 2 durch die Änderung des Benutzers 1 überschrieben wird (*Last Commit Wins*).

Der Ablauf kann nachvollzogen werden, indem zunächst mit Create Artikeldaten eingefügt werden und dann das Programm UpdateTest jeweils in einer eigenen Konsole gestartet wird. Die Eingaben müssen dann in der wie folgt notierten zeitlichen Reihenfolge erfolgen:

	Benutzer 1	Benutzer 2
1	Artikelnummer: 4711	
2		Artikelnummer: 4711
3		Neuer Preis: 3.
4		Update (j/n): j
5		Artikelnummer: q
6	Neuer Preis: 3.5	
7	Update (j/n): j	
8	Artikelnummer: q	

In der Datenbank ist nun der Preis 3.5 gespeichert.

⁶ Das Start-Skript befindet sich im Verzeichnis db des Begleitmaterials.

Eine sinnvolle Reaktion des Programms wäre, dass Benutzer 1 auf den Konflikt hingewiesen wird und seine Änderung nicht berücksichtigt wird. Er hat dann die Möglichkeit eine neue Transaktion zu starten.

JPA bietet hierfür die Möglichkeit des *optimistischen Sperrens* (optimistic locking). Hierzu erhält die Entity-Klasse Artikel ein neues Attribut, in dem die Version des Objekts gespeichert wird. Das Attribut wird mit der Annotation `@Version` versehen:

```
@Version  
private int version;
```

Optimistisch nennt man dieses Sperren deswegen, weil man einfach annimmt, dass alles gut geht, und löst möglicherweise auftretende Probleme später.

Programm 15.11⁷

```
package entity;  
  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Version;  
  
@Entity  
public class Artikel {  
    @Id  
    private int id;  
    private String bezeichnung;  
    private double preis;  
    @Version  
    private int version;  
  
    public Artikel() {  
    }  
  
    public Artikel(int id, String bezeichnung, double preis) {  
        this.id = id;  
        this.bezeichnung = bezeichnung;  
        this.preis = preis;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getBezeichnung() {  
        return bezeichnung;  
    }
```

⁷ Siehe Projekt P11.

```
public void setBezeichnung(String bezeichnung) {
    this.bezeichnung = bezeichnung;
}

public double getPreis() {
    return preis;
}

public void setPreis(double preis) {
    this.preis = preis;
}

public int getVersion() {
    return version;
}

public void setVersion(int version) {
    this.version = version;
}

public String toString() {
    return "Artikel [id=" + id + ", bezeichnung=" + bezeichnung
           + ", preis=" + preis + "]";
}

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import entity.Artikel;

public class Create {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        Artikel[] artikel = new Artikel[] { new Artikel(4711, "Hammer", 2.9),
                                             new Artikel(4712, "Zange", 3.),
                                             new Artikel(4713, "Bohrer", 4.99) };

        em.getTransaction().begin();
        for (Artikel a : artikel) {
            em.persist(a);
        }
        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
```

```
import javax.persistence.OptimisticLockException;
import javax.persistence.Persistence;

import entity.Artikel;

public class UpdateTestOpt {
    public static void main(String[] args) throws IOException {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("demo");
        EntityManager em = emf.createEntityManager();

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String input;

        while (true) {
            input = read("Artikelnummer: ", in);
            if (input.equals("q")) {
                break;
            }

            int id = 0;
            try {
                id = Integer.parseInt(input);
            } catch (NumberFormatException e) {
                System.out.println("Die Artikelnummer muss ganzzahlig sein.");
                continue;
            }

            Artikel artikel = em.find(Artikel.class, id);
            if (artikel == null) {
                System.out.println("Artikel nicht gefunden.");
                continue;
            }
            System.out.println(artikel);

            input = read("Neuer Preis: ", in);
            if (input.equals("q")) {
                break;
            }

            double preis = 0;
            try {
                preis = Double.parseDouble(input);
            } catch (NumberFormatException e) {
                System.out.println("Hier muss eine Zahl eingegeben werden.");
                continue;
            }

            input = read("Update (j/n): ", in);
            if (!input.equals("j"))
                continue;

            try {
                em.getTransaction().begin();
                artikel.setPreis(preis);
                em.getTransaction().commit();
            } catch (Exception e) {
                if (isOptimisticLockException(e))
                    System.out.println("Versionskonflikt");
            }
        }
    }
}
```

```
        em.close();
        emf.close();
    }

    private static String read(String prompt, BufferedReader in)
        throws IOException {
        System.out.print(prompt);
        return in.readLine();
    }

    public static boolean isOptimisticLockException(Throwable t) {
        while (t != null) {
            if (t instanceof OptimisticLockException)
                return true;
            t = t.getCause();
        }
        return false;
    }
}
```

Wer zuerst kommt, mahlt zuerst.

Das folgende Verfahren liegt zugrunde:

Die Version des Objekts in der Transaktion wird beim Update mit der Version in der Datenbank verglichen. Sind beide Versionen gleich, wird die Version automatisch inkrementiert und das Objekt gespeichert. Stimmt die Version des Objekts nicht mit dem Wert in der Datenbank überein, muss eine andere Transaktion diesen Wert inzwischen geändert haben. Die Transaktion löst die Ausnahme `OptimisticLockException` aus.

Beispiel:

Es wird Artikel 4711 geladen:

```
SELECT ID, BEZEICHNUNG, PREIS, VERSION FROM ARTIKEL WHERE ID = 4711
```

Die Spalte `VERSION` hat den Wert 1.

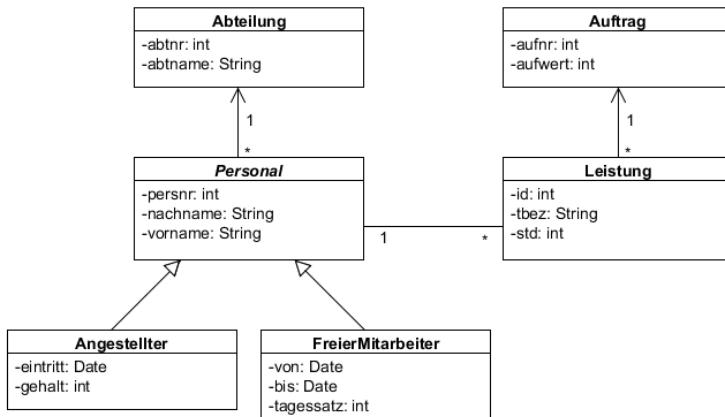
Ein Update erfolgt nur, wenn sich die Version in der Datenbank zwischenzeitlich nicht geändert hat:

```
UPDATE ARTIKEL SET PREIS = 3., VERSION = 2 WHERE ID = 4711 AND VERSION = 1
```

Wiederholt man nun dasselbe Szenario mit Programm `UpdateTestOpt`, sieht man, dass Benutzer 1 auf den Versionskonflikt aufmerksam gemacht wird. In der Datenbank ist der Preis 3. gespeichert.

15.11 Aufgaben

1. In einer Firma werden für Mitarbeiter Leistungssätze zu Aufträgen gespeichert. Zu einer Abteilung werden Abteilungsnummer (Schlüssel) und Abteilungsname gespeichert, zu einem Auftrag Auftragsnummer (Schlüssel) und Auftragswert, zu einem Leistungssatz ID (Schlüssel), Tätigkeitsbezeichnung und Anzahl geleisteter Stunden. Im Personalbestand wird zwischen Angestellten und freien Mitarbeitern unterschieden. Neben der Personalnummer (Schlüssel), Vor- und Nachnamen wird zu einem Angestellten Eintrittsdatum und Gehalt, zu einem freien Mitarbeiter Vertragszeitraum (von, bis) und Tagessatz gespeichert.



- a) Erstellen Sie die Entity-Klassen zu dem in der Abbildung gezeigten Klassendiagramm. Verwenden Sie die Vererbungsstrategie **SINGLE_TABLE**. Zwischen Personal und Abteilung und zwischen Leistung und Auftrag existiert eine unidirektionale Beziehung. Personal und Leistung sind bidirektional miteinander verbunden. Die ID im Leistungssatz soll vom Datenbanksystem vergeben werden.
- b) Schreiben Sie ein Programm, das alle Tabellen mit Testdaten versorgt. Die Tabellen sollen von JPA automatisch generiert werden.
- c) Erzeugen Sie eine Liste der Angestellten und eine Liste der freien Mitarbeiter jeweils nach Nachname und Vorname sortiert.
- d) Schreiben Sie ein Programm, das für einen bestimmten Mitarbeiter das Gehalt ändert.
- e) Schreiben Sie ein Programm, das einen neuen Leistungssatz für einen Mitarbeiter einträgt.

- f) Schreiben Sei ein Programm, das für alle Mitarbeiter die Leistungssätze nach Auftragsnummern sortiert ausgibt.

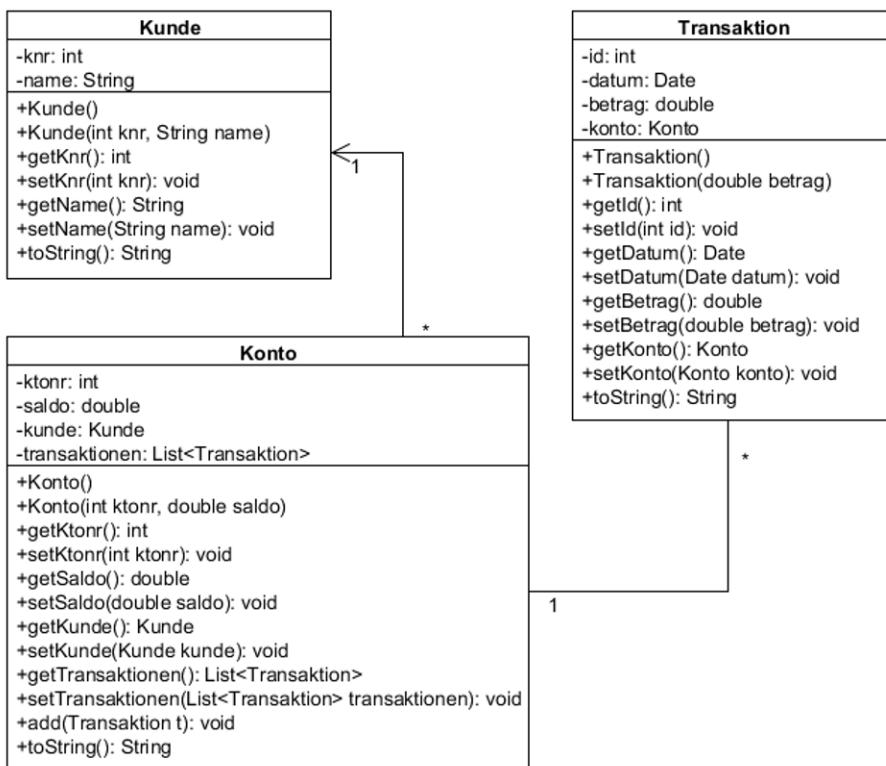
Beispiel:

```
1002 Volt, Gerda
    1111    4 Kalkulation
    2222    7 Kalkulation
    4444    8 Vorbesprechung
```

usw.

- g) Ermitteln Sie diejenigen Mitarbeiter, für die keine Leistungssätze existieren. Nutzen Sie eine JPQL-Anweisung mit `where`-Klausel, die die Liste der Leistungssätze (Attribut in Personal) auf "leer" prüft (`is empty`). Alternativ kann eine Unterabfrage wie in Kapitel 15.6 verwendet werden.

2. Implementieren Sie die folgenden Entity-Klassen mit ihren Beziehungen.



Konto-Methode add:

```
public void add(Transaktion t) {
    transaktionen.add(t);
    t.setKonto(this);
    saldo += t.getBetrag();
}
```

Erstellen Sie die vier Klassen Create, CreateTransaktion, Konten und Transaktionen.

Create: Es wird ein Kunde mit zwei Konten angelegt.

CreateTransaktion: Zu einer Kontonummer wird eine Transaktion angelegt, die den Kontostand entsprechend verändert. Schreiben Sie das Programm so, dass Kontonummer und Betrag der Transaktion variabel sind.

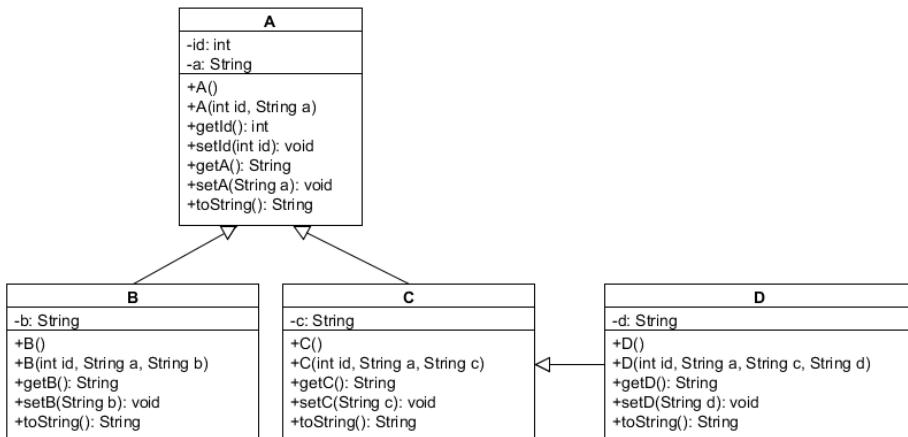
Konten: Auswertung mit JPQL, nach Kontonummer sortiert, z. B.

```
Konto [ktonr=1234, saldo=400.0]
Konto [ktonr=6789, saldo=300.0]
```

Transaktionen: Auswertung mit JPQL, nach Datum absteigend sortiert, z. B.

```
Transaktion [id=3, datum=03.07.2014 15:56:11, betrag=300.0]
    Kontonummer: 6789
Transaktion [id=2, datum=03.07.2014 15:55:52, betrag=200.0]
    Kontonummer: 1234
Transaktion [id=1, datum=03.07.2014 15:55:33, betrag=100.0]
    Kontonummer: 1234
```

3. Gegeben sei das folgende Klassendiagramm:



Testen Sie die drei verschiedenen Vererbungsstrategien SINGLE_TABLE, TABLE_PER_CLASS und JOINED. Die Annotation @Inheritance soll jeweils für die nicht-abstrakte Klasse A erfolgen. Speichern Sie jeweils ein Objekt der vier Klassen und schauen Sie sich die generierten Tabellen an.

16 Exkurs: Die Objektdatenbank db4o

Im Kapitel 15 haben wir im Rahmen des *Java Persistence API* ein *O/R-Mapping-Tool* verwendet, um Java-Objekte auf Datensätze in Tabellen einer relationalen Datenbank abzubilden. *Objektorientierte Datenbanken* können Objekte direkt – ohne Umweg über eine objektrelationale Abbildung – speichern und zurückgeben.

Dieses Kapitel enthält eine Einführung in den Umgang mit dem objektorientierten Datenbanksystem *db4o*.

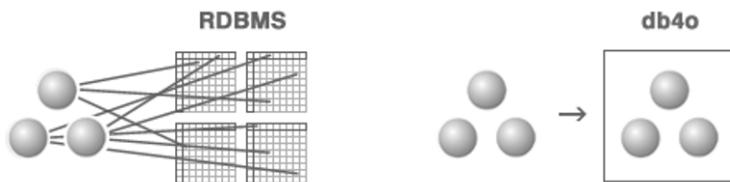


Abbildung 16-1: O/R-Mapping und db4o

Lernziele

In diesem Kapitel lernen Sie

- was Objektorientierung eines Datenbanksystems bedeutet,
- den Umgang mit dem Datenbanksystem db4o zur Speicherung (Einfügen, Ändern, Löschen) und Abfrage von Objekten.

16.1 Einleitung

Ein *objektorientiertes Datenbankmanagementsystem* (*ODBMS*) verwaltet Datenbanken (so genannte *Objektdatenbanken*), in denen Daten als Objekte sowie deren Beziehungen untereinander gespeichert sind.

Kriterien für ein ODBMS

Wichtige Anforderungen an solche Systeme sind:¹

¹ Siehe auch <http://de.wikipedia.org/wiki/Objektdatenbank>.

- die dauerhafte Speicherung von Objekten,
- die Verwaltung komplexer Objekte (*Objektgraphen*),
- die eindeutige Identifizierung von Objekten (*Objektidentität*),
- die Unterstützung des Prinzips der Kapselung (Zugriff auf den Objektzustand über Methoden),
- die Zuordnung von Objekten zu Klassen,
- die Einordnung von Klassen in einer Klassenhierarchie,
- die Unterstützung einer Abfragesprache.

db4o

db4o (database for objects) ist ein ODBMS, dessen kostenfreie Version 8.0 für Java hier eingesetzt wird. Die hier benötigte jar-Datei *db4o-x.x.x.x-core-java5.jar* befindet sich im Begleitmaterial zu diesem Buch. Sie ist im Verzeichnis *lib* des Projekts P01 dieses Kapitels gespeichert und muss in den Klassenpfad für die Übersetzung und Ausführung in jedem Projekt eingebunden werden.

db4o wird hier im *eingebetteten Modus* (Java-Anwendung und *db4o* laufen in derselben JVM) betrieben.

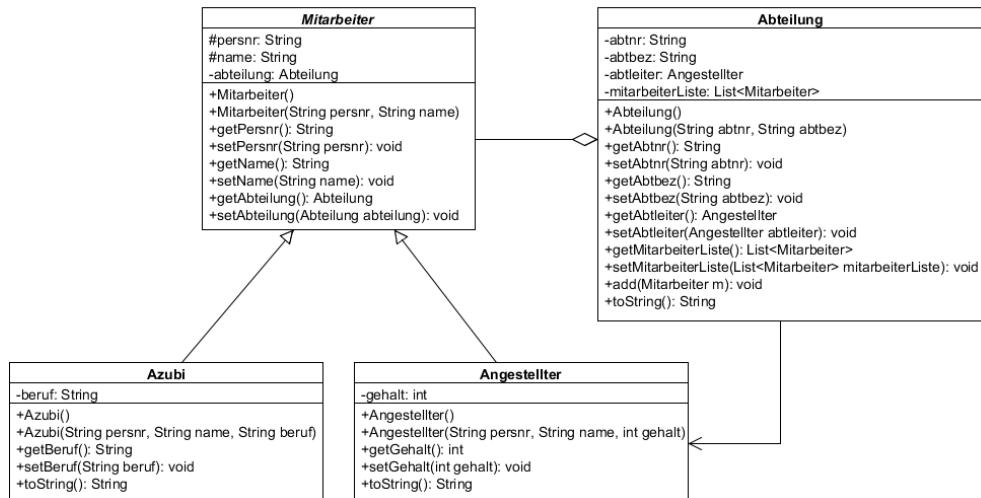


Abbildung 16-2: Klassendiagramm = Datenmodell

Abbildung 16-2 enthält das Klassendiagramm für die folgenden Beispiele. Das Klassendiagramm der Anwendung selbst definiert das Datenbankmodell für die Objektdatenbank.

Programm 16.1

Eine Abteilung hat mehrere Mitarbeiter (Angestellte oder Auszubildende). Genau ein Angestellter leitet diese Abteilung.

Abteilung

```
package model;

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

public class Abteilung {
    private String abtnr;
    private String abtbez;
    private Angestellter abtleiter;
    private List<Mitarbeiter> mitarbeiterListe;

    public Abteilung() {
        mitarbeiterListe = new ArrayList<Mitarbeiter>();
    }

    public Abteilung(String abtnr, String abtbez) {
        this();
        this.abtnr = abtnr;
        this.abtbez = abtbez;
    }

    public String getAbtnr() {
        return abtnr;
    }

    public void setAbtnr(String abtnr) {
        this.abtnr = abtnr;
    }

    public String getAbtbez() {
        return abtbez;
    }

    public void setAbtbez(String abtbez) {
        this.abtbez = abtbez;
    }

    public Angestellter getAbtleiter() {
        return abtleiter;
    }

    public void setAbtleiter(Angestellter abtleiter) {
        this.abtleiter = abtleiter;
    }

    public List<Mitarbeiter> getMitarbeiterListe() {
        return mitarbeiterListe;
    }
}
```

```
public void setMitarbeiterListe(List<Mitarbeiter> mitarbeiterListe) {
    this.mitarbeiterListe = mitarbeiterListe;
}

public void add(Mitarbeiter m) {
    mitarbeiterListe.add(m);
    m.setAbteilung(this);
}

public String toString() {
    return "Abteilung [abtnr=" + abtnr + ", abtbez=" + abtbez + "]";
}
}
```

Mitarbeiter

```
package model;

public abstract class Mitarbeiter {
    protected String persnr;
    protected String name;
    private Abteilung abteilung;

    public Mitarbeiter() {
    }

    public Mitarbeiter(String persnr, String name) {
        this.persnr = persnr;
        this.name = name;
    }

    public String getPersnr() {
        return persnr;
    }

    public void setPersnr(String persnr) {
        this.persnr = persnr;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Abteilung getAbteilung() {
        return abteilung;
    }

    public void setAbteilung(Abteilung abteilung) {
        this.abteilung = abteilung;
    }
}
```

Angestellter

```
package model;

public class Angestellter extends Mitarbeiter {
    private int gehalt;

    public Angestellter() {
    }

    public Angestellter(String persnr, String name, int gehalt) {
        super(persnr, name);
        this.gehalt = gehalt;
    }

    public int getGehalt() {
        return gehalt;
    }

    public void setGehalt(int gehalt) {
        this.gehalt = gehalt;
    }

    public String toString() {
        return "Angestellter [persnr=" + persnr + ", name=" + name
               + ", gehalt=" + gehalt + "]";
    }
}
```

Azubi

```
package model;

public class Azubi extends Mitarbeiter {
    private String beruf;

    public Azubi() {
    }

    public Azubi(String persnr, String name, String beruf) {
        super(persnr, name);
        this.beruf = beruf;
    }

    public String getBeruf() {
        return beruf;
    }

    public void setBeruf(String beruf) {
        this.beruf = beruf;
    }

    public String toString() {
        return "Azubi [persnr=" + persnr + ", name=" + name + ", beruf="
               + beruf + "]";
    }
}
```

16.2 CRUD-Operationen

In diesem Kapitel beschäftigen wir uns mit grundlegenden Datenbankoperationen (CRUD-Operationen):

- Objekte erstmalig speichern (**Create**),
- Objekte lesen (**Read**),
- Objekte aktualisieren (**Update**),
- Objekte löschen (**Delete**).

Create

Das folgende Programm erzeugt Objekte vom Typ `Abteilung`, `Azubi` und `Angestellter`. Azubis und Angestellte werden den Abteilungen zugeordnet.² Dabei entstehen zwei Objektgraphen mit Wurzelobjekten vom Typ `Abteilung`.

Um die im Programm erzeugten Objekte zu speichern, genügen folgende db4o-spezifische Anweisungen:

- Öffnen der Datenbank,
- Speichern der beiden Objektgraphen,
- Schließen der Datenbank.

Eine db4o-Datenbank wird durch das Interface `com.db4o.ObjectContainer` repräsentiert.

Die Klassenmethode `openFile` der Klasse `com.db4o.Db4oEmbedded` stellt die Verbindung zur Datenbank her:

```
static final EmbeddedObjectContainer openFile(String dbName)
```

Die Datenbankdatei mit dem absoluten oder zum aktuellen Verzeichnis relativen Pfadnamen `dbName` wird erstellt, wenn sie noch nicht vorhanden ist.

`EmbeddedObjectContainer` ist ein Subinterface von `ObjectContainer`.

`ObjectContainer`-Methoden:

```
void store(Object obj)
```

speichert das Objekt `obj` sowie alle referenzierten Objekte. Es wird also der gesamte Objektgraph – so wie er im Hauptspeicher existiert – gespeichert. Diese Aussage gilt für die Speicherung neuer, noch nicht in der Datenbank vorhandener Objekte. Für die Aktualisierung von in der Datenbank vorhandenen Objekten beachten Sie bitte die unten stehenden Ausführungen zum Update-Fall.

```
boolean close()
```

schließt die Datenbank.

² Siehe Abbildung 16-2.

```
import model.Abteilung;
import model.Angestellter;
import model.Azubi;

import com.db4o.Db4oEmbedded;
import com.db4o.EmbeddedObjectContainer;

public class Create {
    public static void main(String[] args) {
        EmbeddedObjectContainer db = Db4oEmbedded.openFile("../db/data/db1");

        Abteilung abt1 = new Abteilung("P", "Personal");
        Abteilung abt2 = new Abteilung("T", "Technik");

        Azubi azubi1 = new Azubi("MJ123", "Max Jung", "Fachinformatiker");
        Azubi azubi2 = new Azubi("AN124", "Alfred Neu", "Fachinformatiker");

        abt1.add(azubi1);
        abt2.add(azubi2);

        Angestellter ang1 = new Angestellter("WK111", "Willi Klein", 80000);
        Angestellter ang2 = new Angestellter("HB111", "Hugo Balder", 70000);
        Angestellter ang3 = new Angestellter("HH000", "Hans Hammer", 75000);
        Angestellter ang4 = new Angestellter("OF000", "Otto Frick", 80000);

        abt1.add(ang1);
        abt1.add(ang2);
        abt1.add(ang3);
        abt2.add(ang4);

        abt1.setAbtleiter(ang1);

        db.store(abt1);
        db.store(abt2);

        db.close();
    }
}
```

Read

Um Objekte aus der Datenbank zu Lesen, kann in einfachen Fällen die `ObjectContainer`-Methode `queryByExample` genutzt werden:

```
<T> ObjectSet<T> queryByExample(Object template)
```

`template` kann ein Klassenliteral (z. B. `Abteilung.class`) oder ein so genanntes Musterobjekt sein.

Im ersten Fall werden alle Objekte der entsprechenden Klasse (im Beispiel also alle Objekte vom Typ `Abteilung`) gelesen und zurückgegeben. Im Falle des Musterobjekts werden alle Objekte gelesen, die durch die Vorgabe des Musters bestimmt sind.

Im Musterobjekt sind alle Attribute mit den gewünschten Werten besetzt. Attribute, die den Wert `null` bei Referenzvariablen oder `0` bzw. `false` bei einfachen Datentypen haben, werden für die Abfrage ignoriert. Im folgenden Programm-

beispiel wird also der Angestellte mit der Personalnummer HB111 gelesen. Das Interface `com.db4o.ObjectSet` ist von `java.util.List` abgeleitet.

Das Beispiel zeigt, dass komplette Objektgraphen in den Hauptspeicher geladen werden. Bis zu welcher Ebene referenzierte Objekte im Allgemeinen ausgehend vom Wurzelobjekt geladen werden, wird durch die so genannte *Aktivierungstiefe* bestimmt.³

```

import java.util.List;

import model.Abteilung;
import model.Angestellter;
import model.Mitarbeiter;

import com.db4o.Db4oEmbedded;
import com.db4o.EmbeddedObjectContainer;
import com.db4o.ObjectContainer;

public class Read {
    public static void main(String[] args) {
        EmbeddedObjectContainer db = Db4oEmbedded.openFile("../db/data/db1");

        alleAbteilungen(db);
        einAngestellter(db);

        db.close();
    }

    private static void alleAbteilungen(ObjectContainer db) {
        System.out.println("\n--- Alle Abteilungen ---");
        List<Abteilung> result = db.queryByExample(Abteilung.class);
        for (Abteilung abt : result) {
            System.out.println(abt);
            Angestellter abtleiter = abt.getAbtleiter();
            if (abtleiter != null)
                System.out
                    .println("\tAbteilungsleiter: " + abtleiter.getName());
            for (Mitarbeiter m : abt.getMitarbeiterListe()) {
                System.out.println("\t" + m);
            }
        }
    }

    private static void einAngestellter(ObjectContainer db) {
        System.out.println("\n--- Ein Angestellter ---");
        Angestellter template = new Angestellter("HB111", null, 0);
        List<Angestellter> result = db.queryByExample(template);
        if (result.size() > 0) {
            Angestellter ang = result.get(0);
            System.out.println(ang);
            System.out.println(ang.getAbteilung());
        }
    }
}

```

³ Siehe Kapitel 16.5.

Ausgabe des Programms:

```
--- Alle Abteilungen ---
Abteilung [abtnr=P, abtbez=Personal]
    Abteilungsleiter: Willi Klein
        Azubi [persnr=MJ123, name=Max Jung, beruf=Fachinformatiker]
        Angestellter [persnr=WK111, name=Willi Klein, gehalt=80000]
        Angestellter [persnr=HB111, name=Hugo Balder, gehalt=70000]
        Angestellter [persnr=HH000, name=Hans Hammer, gehalt=75000]
Abteilung [abtnr=T, abtbez=Technik]
    Azubi [persnr=AN124, name=Alfred Neu, beruf=Fachinformatiker]
    Angestellter [persnr=OF000, name=Otto Frick, gehalt=80000]

--- Ein Angestellter ---
Angestellter [persnr=HB111, name=Hugo Balder, gehalt=70000]
Abteilung [abtnr=P, abtbez=Personal]
```

Update

Um ein Objekt in der Datenbank zu ändern, muss dieses gelesen, im Programm geändert und dann wieder mit `store` gespeichert werden.

Im folgenden Programmbeispiel wird das Gehalt des Angestellten HB111 auf 72000 erhöht. Anschließend wird ein neuer Mitarbeiter in der Abteilung T eingestellt.

```
import java.util.List;

import model.Abteilung;
import model.Angestellter;

import com.db4o.Db4oEmbedded;
import com.db4o.EmbeddedObjectContainer;
import com.db4o.ObjectContainer;

public class Update {
    public static void main(String[] args) {
        EmbeddedObjectContainer db = Db4oEmbedded.openFile("../db/data/db1");

        updateGehalt(db);
        neuerAngestellter(db);

        db.close();
    }

    private static void updateGehalt(ObjectContainer db) {
        Angestellter template = new Angestellter("HB111", null, 0);
        List<Angestellter> result = db.queryByExample(template);
        if (result.size() > 0) {
            Angestellter ang = result.get(0);
            ang.setGehalt(72000);
            db.store(ang);
        }
    }
}
```

```

private static void neuerAngestellter(ObjectContainer db) {
    Angestellter ang = new Angestellter("HF999", "Hugo Feldherr", 65000);
    Abteilung template = new Abteilung("T", null);
    List<Abteilung> result = db.queryByExample(template);
    if (result.size() > 0) {
        Abteilung abt = result.get(0);
        abt.add(ang);
        db.store(abt.getMitarbeiterListe());
    }
}
}

```

Durch Ausführung des Programms `Read` kann man sich von dem Erfolg des Updates überzeugen.

Im Programm (Methode `neuerAngestellter`) wird das zu aktualisierende Objekt vom Typ `Abteilung` geladen. Das neue Objekt `ang` vom Typ `Angestellter` wird in die Liste `mitarbeiterListe` mit aufgenommen, zudem wird die Referenz auf die Abteilung im neuen Mitarbeiterobjekt eingetragen (siehe Methode `add` der Klasse `Abteilung`).

store bei Update

Während beim Neuspeichern immer der komplette Objektgraph in der Datenbank gespeichert wird (siehe `Create`), werden bei der Änderung von in der Datenbank bereits vorhandenen Objekten standardmäßig nur die Attribute des der Methode `store` übergebenen Objekts berücksichtigt, nicht aber Änderungen von Attributen weiterer referenzierter Objekte.

Aus diesem Grund enthält das obige Programm die Anweisung

```
db.store(abt.getMitarbeiterListe());
```

und nicht

```
db.store(abt);
```

Delete

Um ein Objekt in der Datenbank zu löschen, muss dieses zuerst gelesen und dann mit der `ObjectContainer`-Methode `delete` gelöscht werden:

```
void delete(Object obj)
```

Es wird ausschließlich `obj` gelöscht. Von `obj` referenzierte Objekte werden nicht gelöscht. Objekte, die vor der Löschoperation das Objekt `obj` referenziert haben, erhalten den Eintrag `null`.

Im folgenden Programmbeispiel wird ein Objekt vom Typ `Angestellter` aus der Datenbank entfernt. Zu beachten ist, dass auch die Referenz auf dieses Objekt aus

der Liste `mitarbeiterListe` im zugeordneten Objekt vom Typ `Abteilung` zu entfernen ist. Diese Liste muss dann mit `store` aktualisiert werden.

```
import java.util.List;

import model.Angestellter;
import model.Mitarbeiter;

import com.db4o.Db4oEmbedded;
import com.db4o.EmbeddedObjectContainer;

public class Delete {
    public static void main(String[] args) {
        EmbeddedObjectContainer db = Db4oEmbedded.openFile("../db/data/db1");

        Angestellter template = new Angestellter("HF999", null, 0);
        List<Angestellter> result = db.queryByExample(template);
        if (result.size() > 0) {
            Angestellter ang = result.get(0);
            List<Mitarbeiter> liste = ang.getAbteilung().getMitarbeiterListe();
            liste.remove(ang);
            db.store(liste);
            db.delete(ang);
        }

        db.close();
    }
}
```

Durch Ausführung des Programms `Read` kann man sich von dem Erfolg der Löschung überzeugen.

16.3 Objektidentität

Jedes Objekt in der Datenbank hat eine systemweit eindeutige Identifikation (*OID, Object Identifier*):

- Zwei Objekte können voneinander verschieden sein, auch wenn alle ihre Attributwerte gleich sind.
- Ein Objekt bleibt dasselbe Objekt, auch wenn sich alle seine Attributwerte ändern.

Dieses Konzept stimmt mit dem Identitätskonzept für Objekte in Programmen überein.

Für das Speichern eines neuen Objekts in der Datenbank und für den Update eines in der Datenbank bereits vorhandenen Objekts wird stets dieselbe Methode `store` verwendet. Wie kann db4o erkennen, ob das als Argument übergebene Objekt in

der Datenbank als neues Objekt gespeichert oder als bereits dort vorhandenes Objekt aktualisiert werden soll?

Session

Innerhalb einer *Session* (Zeitraum zwischen dem Öffnen der Datenbank und dem Schließen der Datenbank) kennt db4o en Zusammenhang zwischen den Objekten in der Datenbank und den Objekten im Programm.

Programm 16.2

Betrachten wir das folgende Programm.

Demo1

```
import model.Abteilung;

import com.db4o.Db4oEmbedded;
import com.db4o.EmbeddedObjectContainer;

public class Demo1 {
    public static void main(String[] args) {
        final String FILE = "../..../db/data/db2";

        Abteilung abt = new Abteilung("M", "Marketing");

        // Session 1
        EmbeddedObjectContainer db = Db4oEmbedded.openFile(FILE);
        db.store(abt);
        db.close();

        // Session 2
        db = Db4oEmbedded.openFile(FILE);
        abt.setAbtbez("Marketing und Vertrieb");
        db.store(abt);
        db.close();
    }
}
```

In der ersten Session wird das Objekt `abt` in der Datenbank als neues Objekt gespeichert. In der zweiten Session wird dasselbe Objekt im Programm geändert und dann mit `store` gespeichert. Wie man sich mit Hilfe des Programms `Read` (für die Datenbank `db2`) überzeugen kann, sind zwei neue Objekte in der Datenbank entstanden:

```
Abteilung [abtnr=M, abtbez=Marketing]
Abteilung [abtnr=M, abtbez=Marketing und Vertrieb]
```

In der ersten Session besteht nach dem Speichern eine *logische Verbindung* zwischen dem Objekt `abt` im Programm und dem in der Datenbank gespeicherten Objekt. Nach Beendigung der Session ist dieses "Wissen" für db4o verloren, sodass db4o in der zweiten Session wieder von einem neuen, "unbekannten" Objekt ausgeht.

Wir führen nun das folgende Programm aus. Als Erstes wird die Datenbank gelöscht.

Demo2

```
import java.io.File;  
  
import model.Abteilung;  
  
import com.db4o.Db4oEmbedded;  
import com.db4o.EmbeddedObjectContainer;  
  
public class Demo2 {  
    public static void main(String[] args) {  
        final String FILE = "../db/data/db2";  
  
        new File(FILE).delete();  
  
        Abteilung abt = new Abteilung("M", "Marketing");  
  
        EmbeddedObjectContainer db = Db4oEmbedded.openFile(FILE);  
        db.store(abt);  
  
        abt.setAbtbez("Marketing und Vertrieb");  
        db.store(abt);  
  
        db.close();  
    }  
}
```

Die Datenbank enthält nur ein Objekt (Read ausführen):

Abteilung [abtnr=M, abtbez=Marketing und Vertrieb]

Aufgrund der nach der ersten Speicherung aufgebauten logischen Verbindung zwischen dem Objekt abt und dem entsprechenden Objekt in der Datenbank wird beim zweiten Speichervorgang das Objekt als bereits in der Datenbank vorhandenes Objekt "wiedererkannt".

Demo3

```
import java.io.File;  
import java.util.List;  
  
import model.Abteilung;  
  
import com.db4o.Db4oEmbedded;  
import com.db4o.EmbeddedObjectContainer;  
  
public class Demo3 {  
    public static void main(String[] args) {  
        final String FILE = "../db/data/db2";  
  
        new File(FILE).delete();  
  
        Abteilung abt = new Abteilung("M", "Marketing");  
  
        EmbeddedObjectContainer db = Db4oEmbedded.openFile(FILE);
```

```

        db.store(abt);
        db.close();

        db = Db4oEmbedded.openFile(FILE);
        Abteilung template = new Abteilung("M", null);
        List<Abteilung> result = db.queryByExample(template);
        if (result.size() > 0) {
            abt = result.get(0);
            abt.setAbtbez("Marketing und Vertrieb");
            db.store(abt);
        }

        db.close();
    }
}

```

Die Ausführung führt zum gleichen Ergebnis.

In der zweiten Session wird das in der ersten Session gespeicherte Objekt abgefragt. Es besteht nun eine logische Beziehung zwischen dem Objekt `abt` im Programm und dem entsprechenden Objekt in der Datenbank. Beim Speichern wird das Objekt "wiedererkannt".

Fazit

In einer Session ist ein Objekt dem System db4o "bekannt", wenn es als neues Objekt in derselben Session gespeichert wurde oder wenn es in derselben Session über eine Abfrage wiedergewonnen wurde.

16.4 Native Abfragen

Die Abfragemöglichkeit aus Kapitel 16.2 (Query By Example, siehe Programm Read) hat verschiedene Einschränkungen:

- Es sind nur UND-Verknüpfungen von Kriterien, keine anderen logischen Verknüpfungen (ODER, NICHT) möglich.
- Da die Belegung von Attributen mit den Default-Werten `null`, `0` bzw. `false` kennzeichnet, dass diese Attribute bei der Suche zu ignorieren sind, kann nach diesen speziellen Werten nicht gesucht werden.

Native Abfragen (Native Queries) sind vollständig in Java formuliert. Um eine solche Abfrage zu erzeugen, muss man zunächst in einer eigenen Klasse, die von der abstrakten Klasse

```
com.db4o.query.Predicate<T>
```

abgeleitet ist, die abstrakte Methode

```
boolean match(T candidate)
```

geeignet implementieren. Liefert die Methode für ein in der Datenbank vorhandenes Objekt vom Typ `T` den Wert `true`, so gehört dieses Objekt zur Ergebnismenge, ansonsten nicht.

Die `ObjectContainer` Methode

```
<T> ObjectSet<T> query(Predicate<T> predicate)
```

führt diese Abfrage aus. `predicate` ist ein Objekt unserer Klasse.

Beispiel:

Gesucht sind alle Angestellten mit einem Gehalt ≥ 78000 .

```
List<Angestellter> result = db.query(
    new Predicate<Angestellter>() {
        public boolean match(Angestellter ang) {
            return ang.getGehalt() >= betrag;
        }
});
```

In diesem Beispiel ist die von `Predicate` abgeleitete Klasse als anonyme Klasse realisiert.

Programm 16.3 liefert alle Angestellten mit einem Gehalt ≥ 78000 , alle Abteilungen mit mindestens 3 Mitarbeitern und eine nach Personalnummern sortierte Liste aller Angestellten.

Programm 16.3

```
import java.util.Comparator;
import java.util.List;

import model.Abteilung;
import model.Angestellter;

import com.db4o.Db4oEmbedded;
import com.db4o.EmbeddedObjectContainer;
import com.db4o.ObjectContainer;
import com.db4o.query.Predicate;

public class Abfragen {
    public static void main(String[] args) {
        EmbeddedObjectContainer db = Db4oEmbedded.openFile("../db/data/db1");

        gehaltGroesser(db, 78000);
        anzahlMitarbeiterGroesser(db, 3);
        sortiere(db);

        db.close();
    }

    private static void gehaltGroesser(ObjectContainer db, final int betrag) {
        System.out.println("\n--- Gehalt größer " + betrag + " ---");
        @SuppressWarnings("serial")
```

```

List<Angestellter> result = db.query(new Predicate<Angestellter>() {
    public boolean match(Angestellter ang) {
        return ang.getGehalt() >= betrag;
    }
});

for (Angestellter ang : result) {
    System.out.println(ang);
}
}

private static void anzahlMitarbeiterGroesser(ObjectContainer db,
    final int anz) {
    System.out.println("\n--- Mitarbeiteranzahl größer " + anz + " ---");

    @SuppressWarnings("serial")
    List<Abteilung> result = db.query(new Predicate<Abteilung>() {
        public boolean match(Abteilung abt) {
            int size = abt.getMitarbeiterListe().size();
            if (size >= anz)
                return true;
            else
                return false;
        }
    });

    for (Abteilung abt : result) {
        System.out.println(abt);
    }
}

private static void sortiere(ObjectContainer db) {
    System.out.println("\n--- Sortierte Liste ---");

    Comparator<Angestellter> comp = new Comparator<Angestellter>() {
        public int compare(Angestellter ang1, Angestellter ang2) {
            return ang1.getPersnr().compareTo(ang2.getPersnr());
        }
    };

    @SuppressWarnings("serial")
    List<Angestellter> result = db.query(new Predicate<Angestellter>() {
        public boolean match(Angestellter ang) {
            return true;
        }
    }, comp);

    for (Angestellter ang : result) {
        System.out.println(ang);
    }
}
}

```

Ausgabe des Programms:

```

--- Gehalt größer 78000 ---
Angestellter [persnr=WK111, name=Willi Klein, gehalt=80000]
Angestellter [persnr=OF000, name=Otto Frick, gehalt=80000]

```

```
--- Mitarbeiteranzahl größer 3 ---
Abteilung [abtnr=P, abtbez=Personal]

--- Sortierte Liste ---
Angestellter [persnr=HB111, name=Hugo Balder, gehalt=72000]
Angestellter [persnr=HH000, name=Hans Hammer, gehalt=75000]
Angestellter [persnr=OF000, name=Otto Frick, gehalt=80000]
Angestellter [persnr=WK111, name=Willi Klein, gehalt=80000]
```

Für die Sortierung wird die folgende query-Variante genutzt:

```
<T> ObjectSet<T> query(Predicate<T> predicate,
                           java.util.Comparator<T> comparator )
```

Die Methode

```
int compare(T o1, T o2)
```

des Interface Comparator bestimmt die Sortierreihenfolge zweier Objekte vom Typ T.

16.5 Tiefe Objektgraphen

Objekte können komplex sein, indem sie auf weitere Objekte verweisen, die ihrerseits wieder auf weitere Objekte verweisen usw.

Beispiel:

Ein Objekt vom Typ `Kunde` referenziert ein Objekt vom Typ `Bestellung`, dieses referenziert eine Liste von Referenzen auf Objekte vom Typ `Bestellposition`. Ein Objekt vom Typ `Bestellposition` referenziert ein Objekt vom Typ `Artikel`, dieses referenziert ein Objekt vom Typ `Lager`.

Der Objektgraph mit dem Wurzelobjekt vom Typ `Kunde` besteht also aus *sechs* Hierarchieebenen (siehe Abbildung 16-3).

Nehmen wir an, dass diese Objekte alle in einer Datenbank gespeichert sind. Fragen wir das Wurzelobjekt vom Typ `Kunde` ab (z. B. mit Query By Example), navigieren wir dann im Programm entlang dem Referenzpfad vom `Kunde`-Objekt bis zum `Lager`-Objekt (vorausgesetzt in den beteiligten Klassen sind entsprechende get-Methoden zum Lesen der Referenzen und sonstigen Attribute vorhanden), so stellen wir fest, dass die Werte des `Lager`-Objekts nicht geladen wurden (wir erhalten die Default-Werte wie `null` bzw. `0`).



Abbildung 16-3: 6-stufige Hierarchie

Aktivierungstiefe

Das liegt daran, dass die *Aktivierungstiefe* eines Objekts standardmäßig auf 5 (fünf Ebenen) festgelegt ist. Diese Begrenzung besteht, um evtl. unnötige Leseoperationen einzusparen und damit die Performance der Datenbank zu erhöhen.

Die Aktivierungstiefe kann vor dem Öffnen der Datenbank nach Bedarf abweichend vom Standardwert gesetzt werden. Dazu muss ein Objekt vom Typ `EmbeddedConfiguration` erzeugt werden.

Die `Db4oEmbedded`-Methode

```
static EmbeddedConfiguration newConfiguration()
```

erzeugt ein Objekt vom Typ

```
com.db4o.config.EmbeddedConfiguration
```

Für ein solches Objekt kann nun die Aktivierungstiefe `depth` wie folgt gesetzt werden:

```
config.common().activationDepth(depth);
```

Die `Db4oEmbedded`-Methode

```
static final EmbeddedObjectContainer openFile(
    EmbeddedConfiguration config, String dbName)
```

öffnet die Datenbank mit Berücksichtigung der Konfiguration.

Beispiel:

```
EmbeddedConfiguration config = Db4oEmbedded.
    newConfiguration();
config.common().activationDepth(6);
EmbeddedObjectContainer db = Db4oEmbedded.openFile(config,
    "../..../db/data/db3");
```

Nun werden auch die Attributwerte des `Lager`-Objekts geladen.

Bei dieser Vorgehensweise wird die Aktivierungstiefe global gesetzt. Sie gilt dann für alle Objekte der Datenbank.

Ohne Konfigurationseinstellung kann die Aktivierungstiefe *dynamisch* für ein einzelnes Objekt gesetzt werden. Dazu wird die folgende `ObjectContainer`-Methode genutzt:

```
void activate(Object obj, int depth)
```

Beispiel: `db.activate(kunde, 6);`

Die Methode

```
void deactivate(Object obj, int depth)
```

setzt die Werte der Ebene `depth` auf null bzw. 0.

Updatetiefe

Bei Änderung eines Objekts werden standardmäßig nur die Attributwerte des Objekts selbst geändert, nicht aber Werte von in diesem Objekt referenzierten Objekten. Dies kann mit Setzen der geeigneten *Updatetiefe* geändert werden.

Die Updatetiefe `depth` kann global wie folgt gesetzt werden:

```
config.common().updateDepth(depth);
```

Diese kann auch gezielt für Objekte einer bestimmten Klasse gesetzt werden, z. B.:

```
config.common().objectClass(Kunde.class).updateDepth(depth);
```

Programm 16.4

Mit den Programmbeispielen des Begleitmaterials zu diesem Buch kann das Verhalten bei unterschiedlichen Einstellungen der Tiefe getestet werden. In der hier benutzten db4o-Version 8.0 ist die Default-Aktivierungstiefe 5 und die Default-Updatetiefe 1.

16.6 Callbacks

Callback-Methoden werden in den Objekten des Datenmodells implementiert. Sie werden von db4o automatisch aufgerufen, wenn bestimmte Ereignisse eingetreten sind:

- Aktivierung,
- Deaktivierung,
- Neuaufnahme,
- Änderung und

- Löschung.

Dabei kann zwischen dem Zeitpunkt vor Ausführung und nach Ausführung der jeweiligen Aktion unterschieden werden.

Callback-Methoden sind:

```
boolean objectCanXxx(ObjectContainer db)
void objectOnXxx(ObjectContainer db)
```

Hier ist Xxx zu ersetzen durch Activate, Deactivate, New, Update, Delete.

Wird bei der ersten Gruppe von Methoden false zurückgegeben, so wird die Aktion nicht durchgeführt.

Um Callback-Methoden nutzen zu können, muss kein spezielles Interface implementiert werden, es reicht die entsprechenden Methoden mit den vorgegebenen Methodenköpfen zu realisieren.

Callback-Methoden können genutzt werden, um die Integrität der Daten zu gewährleisten.

Im Programmbeispiel wird die Gültigkeit von Werten (Artikelpreis) bei der Speicherung geprüft. Das Löschen eines Objekts (Lager) wird unterbunden, wenn noch andere Objekte (Artikel) vorhanden sind, die das zu löschen Objekt referenzieren.

Mit transient gekennzeichnete Attribute eines Objekts (Preistufe) werden nicht in der Datenbank gespeichert. Um solche Werte beim Laden aus der Datenbank auf Basis des Zustands des Objekts zu rekonstruieren, kann die Methode objectOnActivate genutzt werden.

Der Callback-Mechanismus kann mit

```
config.common().callbacks(true);
```

bzw.

```
config.common().callbacks(false);
```

ein- und ausgeschaltet werden.⁴ Die Default-Einstellung ist true.

Im Programm wird das *Logging-API* verwendet, um Fehlermeldungen etc. auf dem Bildschirm bzw. in einer Datei zu protokollieren. Die Datei *logging.properties* enthält Konfigurationseinstellungen.

⁴ Vgl. Kapitel 16.5.

Programm 16.5

```
package model;

import java.util.logging.Logger;

import com.db4o.ObjectContainer;

public class Artikel {
    private final static Logger LOGGER = Logger.getLogger(Artikel.class
        .getName());

    private String id;
    private double preis;
    private transient String preisstufe;
    private Lager lager;

    public Artikel(String id, double preis) {
        this.id = id;
        this.preis = preis;
        berechnePreisstufe();
    }

    public String getId() {
        return id;
    }

    public double getPreis() {
        return preis;
    }

    public void setPreis(double preis) {
        this.preis = preis;
        berechnePreisstufe();
    }

    public Lager getLager() {
        return lager;
    }

    public void setLager(Lager lager) {
        this.lager = lager;
    }

    public String getPreisstufe() {
        return preisstufe;
    }

    public boolean objectCanNew(ObjectContainer db) {
        Artikel template = new Artikel(id, 0);
        if (db.queryByExample(template).size() > 0) {
            LOGGER.severe("Artikel " + id + " ist bereits vorhanden.");
            return false;
        } else
            return checkPreis();
    }

    public void objectOnNew(ObjectContainer db) {
        LOGGER.info("Artikel " + id + " wurde neu angelegt.");
    }
}
```

```
public boolean objectCanUpdate(ObjectContainer db) {
    return checkPreis();
}

public void objectOnUpdate(ObjectContainer db) {
    LOGGER.info("Artikel " + id + " wurde geändert.");
}

public void objectonDelete(ObjectContainer db) {
    LOGGER.info("Artikel " + id + " wurde gelöscht.");
}

public void objectOnActivate(ObjectContainer db) {
    berechnePreisstufe();
}

private void berechnePreisstufe() {
    if (preis > 0 && preis <= 100) {
        preisstufe = "A";
    } else if (preis > 100 && preis <= 500) {
        preisstufe = "B";
    } else if (preis > 500) {
        preisstufe = "C";
    } else {
        preisstufe = "";
    }
}

private boolean checkPreis() {
    if (!(preis > 0 && preis <= 1000.)) {
        LOGGER.severe("Der Preis " + preis + " für Artikel " + id
            + " ist ungültig.");
        return false;
    } else
        return true;
}
}

package model;

import java.util.List;
import java.util.logging.Logger;

import com.db4o.ObjectContainer;
import com.db4o.query.Predicate;

public class Lager {
    private final static Logger LOGGER = Logger
        .getLogger(Lager.class.getName());

    private String id;

    public Lager(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }
}
```

```
public void setId(String id) {
    this.id = id;
}

public boolean objectCanDelete(ObjectContainer db) {
    @SuppressWarnings("serial")
    List<Artikel> result = db.query(new Predicate<Artikel>() {
        public boolean match(Artikel artikel) {
            if (artikel.getLager().getId().equals(id))
                return true;
            else
                return false;
        }
    });

    if (result.size() > 0) {
        LOGGER.severe("Lager " + id + " kann nicht gelöscht werden.");
        return false;
    } else
        return true;
}

public void objectonDelete(ObjectContainer db) {
    LOGGER.info("Lager " + id + " wurde gelöscht.");
}
}

import java.util.List;

import model.Artikel;
import model.Lager;

import com.db4o.Db4oEmbedded;
import com.db4o.EmbeddedObjectContainer;

public class Test {
    private static final String FILE = "../../db/data/db4";

    public static void main(String[] args) {
        System.setProperty("java.util.logging.config.file",
                           "logging.properties");

        session1();
        session2();
        session3();
        session4();
    }

    // Neue Artikel anlegen
    private static void session1() {
        System.out.println("\n--- Session 1 ---");
        EmbeddedObjectContainer db = Db4oEmbedded.openFile(FILE);

        Artikel artikel1 = new Artikel("4711", 10.);
        Artikel artikel2 = new Artikel("4712", 400.);
        Artikel artikel3 = new Artikel("4713", 2000.);
        Artikel artikel4 = new Artikel("4711", 20.);

        Lager lager = new Lager("01");
        artikel1.setLager(lager);
    }
}
```

```
artikel2.setLager(lager);
artikel3.setLager(lager);
artikel4.setLager(lager);

db.store(artikel1);
db.store(artikel2);
db.store(artikel3);
db.store(artikel4);

db.close();
}

// Artikelpreis ändern
private static void session2() {
    System.out.println("\n--- Session 2 ---");
    EmbeddedObjectContainer db = Db4oEmbedded.openFile(FILE);

    Artikel template = new Artikel("4711", 0);
    List<Artikel> result = db.queryByExample(template);
    if (result.size() > 0) {
        Artikel artikel = result.get(0);
        System.out.println("Alte Preisstufe: " + artikel.getPreisstufe());
        artikel.setPreis(110.);
        System.out.println("Neue Preisstufe: " + artikel.getPreisstufe());
        db.store(artikel);
    }

    db.close();
}

// Artikel löschen
private static void session3() {
    System.out.println("\n--- Session 3 ---");
    EmbeddedObjectContainer db = Db4oEmbedded.openFile(FILE);

    Artikel template = new Artikel("4712", 0);
    List<Artikel> result = db.queryByExample(template);
    if (result.size() > 0) {
        Artikel artikel = result.get(0);
        db.delete(artikel);
    }

    db.close();
}

// Lager löschen
private static void session4() {
    System.out.println("\n--- Session 4 ---");
    EmbeddedObjectContainer db = Db4oEmbedded.openFile(FILE);

    Lager template = new Lager("01");
    List<Lager> result = db.queryByExample(template);
    if (result.size() > 0) {
        Lager lager = result.get(0);
        db.delete(lager);
    }

    db.close();
}
}
```

logging.properties:

```
# Der ConsoleHandler gibt die Nachrichten auf std.err aus.  
# Weitere Handler können hinzugenommen werden. Hier z. B. der Filehandler.  
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler  
  
# Die Nachrichten in eine Datei schreiben:  
java.util.logging.FileHandler.pattern = ./db4o.log  
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter  
java.util.logging.FileHandler.append = false
```

Konsolen-Ausgabe des Programms:

```
--- Session 1 ---  
Jul 06, 2014 11:23:13 AM model.Artikel objectOnNew  
INFO: Artikel 4711 wurde neu angelegt.  
Jul 06, 2014 11:23:13 AM model.Artikel objectOnNew  
INFO: Artikel 4712 wurde neu angelegt.  
Jul 06, 2014 11:23:13 AM model.Artikel checkPreis  
SEVERE: Der Preis 2000.0 für Artikel 4713 ist ungültig.  
Jul 06, 2014 11:23:13 AM model.Artikel objectCanNew  
SEVERE: Artikel 4711 ist bereits vorhanden.  
  
--- Session 2 ---  
Alte Preisstufe: A  
Neue Preisstufe: B  
Jul 06, 2014 11:23:13 AM model.Artikel objectOnUpdate  
INFO: Artikel 4711 wurde geändert.  
  
--- Session 3 ---  
Jul 06, 2014 11:23:13 AM model.Artikel objectonDelete  
INFO: Artikel 4712 wurde gelöscht.  
  
--- Session 4 ---  
Jul 06, 2014 11:23:13 AM model.Lager objectCanDelete  
SEVERE: Lager 01 kann nicht gelöscht werden.
```

16.7 Aufgaben

Die folgenden Aufgaben bauen aufeinander auf. Überprüfen Sie Änderungen der Datenbank jeweils mit einem Leseprogramm (Query By Example oder native Abfrage).

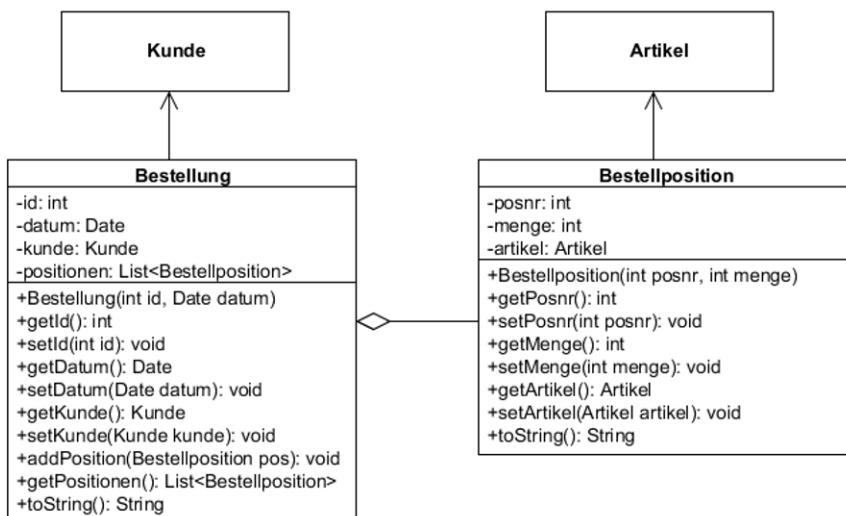
1. Erstellen Sie die Klasse Artikel und fügen Sie fünf Artikel in die Datenbank ein.

Artikel	
-id: int	
-preis: double	
-bestand: int	
+Artikel(int id, double preis, int bestand)	
+getId(): int	
+setId(int id): void	
+getPreis(): double	
+setPreis(double preis): void	
+getBestand(): int	
+setBestand(int bestand): void	
+addBestand(int menge): void	
+toString(): String	

2. Erhöhen Sie die Lagerbestände für einige Artikel.
3. Erstellen Sie die Klasse Kunde und fügen Sie drei Kunden in die Datenbank ein.

Kunde	
-id: int	
-name: String	
+Kunde(int id, String name)	
+getId(): int	
+setId(int id): void	
+getName(): String	
+setName(String name): void	
+toString(): String	

4. Ergänzen Sie das Modell um neue Klassen und Beziehungen. Speichern Sie dann eine Bestellung für zwei Artikel. Vergessen Sie nicht den Kundenbezug.



5. Ermitteln Sie den Gesamtwert einer Bestellung.
6. Reduzieren Sie für eine Bestellung den Lagerbestand der bestellten Artikel um die jeweilige Bestellmenge.
7. Ermitteln Sie die Bestellungen, die mindestens zwei Positionen umfassen. Benutzen Sie eine native Abfrage.
8. Fügen Sie in die Klasse `Artikel` das neue Attribut `mindestbestand` vom Typ `int` ein. Bei Änderung des Bestandes soll im Falle, dass der Bestand negativ ist, die Zahl `0` gespeichert werden. Bei Unterschreitung des Mindestbestandes soll eine Meldung ausgegeben werden. Implementieren Sie hierfür geeignete Callback-Methoden.

Quellen im Internet

Beispielprogramme und Lösungen zu den Aufgaben

Den Zugang zum Begleitmaterial finden Sie auf der Website des Verlags

www.springer-vieweg.de

bei den bibliographischen Angaben zu diesem Buch.

Weiteres hierzu enthält Kapitel 1.6.

Java Standard Edition

<http://www.oracle.com/technetwork/java/javase/>

Hier finden Sie die neueste Version zur *Java Standard Edition* (Java SE) für diverse Plattformen sowie die zugehörige Dokumentation. Beachten Sie die für Ihre Plattform zutreffende Installationsanweisung.

Java-Entwicklungsumgebungen und -Editoren

Eclipse <http://www.eclipse.org>

NetBeans <http://www.netbeans.org>

IntelliJ IDEA <http://www.jetbrains.com/idea/>

JDeveloper <http://www.oracle.com/technetwork/developer-tools/jdev/>

JCreator <http://www.jcreator.com>

Java-Editor <http://www.javaeditor.org>

Datenbanksysteme und Tools

H2 <http://www.h2database.com>

Derby <http://db.apache.org/derby/>

MySQL (Server und Tools) <http://dev.mysql.com>

HeidiSQL (MySQL GUI Tool) <http://www.heidisql.com>

JDBC-Treiber für Microsoft Access

UCanAccess <http://ucanaccess.sourceforge.net/site.html>

Java Persistence Provider

EclipseLink

<http://www.eclipse.org/eclipselink/>**Java GUI Designer**

WindowBuilder

<http://www.eclipse.org/windowbuilder/>

Literaturhinweise

Im Rahmen dieses Buches können einige Themen im Umfeld von Java – wie z. B. objektorientierter Entwurf, Datenbanken, SQL, Kommunikationsprotokolle – nicht ausführlich behandelt werden. Die folgenden Quellen sind für eine Einführung bzw. Vertiefung gut geeignet.

Objektorientierte Softwareentwicklung

Poetzsch-Heffter, A.: Konzepte objektorientierter Programmierung. Springer, 2. Auflage 2009

Seidl, M.; Brandsteidl, M.; Huemer, C.; Kappel, G.: UML@Classroom: Eine Einführung in die objektorientierte Modellierung. dpunkt.verlag 2012

Datenbanken, SQL

Cordts, S.; Blakowski, G.; Brosius, G.: Datenbanken für Wirtschaftsinformatiker: Nach dem aktuellen Standard SQL:2008. Vieweg+Teubner 2011

Emrich, M.: Datenbanken & SQL für Einsteiger: Datenbankdesign und MySQL in der Praxis. CreateSpace Independent Publishing Platform 2013

Schicker, E.: Datenbanken und SQL: Eine praxisorientierte Einführung mit Anwendungen in Oracle, SQL Server und MySQL. Springer Vieweg 2014

Unterstein, M.; Matthiessen, G.: Relationale Datenbanken und SQL in Theorie und Praxis. Springer, 5. Auflage 2012

JDBC

Abts, D.: Masterkurs Client/Server-Programmierung mit Java. Vieweg+Teubner, 3. Auflage 2010

JDBC Database Access Tutorial: <http://docs.oracle.com/javase/tutorial/jdbc/>

Verteilte Systeme, Client/Server-Computing

Abts, D.: Masterkurs Client/Server-Programmierung mit Java. Vieweg+Teubner, 3. Auflage 2010

Bengel, G.: Grundkurs Verteilte Systeme. Springer Vieweg, 4. Auflage 2014

Oechsle, R.: Parallele und verteilte Anwendungen in Java. Hanser, 3. Auflage 2011

Schill, A.; Springer, T.: Verteilte Systeme. Springer, 2. Auflage 2012

Vogt, C.: Nebenläufige Programmierung. Ein Arbeitsbuch mit UNIX/Linux und Java. Hanser 2012

Internet, TCP/IP

Comer, D. E.: TCP/IP. mitp 2011

Schreiner, R.: Computernetzwerke. Von den Grundlagen zur Funktion und Anwendung. Hanser, 4. Auflage 2012

HTTP/1.0: <http://www.ietf.org/rfc/rfc1945.txt>

HTTP/1.1: <http://www.ietf.org/rfc/rfc2616.txt>

HTML

Kobert, T.: HTML 5. bhv 2013

Kröner, P.: HTML 5. Open Source Press, 2. Auflage 2011

SELFHTML: <http://de.selfhtml.org/>

Java Persistence API

Müller, B.; Wehr, H.: Java Persistence API 2: Hibernate, EclipseLink, OpenJPA und Erweiterungen. Hanser 2012

<http://docs.oracle.com/javaee/7/tutorial/doc/persistence-intro.htm>

Java 8

Inden, M.: Java 8 – Die Neuerungen. Lambdas, Streams, Date And Time API und JavaFX im Überblick. dpunkt.verlag 2014

Sachwortverzeichnis

- @
 - @AttributeOverride 496
 - @AttributeOverrides 496
 - @Column 464
 - @DiscriminatorValue 502
 - @Embeddable 495
 - @Embedded 496
 - @EmbeddedId 495
 - @Entity 463
 - @EntityListeners 513
 - @FunctionalInterface 190
 - @GeneratedValue 464
 - @Id 464
 - @Inheritance 501
 - @JoinColumn 490
 - @JoinTable 490
 - @ManyToMany 488
 - @ManyToOne 480
 - @MappedSuperclass 513
 - @OneToMany 482
 - @OneToOne 476
 - @PostLoad 508
 - @PostPersist 508
 - @PostRemove 508
 - @PostUpdate 508
 - @PrePersist 508
 - @PreRemove 508
 - @PreUpdate 508
 - @Table 464
 - @Temporal 483
 - @Transient 464
- A
 - abgeleitete Klasse 49
 - abstract 55
 - Abstract Window Toolkit 279
 - AbstractButton 308
 - AbstractTableModel 355
- abstrakte Klasse 55
- abstrakte Methode 55
- ActionEvent 308, 311, 323, 329, 337
- ActionListener 309, 323, 337
- Adapterklasse 292
- Aktivierungstiefe 534, 544
- Animation 387
- Annotation 462
- anonyme Klasse 73
- Anweisung 25
- Anwendungsschicht 434, 440
- API-Evolution 66
- Applet 373
- AppletContext 379
- appletviewer 373, 375, 379
- Applikation 48, 78
- arithmetischer Operator 17
- ARM 216
- Array 74
- ArrayList 183
- Arrays 147
- Attribut 36
- AudioClip 385
- Aufzählung 80
- Aufzählungstyp 82
- Ausdruck 17
- ausführbare Klasse 48
- Ausgabestrom 209
- Ausnahme 97
- Ausnahmebehandlung 97
- Ausnahmen-Verkettung 106
- Ausnahmetyp 98
- Auswahlkomponente 328
- Autoboxing 124
- AutoCloseable 216
- Automatic Resource Management 216
- Auto-Unboxing 124

AWT 279

B

Basisklasse 49

Bedingungsoperator 23

Bestätigungsdialog 349

Bezeichner 12

Beziehungstyp 473

bidirektional 474

BigInteger 153

BinaryOperator 192

Bitoperator 21

Bivarianz 176

Block 25

BlockingQueue 269

boolean 13

Boolean 120

Border 302, 303

BorderLayout 298

Box 301

BoxLayout 301

break 30

BufferedInputStream 211, 217

BufferedOutputStream 211, 217

BufferedReader 212, 224

BufferedWriter 212, 224

Button 307

ButtonGroup 311

byte 14

Byte 120

ByteArrayInputStream 210

ByteArrayOutputStream 210

Bytecode 3

Bytestrom 210

C

Calendar 157

call by reference 41

call by value 41

CamelCase 13

CascadeType 478, 485

Cast-Operator 24

catch 102

catch or throw 100

ChangeEvent 320, 334

ChangeListener 320

char 14

Character 120

Character Entity 427

CharArrayReader 212

CharArrayWriter 212

Class 142

CLASSPATH 85

Client 413

Client/Server-Anwendung 413

clone 129

Cloneable 129

Closure 195

Collator 441

Collection 181

Collection Framework 132, 133, 181

Color 286

ComboBoxEditor 329

Comparable 147, 172

Comparator 543

Component 280, 282

Connection 397, 404

Consumer 192

Container 132, 280, 316

Content Pane 285, 374

continue 30

Contravarianz 178

Controller 281

Covarianz 177

CRUD 532

CSV-Format 433

Cursor 283

D

Daemon-Thread 250

DataInputStream 211, 218, 219

DataOutputStream 211, 218

Date 155

DateFormat 162

Dateizeiger 234

Datenbank 395

- Datenbankmanagementsystem 395
Datenbanksystem 395
Datenbank-URL 397
Datenkapselung 58
Datenkomprimierung 238
Datenstrom 209
Datentyp 13
db4o 527, 528
Db4oEmbedded 532
DBMS 395
Deadlock 259
DecimalFormat 151
default 63
Default-Methode 63
Default-Paket 86
Dekrementierungsoperator 18
delete 406
Deployment 431
Derby 397
Deserialisierung 229
Desktop 456
Dialogfenster 346
Diamond Operator 171
Dimension 282
Diskriminator 501
do ... while 27
double 15
Double 120
Double-Buffering 387
Downcast 54, 60
Drei-Schichten-Architektur 433
DriverManager 397
dynamisches Binden 56
- E**
EclipseLink 462
einfacher Datentyp 13
Einfachvererbung 52
Eingabedialog 350
Eingabestrom 209
EmbeddedConfiguration 544
EmbeddedObjectContainer 532
EmptyBorder 302
- Entity Manager 465
Entity-Klasse 462
EntityManager 469
EntityManagerFactory 469
enum 82
Enumeration 135
EOFException 219
equals 126
Ereignis 290
Ereignisbehandlung 289
Ereignis-Dispatcher 361
Ereignisempfänger 290
Ereignismodell 289
Ereignisquelle 290
Ereignisschlange 361
err 140
Error 98
Escape-Sequenz 14
Event 290
Event Queue 361
Event-Modell 289
EventQueue 361, 363
Exception 97, 98, 99
exception chaining 106
Exception Handling 97
extends 49, 59
- F**
false 13
Fenster 281
Fetch Join 493
file 413
File 205
FileFilter 208, 352
FileInputStream 210, 215
FileNotFoundException 215
FileOutputStream 210, 215
Filepointer 234
FileReader 213, 223
FileWriter 213, 223
FilterInputStream 211, 227
FilterOutputStream 211, 227
FilterReader 213, 227

FilterWriter 213, 227
final 42, 58
finally 104
flache Kopie 130
Fließkommatyp 15
Fließkommazahl 15
float 15
Float 120
FlowLayout 297
Fokus 326
Font 286
for 28
foreach 30, 76, 134, 183
forEach 200
Fortschrittsbalken 365
FTP 412
Function 192
Funktionsinterface 190

G

ganze Zahl 14
ganzzahliger Typ 14
Garbage Collector 46
Generics 169, 181
generische Klasse 170
generische Methode 179
generischer Typ 170
generisches Interface 170
Generizität 169
GET 419
Gleitkommazahl 15
Grafikkontext 285
Graphical User Interface 279
Graphics 285, 384
GregorianCalendar 156
GridBagConstraints 304
GridLayout 304
GridLayout 299
GUI 279
gzip 238
GZIPOutputStream 238
GZIPInputStream 238

H

H2 396
H2 Database 396
hashCode 127
HashMap 185
Hashtable 135, 186
Heap 46
Hintergrundfarbe 288
Hotkey 336
HTML 314, 373, 375, 418
HTML-Formular 418
HTTP 412, 417
HTTP-Body 422
HTTP-Header 422
HTTP-Request 418
HTTP-Response 418, 422
Hüllklasse 120
Hybridanwendung 381
HyperText Transfer Protocol 417

I

I18N 159
Icon 307
IDE 3
Identität 38
if ... else 25
Image 383, 384
ImageIcon 307
ImageObserver 384
Impedance Mismatch 459
imperativ 11
implements 59
import 85
import static 88
in 140
Initialisierung 16, 38, 46, 48, 75
Initialisierungsblock 46, 48
Inkrementierungsoperator 18
innere Klasse 68
InputEvent 336, 341
InputStream 210, 211
InputStreamReader 213, 222
InputVerifier 323

- insert 405
- instanceof 54, 60
- Instanz 35, 38
- Instanziierung 38
- Instanzklasse 70
- Instanzmethode 48
- Instanzvariable 47
- int 14
- Integer 120
- Integrated Development Environment 3
- Interface 59
- intermediate operation 200
- Internationalisierung 159
- InterruptedException 252, 265
- Invarianz 174
- invokeAndWait 364
- invokeLater 364
- IOException 211, 214, 219, 234
- ItemEvent 311, 329
- ItemListener 311
- Iterable 182, 183
- Iterator 182
- J**
- JApplet 374
- jar 310, 383
- Java Card 2
- Java Database Connectivity 395
- Java EE 2
- Java Foundation Classes 279
- Java ME 2
- Java Persistence API 459
- Java Persistence Query Language 464, 492
- Java Runtime Environment 2
- Java SE 1
- Java SE Development Kit 2
- javadoc 12
- Java-Systemeigenschaft 141
- JButton 307
- JCheckBox 311
- JCheckBoxMenuItem 337
- JComboBox 328
- JComponent 284
- JDBC 395
- JDBC-Treiber 395
- JDialog 346
- JDK 2
- JFC 279
- JFileChooser 351
- JFrame 281
- JLabel 314
- JList 330
- JMenu 335
- JMenuBar 335
- JMenuItem 336
- JOINED 500, 506
- JOptionPane 349
- JPA 459
- JPanel 284
- JPasswordField 323
- JPopupMenu 344
- JPQL 464, 492
- JProgressBar 365
- JRadioButton 311
- JRadioButtonMenuItem 337
- JRE 2
- JScrollPane 316
- JSlider 333
- JSplitPane 317
- JTabbedPane 318
- JTable 355
- JTextArea 325
- JTextComponent 321
- JTextField 322
- JToolBar 337
- JVM 3
- K**
- KeyAdapter 292, 295
- KeyEvent 295, 336
- KeyListener 290, 295
- KeyStroke 336
- Klasse 35, 36
- Klassendiagramm 435

- Klassenliteral 143
Klassenmethode 48
Klassenvariable 47
Klon 129
Kommandozeilen-Parameter 78
Kommentar 11
kompatibel 172
Komprimierung 238
Konstante 42, 58
Konstruktor 36, 44, 51
Kontextmenü 341, 344
kontrollierte Ausnahme 98, 99, 100
Kontrollstruktur 25
Koordinatensystem 283
Kopier-Konstruktor 45
- L**
- Label 314
Lambda-Ausdruck 189, 192
Ländercode 159
LayoutManager 295
Layout-Manager 295
Lebenszyklus (JPA) 467
LineNumberReader 212
Linksschieben 22
List 133, 183
Listener 290
ListSelectionEvent 360
ListSelectionListener 360
ListSelectionModel 331, 360
Literal 13
Locale 160
localhost 416
Logging-API 546
logischer Operator 20
logischer Typ 13
lokale Klasse 71
lokale Variable 42
long 14
Long 120
Lower-Typebound 178
- M**
- main 48, 78
Manifest-Datei 310
Many-To-Many 473
Many-To-One 473
Map 184
Map.Entry 185
markierte Anweisung 30
Math 149
mathematische Funktion 149
Mausaktion 340
Mauszeiger 283
MediaTracker 384
mehrdimensionales Array 77
Menü 335
META-INF 465
Methode 36, 40
Methodenkopf 40
Methodenreferenz 196
Methodenrumpf 40
Microsoft Access 396
Mitteilungsdialog 350
modales Fenster 346
Model 281
Model-View-Controller-Architektur
 281
Modifizierer 57
MouseAdapter 341
MouseEvent 341
MouseInputAdapter 342
MouseInputListener 342
MouseListener 340, 341
MouseMotionAdapter 342
MouseMotionListener 341, 342
Multicatch 103
Multitasking 247
Multithreading 247
MVC-Architektur 281
MySQL 397
- N**
- Namenskonvention 13
native 58

- Native Abfrage 540
Native Query 540
new 38, 74
nicht kontrollierte Ausnahme 98
notify 264, 265, 269
notifyAll 264, 269
null 38
Null-Layout 295
- O**
- O/R-Mapping 460
Object 52, 126
Object Identifier 537
Object Manager Enterprise 528
Object Relational Mapping 460
ObjectContainer 532
ObjectInputStream 211, 229
ObjectOutputStream 211, 229
ObjectSet 534
Object-Tag 376
Objekt 35, 38
Objektdatenbank 527
Objektgraph 528, 543
Objektidentität 528, 537
objektorientiertes
 Datenbankmanagementsystem
 527
 ODBMS 527
 OID 537
 OME 528
 One-To-Many 473
 One-To-One 473
 Operand 17
 Operator 17
 optimistic locking 518
 OptimisticLockException 521
 optimistisches Sperren 518
 out 140
 OutputStream 210, 212
 OutputStreamWriter 213, 222
 Overloading 43
 Overriding 49
- P**
- package 84
paintComponent 285
Paket 57, 84
Panel 284
Parameter 40
Parameter-Tag 377
parametrisierter Typ 170
Persistence Context 466
Persistence Provider 462
Persistence Unit 465
persistence.xml 465
Persistenz 229
Persistenzeinheit 465
Persistenzkontext 466
Persistenzschicht 434, 435, 436
Pipe 269
PipedInputStream 211, 269, 270
PipedOutputStream 211, 269, 270
PipedReader 213
PipedWriter 213
Plug-In 373
Point 282
POJO 459
Policy-Datei 391
Polymorphie 56
Port 414
Portnummer 411, 414, 416
POST 419
Präsentationsschicht 434, 449
Predicate 191, 540
Primärschlüssel 395
primitiver Datentyp 13
PrintStream 211, 226
PrintWriter 213, 225
private 58
Producer/Consumer 264
Properties 138
Property-Liste 138
protected 58
Prozess 247
public 37, 57

- Punktnotation 39
PushbackInputStream 211, 220
PushbackReader 213
- Q**
Quelldatei 37
Query By Example 533, 540
Query String 421
- R**
Random 152
Random-Access-Datei 234
RandomAccessFile 234
Raw Type 175
Reader 212, 214
Rechtsschieben 22
Referenztyp 38, 60
Referenzvariable 38
relationaler Operator 19
relationales Datenbanksystem 395
repaint 286, 387
ResourceBundle 164
Ressourcenbündel 164
Rest-Operator 18
ResultSet 405, 408
ResultSetMetaData 408
return 40
RGB-Farbmodell 286
Rückgabetyp 40
Rückgabewert 40
Runnable 248
Runtime 275
RuntimeException 98
- S**
Schiebeoperator 22
Schleife 27
Schlüsseltabelle 184
Schlüsselwort 12
select 405
SequenceInputStream 211
Serialisierung 229
Serializable 230
serialVersionUID 230
- Server 413
ServerSocket 415
Session 538
Set 185
short 14
Short 120
short circuit 20
Shutdown 275
Signatur 43
SimpleDateFormat 155
SINGLE_TABLE 500, 501
Socket 413, 414
Sprachcode 159
Sprunganweisung 30
SQL 395, 404
SQL-Abfrage 405
SQLException 397
Stack 46
Standarddatenstrom 140, 209
Standardkonstruktor 44
Statement 405, 407
static 47, 48, 68, 69
statische import-Klausel 88
statische Initialisierung 48
statische Klasse 69
statische Methode 48
statische Variable 47
Stream 209
Stream-API 200
String 16, 111
StringBuffer 116
StringBuilder 117
StringReader 213
 StringTokenizer 118
StringWriter 213
Subklasse 49
super 50, 51
Superklasse 49
Supplier 192
Swing 279
switch 26
Symbolleiste 337

- Synchronisation 255, 264
synchronized 57, 256, 258
System 140
System Properties 141
System.err 209
System.in 209
System.out 209
- T**
- Tabelle 355
TABLE_PER_CLASS 500, 505
TableModel 355
TableModelEvent 357
TableModelListener 357
TCP/IP 411
terminal operation 200
Textkomponente 321
this 42, 46, 51
Thread 247, 248
throw 100
Throwable 98, 99
throws 100
tiefe Kopie 130
Time Slicing 247, 252
TimeZone 156
TitledBorder 302
Toolkit 283, 383
Tooltip 308
toString 114
transient 57
TreeMap 186
true 13
try 102
try with resources 216
Typargument 170
Typebound 172
Type-Erasure 174
Types 408
Typ-Inferenz 180, 194
Typparameter 169, 170
typsicher 81
Typumwandlung 17, 24
- U**
- Überladen 43
Überschreiben 49
UCanAccess 396
Umgebungsvariable 141
Umlenkung 209
Unicode 11
unidirektional 473
Uniform Resource Locator 145, 411
Unterbrechungssignal 251
Untermenü 336
Upcast 53
update 406
Updatetiefe 545
Upper-Typebound 177
URI 456
URL 145, 411
URLDecoder 421
UTF-8 218, 219, 222
- V**
- Varargs 79
Variable 13, 36
Variablendefinition 16
Vector 133, 183
Vererbung 49
Verhalten 36
Verzweigung 25
View 281
virtuelle Maschine 3
void 40
volatile 57, 254
Vordergrundfarbe 288
- W**
- Wahrheitswert 13
wait 264, 265
while 27
Wildcard-Typ 176
WindowAdapter 292, 294
WindowBuilder 279
WindowEvent 294
WindowListener 290, 294

- Wrapper-Klasse 120
Writer 212, 214
- Z**
Zeichen 14
Zeichenkette 16
Zeichenmethode 287
Zeichenstrom 212
Zeichtyp 14
ZIP-Dateiformat 240
- ZipEntry 240
ZipFile 241
ZipInputStream 242
ZipOutputStream 240
Zufallszahl 150, 152
Zugriffsrecht 57
Zustand 36
Zuweisungsoperator 22